UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ELIAS TEODORO DA SILVA JÚNIOR

Middleware Adaptativo para Sistemas Embarcados e de Tempo-real

Tese apresentada como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação

Prof. Dr. Flávio Rech Wagner
Orientador

Prof. Dr. Carlos Eduardo Pereira
Co-orientador

Porto Alegre, abril de 2008.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Silva Júnior, Elias Teodoro da

Middleware Adaptativo para Sistemas Embarcados e de Tempo-real / Elias Teodoro da Silva Júnior – Porto Alegre: Programa de Pós-Graduação em Computação, 2008.

127 f.:il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2008. Orientador: Flávio Rech Wagner; Co-orientador: Carlos Eduardo Pereira.

1.Aplicações embarcadas 2.*Middleware* 3.Sistemas de tempo-real 4.MPSoCs 5.Eficiência energética. I. Wagner, Flávio R. II. Pereira, Carlos E. III. Título.

AGRADECIMENTOS

Ao povo brasileiro, que financiou este trabalho através do CNPq e do Instituto de Informática da UFRGS.

Aos professores e demais funcionários do Instituto de Informática, em especial aos meus orientadores (Flávio R. Wagner e Carlos E. Pereira), pelas instruções nestes anos de convivência e pelas contribuições à execução deste trabalho.

A todos os colegas do LSE – Laboratório de Sistemas Embarcados, pelas idéias e sugestões, pela ajuda com as ferramentas de desenvolvimento e simulação, pela companhia na hora do almoço.

Aos engenheiros Leonardo Kunz e Raphael Brum (ex-bolsistas de Iniciação Científica), que ajudaram implementando partes do *middleware*.

Ao CEFET-CE pelo seu programa de incentivo à capacitação de seus docentes, que permitiu a minha dedicação integral ao doutorado.

A Ruth, minha esposa, e a André e Alícia, meus filhos, que pacientemente suportaram as minhas ausências nestes anos, motivadas pelo trabalho de doutorado.

Aos meus pais, pelo exemplo e dedicação e por terem investido na minha formação.

A todos os amigos e familiares que me incentivaram e apoiaram, quer com palavras quer com atitudes.

E principalmente a Deus, fonte de toda a sabedoria, aquele que me socorre e fortalece nas horas difíceis.

O meu mais sincero MUITO OBRIGADO!!!

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	Application Program Interface
ASIC	Application Specific Integrated Circuits
ASIP	Application Specific Instruction set Processors
CAN	Controller Area Network
CLDC	Connected Limited Device Configuration
CORBA	Common Object Request Broker Architecture
DRE	Distributed Real-time Embedded
DSP	Digital Signal Processing
DVS/ DFS	Dynamic Voltage Scaling / Dynamic Frequency Scaling
EDF	Earliest Deadline First
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
Hw	Hardware
IP	Intellectual Propriety
JVM	Java Virtual Machine
KVM	Kilobyte Virtual Machine
MPRE	Multiprocessing Real-time Embedded
MPSoC	MultiProcessor System-on-a-Chip
NoC	Network on Chip
ORB	Object Request Broker
PDA	Personal Digital Assistant
QoS	Quality of Service
RISC	Reduced Instruction Set Computer
RM	Rate Monotonic
RMI	Remote Method Invocation
RTC	Real Time Clock

RTL	Register Transfer Level
RTOS	Real-Time Operating System
RTSJ	Real-Time Specification for Java
SoC	System-on-a-Chip
Sw	Software
TCP/IP	Transmission Control Protocol / Internet Protocol
TLM	Transaction Level Modeling
VHDL	Very High Speed Integrated Circuit (VHSIC) Hardware Description Language
VLIW	Very Long Instruction Word
WCET	Worst Case Execution Time

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Um dos principais desafios no desenvolvimento de ferramentas e metodologias para sistemas multiprocessados, embarcados e de tempo-real é o reuso de software já desenvolvido, mantendo baixa utilização de recursos como memória, energia e desempenho de CPU, e ainda atendendo às restrições temporais.

O presente trabalho procura atacar este problema no nível do *middleware*, comumente utilizado como forma de integrar componentes de software reusáveis, diminuindo o tempo e o esforço desprendido no desenvolvimento de aplicações e serviços com alta qualidade.

Este trabalho especifica e implementa um *middleware* para uma plataforma MPSoC voltada para sistemas embarcados e de tempo-real, permitindo adaptações durante o projeto e/ou execução da aplicação, a fim de otimizar o uso dos recursos e atender às restrições de projeto. Ao projetista da aplicação é permitido reusar os serviços do *middleware* e da plataforma em diferentes aplicações. Igualmente, aplicações escritas sobre o *middleware* podem ser portadas para outras plataformas onde o *middleware* possa ser executado.

O *middleware* proposto oferece serviços implementados em hardware e encapsulamento da comunicação hardware-software na própria aplicação. Além disso, são oferecidos meios para gerenciamento de requisitos não funcionais de energia e tempo-real, como *deadline* e tempo de execução.

Palavras-Chave: Aplicações embarcadas, *Middleware*, Sistemas de tempo-real, MPSoCs, Eficiência energética, Reuso.

Adaptive Middleware for Real-Time Embedded Systems

ABSTRACT

One of the main challenges in the development of tools and methodologies for a multiprocessor real-time embedded system is to reuse already developed software, but at the same time obtaining low memory footprint, low energy consumption, and minimal area, obviously addressing the real-time constraints.

This work aims at facing these problems at the middleware level, frequently used to integrate components of reusable software, accelerating development cycle and reducing the effort to develop applications and services with high quality.

The present work specifies and implements a middleware for an MPSoC platform oriented to real-time and embedded systems, providing adaptations at development and execution time, in order to optimize resources usage and fulfill design restrictions. The designer can reuse middleware services and the platform as well, when developing different applications. Likewise, applications developed under the middleware can be ported to run in other platforms where the middleware was ported to.

The proposed middleware offers hardware implemented services and encapsulates hardware-software communication in the application. Moreover, it permits to specify non-functional requirements of energy and real-time, as deadline and execution time.

Keywords: Embedded applications, Middleware, Real-time systems, MPSoCs, Energy efficiency, Reuse.

1 INTRODUÇÃO

Sistemas computacionais embarcados raramente são usados ou percebidos como computadores. Entretanto, eles têm sido responsáveis pela crescente disseminação de tecnologias de informação e de comunicação nas atividades humanas. Alguns exemplos de aplicações embarcadas são:

- Eletrônica de consumo: câmeras, fornos microondas, tocadores de CD/DVD, receptor de TV.
- Telecomunicações: telefones celulares, centrais telefônicas (PABX).
- Veículos: controle da injeção de combustível, freio antitravamento (ABS), câmbio automático.
- Controle industrial: robôs, esteiras de transporte, sistema de supervisão da planta, instrumentos de medição.

O projeto de sistemas embarcados integrados e complexos (CARRO, 2003) é um dos grandes desafios da atualidade para aquelas corporações que atuam no campo de microeletrônica e sistemas eletrônicos. Embora se possa identificar algumas características que são comuns a todos os sistemas embarcados, eles estão longe da homogeneidade que pode ser encontrada nos computadores de propósito geral. Para cada aplicação, pode existir uma combinação muito distinta de módulos de hardware e software que melhor atenda a requisitos como desempenho, determinismo, custo, consumo de energia, confiabilidade e tamanho.

Em muitas aplicações, a integração de todo o sistema em um único chip (SoC – *System-on-a-Chip*) se faz necessária. Existem situações, onde os requisitos de tamanho, potência e desempenho são muito restritos, o projeto de SoCs na forma de um ASIC (*Application-Specific Integrated Circuit*) pode ser imperativo, elevando os custos de projeto e fabricação. Em outras situações, o sistema pode ser implementado em um FPGA (*Field Programmable Gate Array*), uma alternativa que é mais econômica para produtos com menor volume. Resta ainda a possibilidade de solução na forma de um software executando sobre um processador comercial, que já integra milhões de dispositivos e tem um baixo custo em virtude da sua produção em massa. Qualquer combinação destas estratégias pode ainda ser adotada como solução de implementação de um sistema.

A arquitetura de hardware de um SoC embarcado pode conter um ou mais processadores (MPSoC – *MultiProcessor SoC*), memórias, interfaces e blocos

dedicados. Componentes são interconectados por uma estrutura de comunicação que pode variar desde um barramento único até uma complexa NoC (*Network-on-Chip*) (BENINI, 2002). De acordo com a aplicação, os processadores podem ser de tipos diferentes (RISC, VLIW, DSP, ou mesmo ASIPs – *Application-Specific Integrated Processors*) e freqüentemente ocorrem combinações heterogêneas deles.

Por outro lado, sistemas embarcados e de tempo-real utilizando multiprocessamento (MPRE) vêm se tornando largamente utilizados em algumas aplicações (GILL, 2003). A denominação DRE (*Distributed Real-time Embedded*) surge, particularmente, em sensoriamento remoto e telemetria, ou mesmo em sistemas de controle automotivo e de aeronaves. DREs são freqüentemente associados com sistemas distribuídos conectados via redes de comunicação localizadas fora dos chips que implementam os processadores. Entretanto, os princípios de sistemas conectados por redes, encontrados em sistemas DRE, podem ser reusados quando a rede é intra-chip. Neste caso, uma plataforma MPSoC pode fazer bom uso do conhecimento adquirido no contexto de sistemas DRE. Plataformas MPSoC são mais freqüentemente propostas para aplicações como multimídia (em função do processamento digital de sinais), celulares, PDAs (*Personal Digital Assistant*) e câmaras digitais, sendo que muitos destes domínios de aplicação requerem propriedades de tempo-real.

Muitas destas aplicações embarcadas (em especial as multimídia) estão voltadas para o mercado de consumo e, portanto, estão sujeitas a duas pressões simultâneas: a redução do tempo desde a idéia até a disponibilidade do produto (*time-to-market*) e sua crescente complexidade (SANGIOVANNI-VICENTELLI, 2004). Para atender a estes e outros requisitos, muitas pesquisas têm se realizado. Em particular, o suporte do *middleware* tem sido investigado no contexto de aplicações DRE (SCHMIDT, 1999) (KLEFSTAD, 2002) (KRISHNA, 2003). Todavia, também no contexto de MPSoCs um *middleware* poderia ser uma solução para elevar o nível de abstração, ajudando a alcançar uma redução nos tempos de desenvolvimento de aplicações. Existe uma grande variedade de aplicações embarcadas que demandam o uso de um sistema distribuído, e do *middleware*, conseqüentemente. Dentre elas se pode citar: sistemas de automação residencial, sistemas de automação industrial, meios de transporte, TV digital e processamento digital de sinais. Alguns destes exemplos requerem distribuição pela natureza da aplicação, como em meios de transporte se faz a descentralização dos processadores para aumentar a confiabilidade ou reduzir o cabeamento. Em outras aplicações se faz a distribuição para aumentar a eficiência, como em processamento digital de sinais.

Uma vez que a aplicação é particionada em diversos processos e processadores um *middleware* pode ser usado para criar uma interface de alto nível, tornando aspectos de distribuição transparentes às aplicações, ou seja, para as aplicações é como se houvesse um único processador (BERNSTEIN, 1993). Desta forma, partes da aplicação ou recursos do sistema podem ser reusados mais facilmente, diminuindo o tempo gasto no projeto de novas aplicações.

Sabe-se que as decisões de projeto tomadas em alto nível de abstração resultam em maiores ganhos em termos de recursos como memória, tamanho, velocidade, etc. em face às tomadas em baixo nível (MATTOS, 2004). Neste sentido, alguns trabalhos vêm sendo propostos para viabilizar a exploração do espaço de projeto ainda no nível do modelo, onde se sabe que ganhos maiores podem ser obtidos (OLIVEIRA, 2006).

Ocorre que, para que os objetos possam ser livremente movimentados entre implementações em hardware ou software, ou ainda, no caso de implementações em software, entre acesso local (no mesmo processador) ou remoto, é preciso que a estratégia de comunicação entre eles seja flexível e abstraia a localização do objeto alvo, bem como a sua implementação em hardware ou em software. É necessário que os objetos que compõem a aplicação possam se localizar e se comunicar com o menor grau de dependência possível, dando maior flexibilidade ao sistema de exploração de espaço de projeto e liberando o projetista de detalhes inerentes ao processo de comunicação.

Estas demandas também apontam para a necessidade de uma camada intermediária (o *middleware*) que permita ao projetista adiar a decisão de localidade e estratégia de implementação do objeto de maneira que a exploração fique facilitada.

A exploração da localização do objeto vai permitir o balanceamento da carga em uma rede de processadores visando à minimização de custos e atendimento a restrições de tempo de execução da aplicação. Quando um recurso de comunicação é um barramento simples pode-se explorar o particionamento hardware-software e/ou a distribuição da carga. Por outro lado, recursos de comunicação como NoC e barramentos segmentados ou hierárquicos permitem ainda explorar diferentes arranjos de mapeamento dos objetos (MARCON, 2005).

A decisão de posicionamento final dos objetos poderá ser orientada por energia, desempenho, ou por outro parâmetro. Esta é uma opção do projetista especificada junto à ferramenta que faz esta exploração. Em (OLIVEIRA, 2006) é apresentada uma abordagem de exploração de projeto que depende desta transparência nas interconexões entre os objetos que compõem a aplicação.

Os desafios relacionados com o desenvolvimento de um *middleware* em sistemas embarcados e de tempo-real são: (1) reusar a infra-estrutura (de software e de hardware) já existente, (2) apresentar baixo consumo de recursos, como memória e tempo de processamento e (3) oferecer mecanismos de controle de tempo-real. Além disso, um (4) baixo consumo de energia no sistema como um todo é uma característica importante. Deste último decorre a necessidade de que o *middleware*, ou o sistema operacional, ofereça(m) (5) mecanismo de controle/adaptação da infra-estrutura para gerenciar o consumo de energia.

Um dos desafios de desenvolver um sistema MPRE que atenda a estas restrições decorre da limitação dos recursos oferecidos nas plataformas utilizadas, que geralmente são elementares, contendo CPUs com capacidade limitada, pouca memória, banda de comunicação estreita e fonte de energia pequena. Por outro lado, a pouca flexibilidade que as plataformas oferecem para uso e configuração dos recursos de hardware limita a exploração do espaço de projeto.

Além de oferecer primitivas básicas de comunicação, um *middleware* pode contribuir efetivamente de duas maneiras no desenvolvimento de aplicações sobre plataformas MPSoC. Em tempo de projeto, o *middleware* torna o acesso a um objeto independente de sua localização ou mesmo de sua implementação em hardware ou em software. Esta elevação no nível de abstração vai reduzir o tempo de desenvolvimento de aplicações e simplificar o trabalho do projetista em duas tarefas importantes: (1) Reusar componentes de hardware e software desenvolvidos em projetos anteriores e (2) explorar as diversas possibilidades de implementação visando a ganhos em previsibilidade, energia, desempenho, tamanho etc.

Já em tempo de execução, o *middleware* pode gerenciar a distribuição da aplicação oferecendo serviços adaptáveis em função de restrições de energia ou desempenho, por exemplo. O acoplamento fraco entre os objetos da aplicação, provido pelo *middleware* em tempo de projeto, vai permitir que estes objetos sejam realocados dinamicamente ou até mesmo que a opção por uma implementação em hardware ou em software seja feita durante a execução da aplicação. Por localizar-se logo abaixo da camada de aplicação, o *middleware* pode monitorar o comportamento da aplicação com mais facilidade e providenciar as adaptações necessárias na plataforma ou nos seus serviços.

Finalmente, em função da limitada capacidade computacional encontrada nas plataformas utilizadas em aplicações embarcadas, os serviços do RTOS (*Real time Operating System*) e do *middleware* tendem a se misturar em um pacote visando a diminuir o *overhead* que um elevado número de camadas poderia causar.

1.1 Objetivos e delimitação do objeto de estudo

O principal objetivo deste trabalho é dar transparência no acesso a objetos tanto na localização quanto na forma de implementação (hardware/software), deixando esta decisão para a fase final do projeto ou para o tempo de execução.

O *middleware* deste trabalho conecta os componentes de uma aplicação embarcada distribuída dentro de chips multiprocessados (MPSoC), utilizando redes em chip (NoCs) ou estruturas mais simples, como barramento. Além disso, componentes em hardware da aplicação ou do *middleware* são integrados de forma transparente.

Neste trabalho será usado como plataforma um processador Java, apresentado na Seção 3.1. Nos últimos anos, a linguagem Java vem ganhando popularidade no desenvolvimento de sistemas embarcados e de tempo-real. A definição da especificação Java para tempo-real (RTSJ – *Real-time Specification for Java*) (BOLLELLA, 2000) é o exemplo mais notável de popularização no domínio de tempo-real.

O domínio de aplicação escolhido dá ênfase à economia de energia, atendendo às restrições de tempo impostas pela aplicação. Para enfrentar estes desafios, adaptações devem ocorrer durante a execução da aplicação, atuando sobre as configurações do hardware, ou mesmo decidindo entre implementações hardware ou software de serviços e tarefas. Assim, o *middleware* pode buscar minimizar a energia, baseado em restrições definidas durante o projeto (tempo, potência, desempenho). Para aplicações tempo-real mais críticas, a preferência pode ser dada a atender às restrições de tempo, procurando minimizar a energia. De outro lado, em aplicações onde as restrições de tempo forem moderadas, pode-se definir máximos de energia, admitindo a perda do *deadline* de algumas tarefas. Estes parâmetros são informados aos escalonadores de tarefas, que fazem parte do pacote *middleware*/RTOS.

Adaptabilidade em *middleware* não é uma novidade, particularmente no domínio de aplicações móveis e embarcadas. Alguns *middlewares* de propósito geral oferecem meios para que a aplicação se adapte ou ele mesmo faz adaptações, mas sempre atuando no software. Este trabalho, entretanto, oferece meios para que estas adaptações ocorram também com interferência no hardware do sistema.

1.2 Contribuições e inovações

A abordagem desta proposta tem alguns pontos de similaridade com a que é apresentada em (PAULIN, 2004), tendo, entretanto, a preocupação de permitir atuação em aspectos não funcionais, como tempo-real e energia, e provendo mecanismo de gerenciamento para estes mesmos requisitos. Particularmente o requisito de energia é atacado por dois mecanismos. Um deles é a adaptação no hardware, estratégia ainda não utilizada nos trabalhos da comunidade de *middlewares* para sistemas embarcados. O outro mecanismo para obter baixa energia está na realocação (ou migração) de objetos na rede, tópico ainda pouco explorado em MPSoCs.

Assim como este, outros trabalhos abordam a utilização de Java em tempo-real para aplicações distribuídas (BORG, 2003) (KRISHNA, 2003). Entretanto, nenhum deles tem o olhar sobre plataformas MPSoC, mas somente sobre redes convencionais.

Finalmente, este trabalho, ao elevar o nível de abstração no desenvolvimento de aplicações embarcadas multiprocessadas (SoCs e NoCs), oferece meios para que se faça uma exploração do espaço de projeto em alocação de objetos distribuídos. Além disso, adaptações não funcionais no nível do *middleware* permitem melhorar a eficiência energética de aplicações embarcadas de tempo-real.

Adicionalmente, são oferecidas contribuições na pesquisa em *middlewares*, apresentando meios para que adaptações ocorram também com interferência no hardware do sistema. Tradicionalmente os *middlewares* adaptativos atuam no software, da aplicação ou deles mesmos.

O *middleware* poderá ser utilizado em outro domínio de aplicação onde existam computadores embarcados e distribuídos. Suas propriedades de tempo-real e de baixa energia poderão ser aproveitadas em dispositivos móveis ou em aplicações de controle, notadamente em redes de sensores, onde a capacidade computacional requerida dos nodos da rede seja baixa.

Este trabalho partiu de uma implementação prévia da especificação RTSJ (WEHRMEISTER, 2004), que oferece uma base semelhante a um RTOS. A esta base foi acrescentado um serviço de comunicação entre processadores que utiliza troca de mensagens (SILVA JÚNIOR, 2006a). Um mecanismo de escalonamento dinâmico de freqüência e tensão (DVS - *Dynamic Voltage Scaling*) foi também adicionado ao escalonador de tarefas da implementação RTSJ inicial. Ainda neste nível de base foram propostos e implementados serviços do *middleware* em hardware (SILVA JÚNIOR, 2005b), (SILVA JÚNIOR, 2007b), (KUNZ, 2007).

Em mais alto nível o presente trabalho oferece mais duas contribuições originais: objetos da aplicação podem ser implementados em hardware (SILVA JÚNIOR, 2008) e tarefas em software podem migrar de um processador para outro. A migração de tarefas toma como base o trabalho de Barcelos (2008) e eleva o nível de abstração daquela proposta. Adicionalmente, um serviço de invocação remota de métodos com restrições tempo-real foi implementado e um serviço de alocação de tarefas foi especificado.

1.3 Visão geral do texto

O restante deste texto está organizado em mais cinco capítulos. O capítulo 2 apresenta os principais trabalhos em *middleware* para aplicações de tempo-real e embarcadas, bem como as restrições que norteiam este tipo de projeto. No capítulo 3 a plataforma de desenvolvimento e experimentação é descrita, detalhando suas propriedades e restrições. O capítulo 4 apresenta a arquitetura do *middleware* deste trabalho juntamente com a descrição dos seus serviços. No capítulo 5 são dados exemplos de utilização dos serviços do *middleware*, assim como resultados de simulação. Finalmente, o capítulo 6 traz as considerações finais sobre o trabalho e perspectivas de continuidade.

2 ESTADO DA ARTE

O *middleware* clássico tem como propósito integrar componentes de software reusáveis para diminuir o tempo e o esforço despendido no desenvolvimento de aplicações e serviços com alta qualidade. O *middleware* também fornece uma visão uniforme de redes, protocolos e recursos de sistemas operacionais heterogêneos, ajudando a simplificar o desenvolvimento de aplicações *multithread* e multiprocessador.

Duas propriedades podem ser agregadas pelo *middleware* a uma plataforma. São elas programabilidade, que se refere à habilidade de descrever um problema para uma máquina de uma maneira conveniente e eficiente, independente da configuração da máquina, e extensibilidade, que é a facilidade de inserção de novos módulos ou propriedades no sistema.

2.1 Os problemas do *Middleware* para sistemas embarcados

No contexto de sistemas embarcados e de tempo-real, os aspectos que vão nortear a especificação do *middleware* são:

1) Limitada capacidade computacional. Os nodos da rede são dispositivos elementares com recursos restritos de CPU, memória, banda de comunicação e autonomia de baterias limitada. Assim, é preciso que o *middleware* gere a menor carga possível para este sistema e que procure otimizar o uso destes recursos.

2) Atendimento a restrições de tempo. Sistemas de tempo-real dependem não somente dos resultados lógicos da computação, mas também do tempo de ocorrência destes resultados (STANKOVIC, 1996). Não necessariamente o melhor desempenho precisa ser perseguido, mas aquele que permita satisfazer a especificação de tempo, apresentando comportamento previsível, determinístico.

3) Heterogeneidade de plataformas. O hardware das estações é muito simples e seus recursos podem variar quanto aos padrões e protocolos de rede. Há também a diversidade de sistemas operacionais e de processadores, que geralmente são distintos daqueles encontrados em estações de propósito geral. Além disso, estes dispositivos têm recursos computacionais diferentes, requerendo do *middleware* um certo grau de flexibilidade e de configurabilidade. Em sistemas discretos, que usam redes inter-dispositivo, os protocolos são padronizados e diversos. Em um carro, por exemplo, pode-se usar uma interface no nível físico conectada por fios, como o CAN-bus. Já para PDAs ou em domótica são mais usuais as interfaces sem fio, como o Bluetooth, com outro conjunto de protocolos. Por outro lado, para as redes usadas em MPSoCs (intra-

chip) os protocolos não são padronizados. Os trabalhos já realizados discutem superficialmente os protocolos de comunicação (BENINI, 2002) (GOOSSENS, 2002) ou implementam apenas as camadas de mais baixo nível (DALLY, 2001) (WINGARD, 2001). Em (MILLBERG, 2004) é proposto um protocolo de pilhas, à semelhança do modelo OSI, indo da camada física até a de transporte. O trabalho realizou uma experimentação na forma de simulação, que se concentrou na camada de rede.

Bray (1997) define *middleware* como sendo um software de conectividade composto de um conjunto de serviços que permitem que múltiplos processos, sendo executados em um ou mais processadores, possam interagir através da rede. Classicamente, o papel do *middleware* é abstrair a complexidade da distribuição da aplicação e/ou serviços em uma rede (BERNSTEIN, 1993). Ele é um conjunto de softwares localizado entre as aplicações e o sistema operacional junto com a infra-estrutura de rede. Ocorre que, em função das limitações de recursos computacionais de uma plataforma embarcada, é preciso buscar um compromisso entre a transparência total oferecida pelo conceito de *middleware* oriundo de sistemas distribuídos clássicos e a eficiência necessária ao sistema embarcado, de maneira a assegurar menor custo e melhor uso dos recursos da plataforma. Assim, é comum em sistemas deste porte a sobreposição dos conceitos de *middleware* com o de sistema operacional, oferecendo os dois tipos de recursos em um só pacote. No restante deste texto, será usado o termo "*middleware*" para identificar o conjunto de recursos propostos nesta tese, apesar deste conjunto apresentar diferenças em relação ao conceito clássico de *middleware* advindo da área de sistemas distribuídos.

Por outro lado, em um ambiente de multiprocessadores embarcados é freqüente se encontrar uma rede inter-dispositivo, comum no domínio de sistemas distribuídos, onde os processadores estão em chips diferentes e quase sempre se usa uma arquitetura tipo barramento. Porém, com o aumento da complexidade dos sistemas embarcados e com a necessidade de reduzir a energia gasta nos processadores, as redes intra-dispositivo (BENINI, 2002) ou intra-chip começam a se tornar uma realidade.

Estes dois tipos de redes de comunicação têm muito em comum, mas guardam algumas particularidades.

Nas redes inter-dispositivo a maior parte dos protocolos (camadas mais altas) é executada em software. Entretanto, para sistemas onde a energia é crítica, executar estes protocolos em circuitos dedicados pode ser mais conveniente.

As redes intra-chip podem ainda explorar um número maior de conexões físicas. Em função das curtas distâncias, arquiteturas alternativas passam a ser viáveis, como é o caso das malhas, propostas para NoCs. Além disso, a comunicação serial deixa de ser imperativa, permitindo ainda múltiplos canais de comunicação.

Trabalhos mais recentes, como (MARTIN, 2006) e (JERRAYA, 2006) são fortes indicadores de tendência, apontando o uso de MPSoCs como plataforma para aplicações embarcadas e a necessidade de elevar o nível de abstração em projetos envolvendo MPSoCs. Martin (2006) faz considerações quanto aos desafios no projeto e Jerraya (2006) discute modelos de programação e interfaces para abstração da comunicação hardware e software.

Muitas aplicações de tempo-real rodam em sistemas embarcados. Por esta razão é comum se encontrar trabalhos que abordam os dois tipos de aplicação em conjunto. O termo Sistema DRE (*Distributed, Real-time and Embedded*) é freqüentemente usado

para se referir a aplicações ligadas às duas áreas. Neste trabalho será considerado que o *middleware* atenderá a aplicações embarcadas e de tempo-real, com ênfase na eficiência energética.

2.2 Exemplos de *middleware* para sistemas embarcados

Middlewares com foco em sistemas embarcados conectados e de pequeno porte computacional (redes inter-dispositivos) são tratados em diversas iniciativas. MinimumCORBA (OMG, 1998) e Java 2 Micro Edition (SUN, 2001) são especificações que vão na linha de reduzir a sobrecarga de memória presente nos padrões dos quais eles são originários. O minimumCORBA, por exemplo, deriva do padrão CORBA para sistemas distribuídos de maior porte. Trata-se de um padrão que pode ser adotado por qualquer implementação em particular. O J2ME é uma formatação de Java visando aos dispositivos com menor capacidade computacional que, através da CLDC (*Connected Limited Device Configuration*), oferece meios para que seja implementada uma JVM de baixo consumo de memória, chamada KVM. Pelo seu custo em memória e recursos computacionais, estas soluções estão longe de poder se adequar ao tipo de plataforma pretendido para o escopo deste trabalho (MPSoCs).

Algumas soluções proprietárias baseadas em Java oferecem otimizações de código voltadas para sistemas embarcados. É caso do Jeode, da Esmertec (2003), que faz um misto de compilação e interpretação, chamado compilação adaptativa, para obter um código mais rápido e menor.

ZEN (KLEFSTAD, 2002) é um *middleware* de código aberto, desenvolvido em linguagem Java, que é voltado para atender às necessidades das aplicações DRE. Ele procura atender às especificações CORBA, que por sua vez estão mais ligadas aos problemas de aplicações em tempo-real do que aos de sistemas embarcados. A redução no gasto de memória no ZEN é obtida pelo que o autor chama de micro-ORB dinâmico. Neste caso um ORB (*Object Request Broker*) mínimo é implementado, mas com a capacidade de ser ampliado ou modificado em tempo de execução. A compatibilidade com CORBA impõe serviços que fazem com que os custos em desempenho e memória fiquem elevados, porém assegura alguma interoperabilidade. Existe uma versão tempo-real, chamada RT-ZEN (KRISHNA, 2003), que pode ser executada sobre uma JVM convencional ou sobre uma plataforma RTSJ (*Real-Time Specification for Java*) (BOLLELLA, 2000) compilada e interpretada.

DynamicTAO (KON, 2000) e LegORB (ROMAN, 2000) são *middlewares* reflexivos voltados para sistemas de pequena capacidade computacional. A reflexão permite que o *middleware* possa, dinamicamente, em pleno funcionamento, adaptar-se a mudanças de contexto, tais como limitações de recursos ou restrições de segurança. Para isso ele faz monitoramentos em parâmetros estabelecidos pela aplicação. O *middleware* procura, assim, automaticamente, minimizar o seu tamanho. A reflexão também poderia ser explorada em tempo de projeto como forma de ter um *middleware* ainda mais otimizado, ainda que com menor flexibilidade. Entretanto, os protótipos usando a abordagem de reflexão apresentam um custo em memória e de desempenho que pode ser inviável para plataformas realmente pequenas. Os experimentos são conduzidos em PCs portáteis ou em PDAs. Em particular o LegORB (versão dinâmica) foi implementado em um Handheld PC Jornada 680 com processador SH3 a 133MHz e 16MB de RAM executando o WindowsCE.

Embora exista uma quantidade significativa de trabalhos envolvendo *middleware* para aplicações embarcadas e/ou de tempo-real, eles não consideram o uso de plataformas MPSoC. Alguns trabalhos se preocupam com eficiência energética, porém concentram-se nos níveis mais baixos de abstração. Também não se encontram trabalhos que facilitem a interação com serviços implementados em hardware, destacada a exceção do RCSM, descrito em (YAU, 2002). Trata-se de um *middleware* focado em aplicações que requerem consciência do contexto e comunicação *ad hoc*. Todavia, embora o RCSM implemente parte de seus serviços em hardware (FPGA), ele não se propõe a dar suporte ao projeto de sistemas embarcados, mas ao desenvolvimento de aplicações tolerantes a variações no seu contexto. A abordagem de construir serviços em hardware poderia ter sido usada para oferecer ganhos em energia, mas o trabalho não tinha como objetivo atender a esta restrição.

Todos estes trabalhos se destinam a redes inter-dispositivo. E mesmo quando atendem a dispositivos embarcados eles são do porte de PDAs, com recursos limitados, mas não tanto quanto uma rede de sensores, por exemplo. Por isto, estes trabalhos dificilmente atenderiam às necessidades de comunicação intra-chip ou mesmo de dispositivos menores, como é o caso de redes de sensores.

2.2.1 *Middleware* para redes de sensores

O domínio de aplicação 'redes de sensores' merece um destaque, por ser uma categoria de sistema embarcado em que as restrições de energia são tão fortes que determinam todo o restante dos requisitos. Para um equipamento portátil o usuário, mesmo que fique privado do serviço por um tempo, sempre pode recarregar as baterias. Já para um dispositivo de uma rede de sensores, o fim da bateria significa, em muitos casos, o fim da vida do equipamento.

Em (ROMER, 2004) é apresentada uma lista com os requisitos que inviabilizam o reaproveitamento de *middlewares* clássicos para este domínio de aplicações. São eles: (1) Paradigma de programação – voltado para monitorar uma grande quantidade de variáveis ambientais; (2) Recursos restritos – em função das limitações de energia já comentadas e da necessidade de reduzir custos; (3) Redes dinâmicas – o posicionamento dos nodos não é conhecido a priori e, em geral, pode mudar a qualquer instante; (4) Escala – a quantidade de nodos pode chegar a milhares de unidades, em função de necessidades da aplicação; (5) Integração com o mundo real – muitas aplicações requerem correlação entre o tempo e o local da leitura, ou ainda a sincronização entre diferentes nodos; e (6) Suporte do sistema operacional – os *middlewares* para redes de sensores vão estar sobre um sistema operacional com bem poucos recursos ou sobre nenhum sistema operacional, diferentemente do que ocorre normalmente em outras plataformas. É ainda necessária uma (7) integração com uma estrutura de apoio, que recebe os dados coletados.

Sendo assim, os desafios para o *middleware* neste contexto de aplicações embarcadas têm peculiaridades bem específicas e requerem uma abordagem particular. Em resposta a este problema, diversas propostas têm sido feitas, tanto no nível do *middleware* quanto de sistemas operacionais (MADDEN, 2002) (SHEN, 2004) (BOULIS, 2003) (LI, 2003) (BARR, 2002).

O *middleware* proposto aqui não pretende atacar os problemas acima de maneira abrangente. Entretanto, as características de números (2) e (6) são válidas para outros sistemas embarcados que pretendem ser eficientes energeticamente, ainda que não tão intensamente quanto as redes de sensores. Estas duas propriedades estão presentes na especificação do *middleware* deste trabalho.

2.3 *Middleware* para redes intra-chip

Para fazer um *middleware* viável para comunicação intra-chip, os protocolos e serviços utilizados em redes de computadores tradicionais não podem ser usados diretamente; eles precisam ser modificados de modo que o custo de implementação assim como a sua taxa efetiva de comunicação (*throughput*) sejam aceitáveis, sem perder de vista as restrições já impostas às aplicações embarcadas. Estas modificações podem se dar do ponto de vista funcional, implementando um conjunto limitado de funções, tanto nos serviços quanto nos protocolos. Complementarmente, modificações podem ocorrer de forma não funcional, implementando funções em hardware, ou utilizando estrutura de comunicação que ofereça maior paralelismo, alcançando melhor desempenho e/ou previsibilidade.

Redes intra-chip são diferentes de redes convencionais (inter-dispositivos) pela proximidade dos seus nodos e por apresentarem maior determinismo (BENINI, 2002). Preocupação com restrições de energia e possibilidade de especializações em tempo de projeto são mais comuns em redes intra-chip.

Por outro lado, as redes maiores priorizam comunicação de propósito geral e modularidade. O projeto da rede tradicionalmente é desacoplado da aplicação final e fortemente influenciado pela padronização e pela compatibilidade com uma infra-estrutura de rede legada. No projeto de uma rede para MPSoC estas restrições não estão presentes, já que o sistema final vai ser sintetizado do zero e o reuso não será de componentes prontos, mas de sua representação em alguma linguagem de descrição de hardware (HDL). Assim, do ponto de vista da aplicação, somente a interface de mais alto nível precisa ser padronizada e o projetista tem liberdade para fazer modificações nas diversas camadas da infra-estrutura de comunicação.

Outro aspecto já levantado anteriormente é que, em função da necessidade de reduzir a energia, os protocolos de comunicação poderão ser implementados em hardware. Não somente os protocolos como quaisquer outros serviços poderão ser em hardware. Isto vai demandar do *middleware* a possibilidade de fazer a interface entre operações implementadas em hardware com operações implementadas em software. Desta forma, um de seus serviços deverá ser o de permitir que a chamada a uma operação não seja dependente de sua implementação estar em hardware ou em software. O trabalho do Nostrum Backbone (MILLBERG, 2004) sugere uma RNI – *Resource Network Interface* – que funcionaria como um *Wrapper*, permitindo que um componente IP tenha acesso à rede. Entretanto, como já foi dito, os estudos publicados se limitam a um protocolo no nível de rede. Além de não ser oferecida uma estratégia de conexão entre os nodos comunicantes, a abstração oferecida pelo nível de rede é dependente do recurso e da plataforma, o que limita a exploração do espaço de projeto e a adaptação dinâmica da plataforma.

Recentemente, as questões relacionadas ao modelo de programação para MPSoCs têm sido abordadas em alguns trabalhos. Destaca-se o MultiFlex (PAULIN, 2004), uma metodologia para dar suporte ao desenvolvimento de aplicações sobre uma plataforma MPSoC, chamada StepNP. A plataforma é composta por elementos de processamento em hardware e em software e oferece dois modelos de programação concorrente: passagem de mensagens e memória compartilhada, embora em (PAULIN, 2006) os mesmos autores tenham feito claramente uma opção pelo mecanismo de troca de mensagens. A programação das aplicações é feita em linguagem de alto nível, C ou C++, acessando os serviços através de APIs. Para aumentar o desempenho são implementados em hardware os serviços de troca de mensagens, alocação dinâmica de tarefas e troca de contexto. O foco do MultiFlex é permitir o mapeamento direto dos modelos de programação na plataforma, recorrendo, sempre que preciso, a implementações em hardware para minimizar latências. Em (PAULIN, 2004) uma aplicação exemplo é mapeada em uma rede de 9 processadores ARM utilizando tanto o modelo de troca de mensagens quanto de memória compartilhada. O custo em área dos aceleradores de hardware do MultiFlex é 58K gates e 18K bytes de memória.

Quando do início deste trabalho, eram poucos os esforços de pesquisa em redes intra-chip que consideravam a possibilidade de prover recursos de programação em alto nível. Particularmente, a possibilidade de gerenciamento de restrições de tempo e de energia ainda não foi adequadamente contemplada na literatura. Nesta área, os pesquisadores que trabalham mais próximos do hardware concentram os seus esforços em prover arquiteturas eficientes, mas sem muita preocupação com o modelo de programação. Já os grupos que pesquisam em *middleware* têm somente as redes convencionais como cenário, e quando se voltam para sistemas embarcados e de tempo-real, lidam com plataformas que dispõem de recursos computacionais relativamente abundantes para um sistema embarcado.

2.4 O escalonador de tarefas

O termo ‘tarefa’ deriva da literatura de sistemas operacionais, representando um elemento escalonável no contexto do sistema. Neste trabalho uma tarefa é sinônimo de uma *thread* ou ainda de um objeto escalonável.

Um *middleware* para aplicações de tempo-real precisa ter o suporte de um escalonador de tarefas de tempo-real. Isto porque o gerenciamento da execução das tarefas é um ponto-chave no atendimento das restrições temporais destas. Adicionalmente, alguns trabalhos consideram a ação do escalonador na limitação de energia para aplicações de tempo-real. Em (ALENAWY, 2004) são explorados os problemas de desempenho e viabilidade para sistemas de tempo-real que precisam se manter funcionais durante um determinado tempo, operando com uma cota de energia. Naquele trabalho uma estratégia DVS (*Dynamic Voltage Scaling*) consegue aproveitar os tempos de CPU não usados quando uma tarefa não chega a utilizar todo o tempo de execução previsto na análise de escalonabilidade, geralmente o WCET – *Worst Case Execution Time*.

Por outro lado, no contexto de orientação a objetos, e tomando o ambiente Java como referência, a especificação RTSJ (*Real-Time Specification for Java*) (BOLLELLA, 2000) deve ser considerada. Várias implementações da RTSJ estão

disponíveis, implementando diversos algoritmos de escalonamento, embora nenhuma delas considere restrições de energia. A proposta jRate (CORSARO, 2002) estende o compilador Java GNU (GCJ) e oferece uma versão pré-compilada da JVM. Na verdade, não há JVM nem bytecodes a interpretar, mas a aplicação, juntamente com as classes RTSJ, é compilada em código nativo da arquitetura alvo. Uma linha de trabalho alternativa é usar um processador que execute Java como código nativo. É o caso da API-RTSJ proposta em (WEHRMEISTER, 2004) que é executada sobre o processador FemtoJava (ITO, 2001). Estas duas alternativas de implementação RTSJ são interessantes e podem ser úteis quando o objetivo é ter um baixo consumo de energia. Neste trabalho optou-se por usar esta última plataforma para fazer experimentação.

Usar serviços implementados em hardware, estendendo a parte operativa do processador, por exemplo, é uma abordagem que traz redução na energia usada pelo sistema (AYDIN, 2003). Exemplos que aliam esta abordagem com sistemas de tempo-real são apresentados em (KUACHAROEN, 2003) e em (ANDREWS, 2005), onde serviços do sistema operacional são sintetizados como módulos de hardware em um SoC.

Em (KUACHAROEN, 2003) é apresentada uma implementação em FPGA de um algoritmo de escalonamento parametrizado. Nesta proposta, o algoritmo de escalonamento pode ser trocado em tempo de execução sem a necessidade de reprogramar o FPGA. A estratégia adotada é de implementar alguns algoritmos de escalonamento em um único circuito, os quais, na verdade, compartilham muitos dos componentes de hardware. O algoritmo de escalonamento desejado é selecionado através da manipulação de alguns parâmetros. O foco do trabalho está na flexibilidade na escolha do escalonador e no desempenho oferecido pelo hardware, sendo que aspectos de energia não foram considerados.

Andrews (2004) amplia o conceito de programação *multithread* para incluir componentes de um FPGA, os quais estão conectados ao barramento de uma CPU. O modelo de programação proposto é chamado HybridThreads e permite a especificação de aplicações como um conjunto de *threads* que vão ser materializadas em uma combinação de software e hardware, na CPU e no FPGA. O mesmo grupo de pesquisa apresenta em (AGRON, 2004) o projeto de hardware de um módulo escalonador de tarefas desenvolvido como parte do núcleo (*kernel*) de um sistema operacional. O objetivo é prover uma interface uniforme para ser usada por projetistas e programadores no desenvolvimento de componentes híbridos, ou seja, *threads* de propósito geral que podem ser implementadas tanto em hardware quanto em software, para aplicações embarcadas. Um aspecto interessante deste escalonador é que ele também pode escalonar tarefas em hardware. Um conjunto de registradores de propósito geral é usado para fazer a interface entre o componente hardware do usuário e o restante do sistema. Também neste trabalho, a ênfase está na flexibilidade, desta vez procurando prover meios para simplificar o processo de especificação e desenvolvimento de aplicações. Neste trabalho a plataforma é o que se costuma chamar de *chip* híbrido, uma CPU ligada a um FPGA onde fica a parte (re)configurável do sistema.

2.4.1 Alocação e migração de tarefas

No contexto de plataformas MPSoC existem ainda dois conceitos próximos do escalonamento de tarefas: a alocação e a migração de tarefas.

Alocar significa decidir em qual processador colocar o objeto a partir de uma função de distribuição. Este mecanismo de alocação interage com o escalonamento já que, em uma aplicação com restrições temporais é preciso não somente buscar uma alocação que dê baixa energia, por exemplo, mas também que assegure a escalonabilidade sem perda de *deadlines*.

Quando a decisão de onde alocar um objeto ocorre em tempo de projeto diz-se que a alocação é estática. Quando a decisão de alocação pode ser tomada também em tempo de execução, um mecanismo de migração dos objetos se faz necessário. Da mesma forma que na alocação, a migração também precisa interagir com o escalonador local de tarefas em cada processador. A migração ocorre quando uma tarefa já alocada em um processador e em execução precisa ser movida para outro processador e continuar a executar de lá. As tarefas em execução nos dois lados da migração devem continuar atendendo às suas restrições temporais enquanto a migração ocorre e após a sua conclusão.

Vários trabalhos investigam a alocação de tarefas em MPSoCs, mas na maioria deles as decisões são tomadas em tempo de projeto. Alguns trabalhos, como (WRONSKI, 2006), (BRIÃO, 2008) e (ACQUAVIVA, 2008) já começam a propor estratégias para alocar tarefas nos processadores do MPSoC em tempo de execução, o que vai requerer um mecanismo de abstração da localidade no processo de comunicação.

2.5 Discussão

No desenvolvimento de aplicações embarcadas é freqüente a dificuldade em usar ferramentas de produtividade, mais comuns quando a plataforma alvo é um computador de propósito geral. A maior dificuldade está na pluralidade de plataformas embarcadas e no alto grau de personalização de cada produto.

Este capítulo apresenta as características gerais de um *middleware*, considerando-o como alternativa para aumentar a produtividade no projeto de sistemas embarcados, bem como as restrições que uma rede de sistemas embarcados impõe ao *middleware*. Como agravante, é comum encontrar aplicações embarcadas que devem atender também a restrições de tempo-real, tais como multimídia e controle de veículos.

A Tabela 2.1 mostra uma lista de requisitos desejáveis para um *middleware* no contexto de MPSoC em sistemas embarcados. Para os principais *middlewares* analisados neste capítulo é indicado o atendimento aos requisitos. Para os dois primeiros requisitos a tabela não pode ser vista de maneira binária. O conceito de capacidade computacional limitada, bem como de memória limitada é relativo. Para esta avaliação considerou-se que uma memória superior a 128Kbytes já não é mais limitada. Para os processadores, a freqüência de relógio limite foi definida em 300 MHz.

Dois aspectos importantes se destacam na tabela: (1) nenhuma das abordagens considera o gerenciamento da energia do sistema; (2) a proposta feita especificamente

para MPSoCs, o MultiFlex, além de não contemplar estudos para otimizar a energia, não propõe mecanismo para lidar com aplicações de tempo real.

Tabela 2.1: Comparação dos *middlewares* analisados quanto aos requisitos

Middlewares / Requisitos	RT-ZEN	LegORB	RCSM	MultiFlex (*3)	HybridThreads
Nodos com limitada capacidade computacional		X	X	X	X
Limitada memória de dados e código	X (*1)	X (*2)	X	Não informado	X
Paradigma de orientação a objetos	X	X	X	X	
Expressão de requisitos tempo-real	X				
Mecanismo de controle/limitação de energia					
Implementações hardware na aplicação				X	X
Implementações hardware nos serviços			X	X	X
Adaptação (software)	X	X	X		
Adaptação (hardware)					X

*1 – 98 a 4260 KB
*2 – 20 a 141 KB
*3 – Proposto para MPSoCs

Particularmente, o tipo de rede adotada para estudo neste trabalho é composta de vários processadores em um único *chip*. A plataforma alvo que será utilizada é detalhada no capítulo seguinte, tanto a rede quanto o processador.

3 PLATAFORMA DE EXPERIMENTAÇÃO

Para suporte a aplicações distribuídas foi montada uma plataforma de simulação MPSoC composta de processadores FemtoJava (ITO, 2001) conectados pela NoC SoCIN (ZEFERINO, 2003).

Tanto o processador FemtoJava quanto a NoC SoCIN estão disponíveis em modelos para simulação e em versões VHDL (VHSIC *Hardware Description Language*), que permitem a síntese lógica em um ASIC (*Application-Specific Integrated Circuit*), por exemplo. Neste trabalho a maior parte dos experimentos foi feita utilizando o simulador SERPENS, que será apresentado mais adiante.

3.1 Processador Java tempo-real

O processador FemtoJava (ITO, 2001), utilizado neste trabalho, é um microcontrolador de pilha que executa bytecodes Java nativamente. As principais características do FemtoJava são o seu conjunto de instruções reduzido e configurável, arquitetura Harvard e tamanho pequeno, tirando vantagem do reuso proporcionado pelo paradigma da orientação a objetos. O FemtoJava implementa uma JVM (*Java Virtual Machine*) em hardware, através de uma máquina de pilha que implementa um subconjunto dos bytecodes da JVM padrão. Estas instruções são operações com inteiros e de manipulação de bits, operações de leitura e armazenamento na memória, desvios condicionais e incondicionais, operações de pilha e dois pseudo-bytecodes para leitura e escrita arbitrária.

Dentre as diversas implementações do processador a multiciclo foi escolhida por apresentar maior previsibilidade sendo, portanto, vantajosa para aplicações de tempo-real. Neste processador, todas as instruções são executadas em um número fixo de ciclos (03, 04, 07 ou 14 ciclos) e não há memória *cache*.

Após a versão inicial do processador, publicada por ITO (2001), os bytecodes utilizados para operações com objetos foram incluídos, como `putfield`, `getfield`, `invokevirtual`, `invokespecial` e `instanceof`. Para permitir aplicações multitarefa a solução adotada foi a introdução de mais dois pseudo-bytecodes, `save_ctx` e `restore_ctx`, usados para a troca de contexto (ROSA, 2003). Estas adaptações/expansões no FemtoJava multiciclo original, juntamente com a inclusão de um relógio de tempo-real (RTC), permitiram a implementação de uma API compatível com a especificação RTSJ (WEHRMEISTER, 2004).

Para usar esta plataforma, o conjunto de classes de uma aplicação deve passar por um compilador Java padrão e em seguida por uma ferramenta de pós-compilação, chamada SASHIMI. Esta ferramenta lê todas as classes da aplicação e gera o código final, que pode ser usado tanto para simulação quanto para síntese lógica. Este código final é um conjunto de arquivos que inclui a descrição VHDL do processador (cujo conjunto de instruções contém somente aquelas usadas pela aplicação) a as memórias ROM (métodos) e RAM (atributos). O SASHIMI ainda remove do código final os métodos e atributos não alcançados, ou seja, aqueles que não são utilizados pela aplicação em nenhum momento da execução. Assim, o SASHIMI é também uma ferramenta de otimização de JVM.

No desenvolvimento de aplicações sobre a plataforma SASHIMI-FemtoJava os objetos da aplicação devem ser alocados estaticamente em tempo de projeto. Em outras palavras, todos os objetos no sistema são construídos *a priori* (pelo SASHIMI), permitindo que seja determinada a quantidade total de memória RAM para comportar a aplicação. Embora esta prática leve a um uso maior de memória, ela é conveniente no desenvolvimento de sistemas de tempo-real, uma vez que ela evita o uso do *Garbage Collector*, cujo custo computacional introduz um indeterminismo intolerável para esses sistemas.

3.1.1 API-RTSJ

A RTSJ (*Real-Time Specification for Java*) é um conjunto de especificações de comportamento voltado para o desenvolvimento de aplicações de tempo-real usando a linguagem de programação Java. Dentre as suas principais propriedades estão: escalonamento multitarefa adequado para aplicações tempo-real com possibilidade de descrever tarefas periódicas e esporádicas, orçamento de tempo de processamento e *deadline* de tarefas.

A RTSJ permite o uso de objetos escalonáveis, que são instâncias de classes que implementam a interface chamada de `Schedulable`, como a `RealtimeThread`. A especificação também define um conjunto de classes para armazenar parâmetros que representam a demanda de um ou mais objetos `Schedulable` (escalonáveis) por um recurso em particular. Por exemplo, a classe `ReleaseParameters` (superclasse de `AperiodicParameters` e `PeriodicParameters`) inclui alguns parâmetros úteis para especificar requisitos de tempo-real, como ativação cíclica e *deadline*.

Este trabalho usa uma API baseada na RTSJ, apresentada em (WEHRMEISTER, 2004). Naquele trabalho o conceito de *thread* é um pouco diferente do conceito clássico (SILBERSCHATZ, 2004) e inclui o fluxo de controle associado a seu espaço de endereçamento. Esta associação também vale para o restante deste texto.

Na linguagem Java padrão a JVM faz o escalonamento das *threads* da aplicação e é executada sobre o sistema operacional. Na plataforma aqui utilizada, a JVM é o processador e sobre ele é executada a API RTSJ de Wehrmeister (2004), que faz as funções de gerenciamento de *threads*, típicas do RTOS. Isso requer a flexibilização dos conceitos clássicos de sistema operacional, JVM e *thread*.

A *thread* na API RTSJ não é criada (construída) dinamicamente (em tempo de execução), mas é definida em tempo de projeto, juntamente com os objetos aos quais ela

tem acesso. Portanto, o seu espaço de endereçamento é conhecido *a priori*. O compartilhamento de dados com outras *threads* da aplicação é definido explicitamente.

A implementação de algumas das classes da API tem pequenas diferenças em comparação com o padrão RTSJ. Isto ocorre devido a restrições na plataforma adotada e também para aumentar a clareza. Um exemplo destas diferenças aparece na classe `RealtimeThread`. Esta classe usa dois métodos abstratos que devem ser implementados nas suas subclasses: `mainTask()` e `exceptionTask()`. Eles representam, respectivamente, o corpo da tarefa – equivalente ao método `run()` de uma *thread* Java normal – e ao código de tratamento de exceções, necessário para lidar com as perdas de *deadline*.

A estrutura de escalonamento de tarefas consiste em um processo adicional que é responsável pela alocação do processador a um dos processos da aplicação que esteja pronto para executar, exatamente como em qualquer sistema operacional de tempo-real (RTOS – *Real-Time Operating System*). O desenvolvedor da aplicação deve escolher o algoritmo de escalonamento mais adequado em tempo de projeto. Então, o escalonador selecionado é sintetizado junto com a aplicação no código final do sistema embarcado.

3.2 A NoC SoCIN

Como principal rede de conexão dos processadores foi utilizada a NoC SoCIN por estar implementada no simulador SERPENS, utilizado para a maior parte dos experimentos.

A NoC SoCIN (ZEFERINO 2003) utiliza chaveamento de pacotes (*packet switching*) e foi desenvolvida visando a sua prototipação em FPGAs (*Field Programmable Gate Array*). O projeto também procurou desenvolver uma rede escalável, baseando-se em um roteador parametrizável denominado RaSOC (ZEFERINO 2004). A NoC SoCIN utiliza chaveamento *wormhole* podendo, por isso, contar com *buffers* de menor capacidade nos roteadores, economizando em tamanho e em energia. O roteamento é do tipo XY, livre de *deadlocks*. A comunicação é baseada no modelo de troca de mensagens. Os pacotes que compõem as mensagens são compostos de *flits*. Um *flit* (***f****low control un****it***) é a menor unidade sobre a qual o controle de fluxo atua. Na SoCIN, um *flit* coincide com a palavra do canal físico (ou *phit* – ***ph****ysical un****it***).

A parametrização da NoC permite um ajuste fino da mesma com o objetivo de atender da melhor maneira possível os requisitos das aplicações. A relação desempenho-consumo pode ser explorada através da alteração dos parâmetros da rede. O tamanho de um *flit* é parametrizável e pode adotar qualquer valor múltiplo de oito.

Embora conceitualmente a SoCIN possa suportar outros dispositivos conectados a seus roteadores a implementação usada neste trabalho considera que apenas processadores estão conectados através da rede. Em Silva Júnior (2007) é proposta uma investigação do uso de memória compartilhada em NoCs conectando memórias e processadores à rede SoCIN.

3.3 O simulador

A verificação do funcionamento de uma aplicação na plataforma FemtoJava pode ser feita por implementação direta em FPGA ou por simulação. No contexto deste trabalho optou-se pela simulação feita no nível de instruções do processador, que são bytecodes Java. Assim, é possível validar o código Java implementado sem recorrer à síntese em FPGA ou ao uso de ferramentas de simulação de hardware.

No desenvolvimento deste trabalho dois simuladores foram usados para verificação do comportamento do processador FemtoJava quando executando o código do *middleware*, o SERPENS e o CACO-PS. Nas seções seguintes serão apresentadas as principais características de cada um deles.

3.3.1 SERPENS

O simulador SERPENS foi empregado pela primeira vez em (WRONSKI, 2006). Como linguagem de descrição do processador e da NoC, o SERPENS utiliza o SystemC (GRÖTKER, 2002). Para aquele trabalho foi utilizada uma descrição simplificada do processador enquanto que a NoC foi descrita em nível TLM (*Transaction Level Modeling*).

Em (BARCELOS, 2006) o SERPENS foi modificado pela inclusão de um modelo RTL (*Register Transfer Level*) do processador FemtoJava, permitindo a execução do código Java obtido da ferramenta SASHIMI. Este modelo permite a simulação do processador em nível de ciclo de relógio (*clock*), sendo mais preciso (e de custo computacional mais elevado) que o modelo da NoC. Como neste trabalho o foco está no comportamento dos processadores torna-se conveniente esta característica do simulador.

3.3.2 CACO-PS

O CACO-PS (*Cycle-Accurate COnfigurable Power Simulator*) (BECK FILHO, 2003) é uma ferramenta que permite simular o comportamento de um processador usando o código compilado. Com precisão de tempo de relógio, tanto em relação ao desempenho quanto à energia, o simulador permite uma análise detalhada do comportamento dos componentes simulados. No CACO-PS é possível usar descrição estrutural ou comportamental de qualquer arquitetura e novos componentes podem ser adicionados ao sistema.

O simulador CACO-PS foi desenvolvido na linguagem C procurando-se obter um desempenho melhor na velocidade das simulações, comparado à simulação de hardware. A partir de uma biblioteca de componentes descritos também na linguagem C, é possível instanciar diferentes arquiteturas através da combinação desses componentes. A descrição da arquitetura é lida uma única vez no início da execução da simulação.

Diversas organizações de processadores Femtojava já se encontram disponíveis para o simulador CACO-PS, embora, originalmente, este não forneça suporte à execução de sistemas multiprocessados. Não existem, também, descrições de meios de comunicação, sejam eles barramentos ou redes-em-chip. Para este trabalho, um modelo de interface de rede foi descrito em linguagem C e integrado ao processador FemtoJava. O modelo

utiliza sockets TCP/IP para emular o comportamento da rede simulada. Desta forma, a simulação pode ser feita com processos sendo executados em uma única estação de trabalho, ou pode-se ter uma simulação distribuída, com os processos sendo executados em estações distintas. Para cada processador sendo simulado, um CACO-PS é instanciado e todos eles se comunicam através de um simulador da rede. Este recebe as mensagens de cada CACO-PS conectado e envia ao destino, conforme o comportamento da rede simulada.

3.4 Discussão

Neste capítulo foi apresentado o ambiente onde o *middleware* deste trabalho foi desenvolvido e experimentado. O processador FemtoJava oferece uma máquina virtual Java implementada em hardware. O *middleware* pode, portanto, ser escrito em Java, desde que se atendam as restrições determinadas pelo ambiente SASHIMI, responsável pela construção dos objetos.

Para verificação do funcionamento do código Java utilizam-se simuladores com precisão de ciclo de relógio do processador, permitindo a análise detalhada do comportamento de todos os componentes do sistema. Esta propriedade é particularmente útil quando o *middleware* faz interface entre a aplicação e algum componente específico em hardware, seja este componente parte da aplicação ou do *middleware*.

O capítulo seguinte traz a visão geral do *middleware*, bem como as propriedades de cada um de seus componentes.

4 O *MIDDLEWARE*

Este capítulo apresenta uma arquitetura de *middleware* para atender às necessidades de projeto de sistemas embarcados distribuídos e com restrições de tempo e energia. Como estrutura de hardware se presume uma plataforma MPSoC (*MultiProcessor System on Chip*), embora as propriedades do *middleware* aqui descrito possam ser aproveitadas em aplicações distribuídas em uma rede embarcada convencional (não intra-chip).

No contexto deste trabalho, o *middleware* tem como principal requisito facilitar o reuso da infra-estrutura de hardware e software já existente, aliando mecanismos que auxiliem na expressão de propriedades de tempo-real. Estas propriedades devem ser alcançadas sem perder de vista as limitações de recursos como energia, memória e tempo de processamento.

O *middleware* deve gerenciar adaptações em tempo-real, deixando o programador das aplicações livre desta tarefa. Além disso, a complexidade das ações de comunicação entre processadores e entre hardware e software deve ser encapsulada em primitivas de alto nível.

A arquitetura proposta para o *middleware* está organizada em camadas com níveis diferenciados de abstração. Ela está direcionada para facilitar a exploração do espaço de projeto atuando em variáveis como localização na rede e implementação hardware/software, buscando melhores resultados em parâmetros como energia ou memória, além de atender a restrições temporais. A Figura 4.1 apresenta uma visão organizacional da arquitetura, que está dividida em dois níveis de abstração: O nível de estrutura e o nível de serviço. O nível de estrutura oferece os recursos mais elementares do *middleware*, que são a comunicação em rede e o gerenciamento multitarefa. A fim de atender a requisitos de tempo-real estes recursos permitem expressão de propriedades de tempo, definidas na RTSJ, como será mostrado nas seções seguintes. Utilizando-se um conceito de *middleware* mais conservador, este nível de estrutura poderia ser caracterizado como o RTOS sobre o qual o *middleware* executa. Entretanto, em função da necessidade de otimizar o uso dos recursos da plataforma, esta camada não é completamente abstrata para a aplicação, como se verá nas seções seguintes. O nível de serviços oferece uma abstração maior e utiliza os recursos implementados no nível de estrutura. São serviços básicos, se considerada a complexidade de um sistema distribuído de propósito geral, como os citados no início do Capítulo 2. Entretanto, estes serviços são suficientes para dar suporte ao projeto de aplicações embarcadas

multiprocessador, permitindo a exploração de diferentes arranjos para alocação das tarefas tanto em tempo de projeto quanto em tempo de execução.

Cabe destacar aqui os serviços de supervisão e DVS, no nível de estrutura, dentro do bloco RTSJ. Estes serviços não fazem parte da especificação original RTSJ, mas foram definidos no *middleware* como suporte ao nível de serviços. As seções seguintes explicam cada um dos componentes do *middleware* vistos na figura.

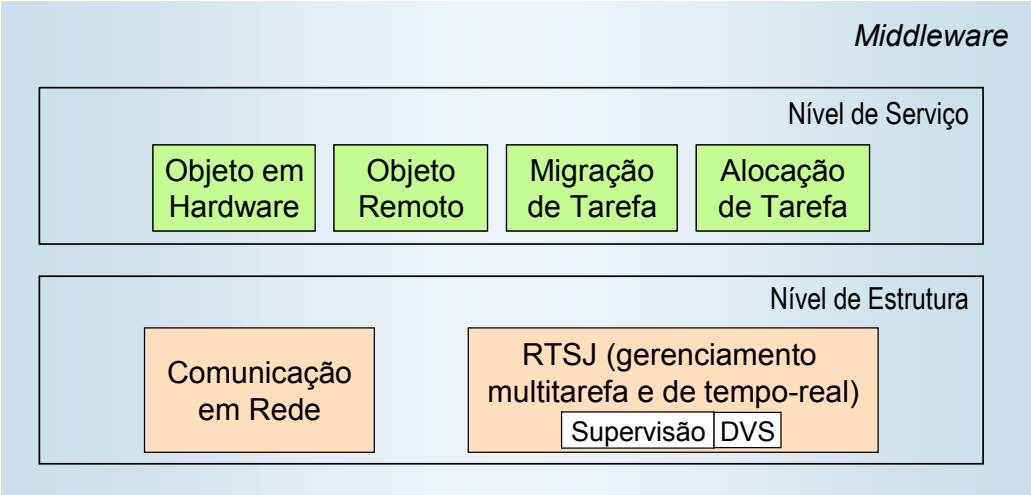

Figura 4.1: Arquitetura geral do *middleware*

4.1 Gerenciamento multitarefa e de tempo-real

4.1.1 Especificação Tempo-real para Java (RTSJ)

Para escalonamento multitarefa este trabalho usa uma API baseada na RTSJ, introduzida em (WEHRMEISTER, 2004) e já apresentada na Seção 3.1.1. Sobre esta API é possível descrever tarefas periódicas e esporádicas, orçamento de tempo de processamento e *deadline* de tarefas.

A RTSJ (*Real-Time Specification for Java*) define um conjunto de especificações de comportamento para permitir o desenvolvimento de aplicações tempo-real usando a linguagem de programação Java.

Entretanto, o que a especificação RTSJ não contempla é a necessidade de se otimizar o uso de recurso computacional e de energia. Falta ainda uma definição de mecanismos de comunicação entre tarefas localizadas em processadores distintos, como discutido por Borg (2003). Na Seção 4.2 será apresentada uma estratégia de comunicação que utiliza a RTSJ para permitir especificar restrições temporais.

4.1.2 Funções adicionais à RTSJ

A função chamada ‘Supervisão’ visa à monitoração contínua dos recursos do processador local, como ocupação e uso de memória. Tal função é oferecida ao serviço

de ‘Alocação de Tarefa’ para subsidiar a tomada de decisão no momento de adicionar ou remover tarefas de um processador.

A função DVS/DFS, que será melhor explicada adiante, foi adicionada aos escalonadores para que a decisão de aumentar ou diminuir a freqüência de operação do processador seja tomada de maneira transparente para a aplicação.

4.2 Comunicação em rede

Aplicações distribuídas pressupõem o uso de algum serviço de comunicação. Assim, o processo de especificação e desenvolvimento do *middleware* passa necessariamente por uma infra-estrutura de comunicação, que torne possível a troca de dados entre os componentes do sistema.

Uma API de comunicação foi especificada e desenvolvida para prover a troca de mensagens entre aplicações executadas em diferentes processadores FemtoJava. A API, introduzida em (SILVA JÚNIOR, 2006a), permite que a aplicação estabeleça um canal de comunicação através da rede, que pode ser usado para enviar e receber mensagens.

O serviço permite a associação de diferentes prioridades às mensagens bem como a definição de tempos máximos para envio ou espera por mensagens. Para especificação dos tempos são utilizados os tipos estabelecidos pelo padrão RTSJ.

Do ponto de vista da aplicação, o sistema é capaz de abrir e fechar conexões, assim como enviar e receber mensagens, podendo ser acessado por diferentes tarefas simultaneamente. A aplicação pode ainda enviar mensagens no modelo cliente-servidor (orientado a conexão – ponto-a-ponto) ou por *publish-subscribe* (sem conexão prévia – ponto-multiponto).

A Figura 4.2 mostra uma arquitetura multiprocessador que ilustra onde a API de comunicação se insere. Da API RTSJ são usados os recursos de escalonamento de tarefa e de definição de tempos. Além disso, é usado o recurso de tratamento de eventos, que será detalhado em uma subseção em separado.

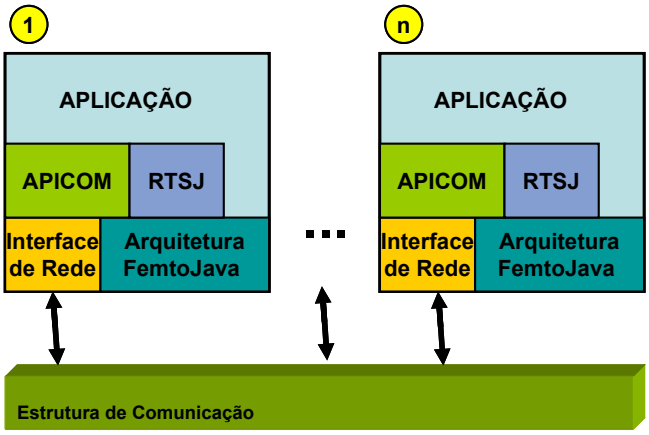

Figura 4.2: Arquitetura simplificada (no contexto da API de comunicação)

Abaixo da API de comunicação se assume que existe uma interface de rede capaz de oferecer alguma qualidade de serviço (QoS). As garantias que devem ser providas pela rede são: (1) garantia de entrega – a rede não perde pacotes; (2) garantia de seqüência – o primeiro pacote a ser enviado é o primeiro a chegar ao destino, ou seja, a rede estabelece caminho único para todos os pacotes de uma mensagem; (3) a rede oferece mecanismo de prioridade associado aos pacotes, que é usado em caso de contenção. Estas propriedades derivam do CAN-bus (BOSCH, 1991), definido na primeira versão da API como rede de suporte. Pelo mesmo motivo, a API não oferece mecanismos mais sofisticados de QoS, como de reserva ou contrato de banda.

O sistema de comunicação foi dividido em camadas ou níveis que prestam serviços um para o outro de forma a atingir o objetivo de prover a comunicação desejável à aplicação. Para isto, adotou-se o modelo de referência OSI-ISO de pilha de protocolos de comunicação (TANENBAUM, 2003). A API apresenta as camadas de transporte, rede e enlace. Cada uma destas camadas tem funções bem específicas e complementa as funções das demais.

A Figura 4.3 mostra o diagrama de classes da API de comunicação. A classe `Transport` implementa o nível de transporte e é a interface do sistema de comunicação com a aplicação final ou com outros serviços acima. Neste nível se encontra o particionamento das mensagens em pacotes que serão utilizados pelos níveis inferiores bem como a remontagem da mensagem quando os pacotes são recebidos. A classe `Message` representa a informação que chega da aplicação (ou será recebida da conexão remota). `TransportConnection` é a classe que torna possível a individualização de cada conexão armazenando o endereço lógico e a porta utilizados. A classe `Network` trabalha como um filtro, selecionando pacotes destinados ao nodo (*host*) local e direcionando-os para o `Transport`. Para que a API de comunicação pudesse ser reusada em diferentes tecnologias e padrões de rede, a classe `DataLink` foi modelada como abstrata. Assim, estendendo-se esta classe é possível implementar classes específicas para um determinado tipo de rede. As classes `DataLinkCan` e `DataLinkSOCIN` fornecem a implementação específica para a rede CAN (*Control Area Network*) e para a NoC SoCIN, respectivamente. A classe `Pack` é utilizada para transportar um pacote de dados em um formato não dependente do tipo de rede utilizado.

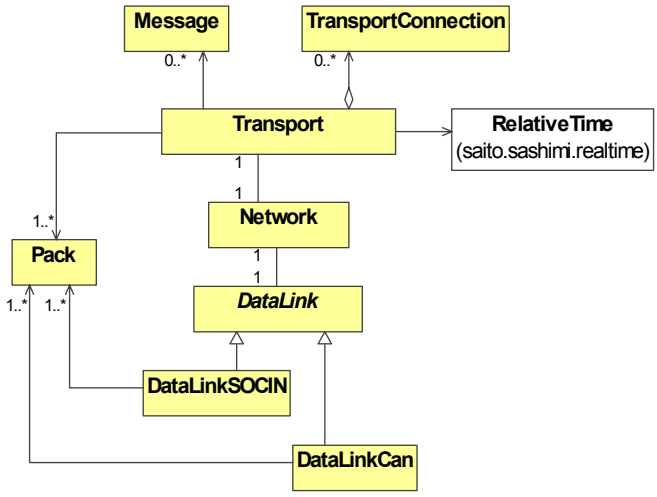

Figura 4.3: Diagrama de classes da API de comunicação (básico)

Tabela 4.1: Etapas do processo de receber uma mensagem

Etapa	Descrição	Executor
1	Um objeto `Pack` é preenchido	`DataLink`
2	Caso seja um pacote de dados a fragmentação é analisada e uma mensagem é montada	`Transport`
3	Uma variável booleana é atualizada indicando uma mensagem pronta	`Transport`
4	Testa se existe uma mensagem pronta na conexão aberta	Aplicação
5	Recebe a mensagem no objeto `Message` informado	Aplicação

O fluxo seguido no processo de recepção de uma mensagem é resumido na Tabela 4.1. Para cada etapa é dada uma descrição do que ocorre, sendo que o principal ator na etapa está indicado na coluna executor. A Tabela 4.2 traz uma descrição geral dos serviços que são disponibilizados pela API de comunicação.

Tabela 4.2: Serviços oferecidos pela API de comunicação

Serviço	Descrição
Estabelecimento de conexão	Aplicações podem requisitar e esperar por conexões. A API devolve um número que identifica a conexão e é usado nas operações de envio e/ou recepção de mensagens.
Troca de mensagens	Aplicações trocam informações enviando e recebendo mensagens, que são seqüências de bytes.
Estabelecimento de endereço lógico	A aplicação pode definir o endereço local, que será usado posteriormente para identificação dos nodos (ou estações) da rede.
Broadcast	Mensagens podem ser enviadas diretamente para um *host* (nodo) específico, através de uma conexão predefinida, ou por difusão na rede. Esta opção é feita chamando primitivas diferentes da API quando do envio de uma mensagem. Para receber mensagens de broadcast um *host* deve antes fazer uma operação de assinatura.

Embora este trabalho tenha utilizado as redes CAN e SoCIN para comunicação, nada impede que a estrutura de classes da API de comunicação seja usada para encapsular comunicação por meio de outros protocolos de comunicação e topologias de rede. Para isso, é bastante escrever uma classe concreta que estenda `DataLink`, a exemplo das classes `DataLinkSOCIN` e `DataLinkCan`.

4.2.1 Eventos na comunicação

As funções de comunicação podem ser um fator importante nos custos de um sistema embarcado, tanto no custo computacional quanto em energia. Além disso, a comunicação também pode introduzir indeterminismo para aplicações de tempo-real, devido à sua natureza assíncrona. Assim, são necessárias estratégias para reduzir o impacto da comunicação no custo geral do sistema e torná-lo mais previsível.

Este trabalho propõe a integração do mecanismo de eventos assíncronos oferecido pela RTSJ com o serviço de comunicação. Usando a abordagem de eventos, as *threads* que estão esperando por mensagens podem dormir até que a mensagem chegue, abrindo assim espaço para uma otimização de energia. Mais adiante esta estratégia será demonstrada utilizando um escalonamento de tarefas com DVS (*Dynamic Voltage Scaling*) / DFS (*Dynamic Frequency Scaling*).

Eventos são imprevisíveis por natureza. Para lidar com isso foi usado um *Pooling Server* (SPURI, 1994), uma *thread* periódica e com um orçamento de tempo limitado. O objeto `PoolingServer` executa periodicamente e usa seu orçamento para executar *threads* esporádicas e não periódicas, como os tratadores de eventos. Quando o orçamento termina, o *Pooling Server* suspende qualquer *thread* esporádica ou não periódica eventualmente em execução até o seu próximo ciclo. Com isso, as *threads* periódicas não são afetadas e a previsibilidade do sistema é assegurada.

Dois serviços da APICOM se caracterizam por esperar um evento. São eles: espera por um pedido de conexão e espera por uma mensagem. Sem usar o recurso de eventos, estas duas esperas podem ser feitas por uma chamada de método bloqueante, que aguarda pelo evento ou volta após um *time out*. O problema com esta abordagem é que ela não oferece ao escalonador de tarefas a possibilidade de passar o processador a outra tarefa, desperdiçando tempo e energia. Além disso, esta espera pode introduzir um *jitter* importante no tempo de execução da *thread*.

A introdução do conceito de eventos é feita da seguinte maneira (SILVA JÚNIOR, 2007a). A aplicação que deseja esperar por mensagens fará isso passando uma referência de objeto `AsyncEvent` para a APICOM, que representa o evento propriamente, utilizando para ele um objeto tratador, `AsyncEventHandler`. Além disso, um mecanismo de tratamento de eventos (`AsyncEventsMechanism`) deve ser escolhido e informado ao escalonador de tarefas. Na implementação de teste o mecanismo utilizado foi um *Pooling Server*.

A APICOM oferece o método `setMsgRdyEvent()`, que permite informar o objeto evento que será acionado quando uma mensagem estiver pronta na porta de conexão informada. O objeto tratador implementa a ação de recebimento de mensagens, que deve ser uma chamada direta ao método `receiveMsg()` da APICOM, mas o programador é livre para fazer outras operações dentro do tratador. Da mesma forma o método `setRequestEvent()` serve para que a aplicação informe o objeto evento que será disparado quando um pedido de conexão chegar. Então, a aplicação não precisa esperar, bloqueada, pela chegada de mensagens ou pedidos de conexão. O projetista precisa apenas escrever um tratador de eventos que contenha o comportamento necessário e informar à APICOM este tratador. Quando uma mensagem chega (etapa três da Tabela 4.1) o evento de tratamento é ativado e a aplicação não precisa executar a etapa 4 da mesma tabela.

O recebimento de pacotes, realizado pela classe `DataLink`, também pode ser feito por tratamento de eventos. Para isso, a aplicação pode invocar o método `setEventStatus()`, passando `TRUE` para ativar o recurso. Quando um pacote chega até a interface de rede uma interrupção é gerada. O serviço de interrupção somente coloca o pacote em uma fila e dispara um evento na classe `PackHandler`. O tratamento deste pacote (etapa três da Tabela 4.1) ocorrerá dentro de uma janela de tempo pré-determinada sob o controle do `AsyncEventsMechanism` da RTSJ, e não imediatamente

após a interrupção. Desta forma se assegura um custo controlado da comunicação, evitando que uma chegada de mensagem possa monopolizar o processador por algum tempo. A Figura 4.4 mostra o diagrama de classes da APICOM incluídas as ligações com o tratamento de eventos da RTSJ. O `Transport` tem uma estrutura onde armazena referências para os eventos informados pela aplicação e que serão disparados por chegada de mensagens ou pedidos de conexão. Já o `DataLink` tem seus próprio evento e tratador, que são associados à chegada de pacotes.

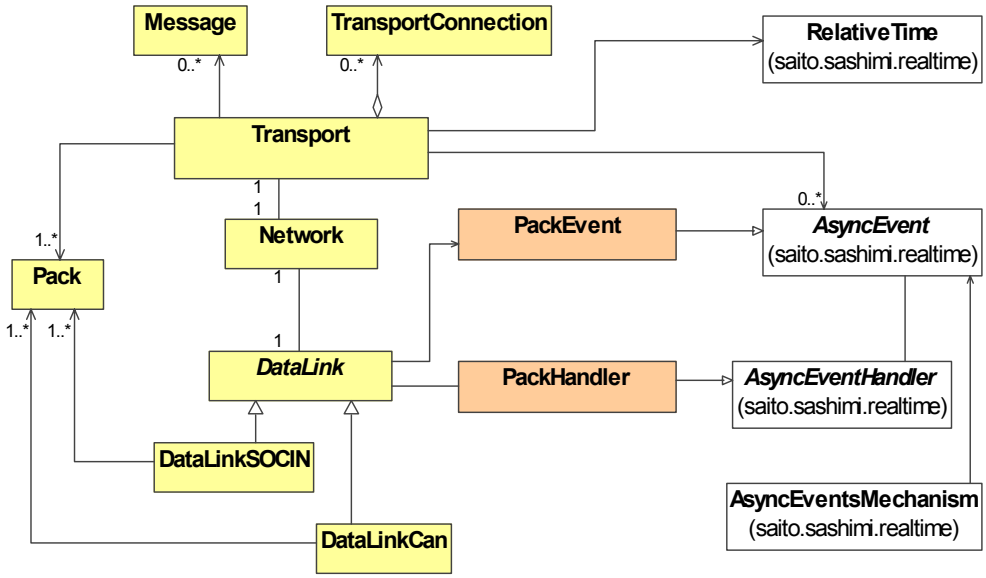

Figura 4.4: Diagrama de classes da API de comunicação (uso de eventos)

4.3 Abstração da Localização

Uma demanda importante na exploração do espaço de projeto em sistemas MPSoC é permitir que os objetos da aplicação possam estar alocados em qualquer nodo da rede e que esta localização possa ser resolvida em tempo de execução da aplicação. Isto traz flexibilidade e simplicidade ao projeto de aplicações distribuídas. Para isso um mecanismo de localização abstrata de objetos deve ser oferecido para que métodos de outros objetos possam ser invocados sem que a sua localização seja conhecida em tempo de projeto. Mais ainda, este mecanismo de localização abstrata de objetos remotos precisa ser integrado à RTSJ e oferecer garantias de tempo de entrega de mensagens.

A linguagem Java oferece um mecanismo chamado RMI (*Remote Method Invocation*) (SUN, 1996) que permite que uma operação seja executada em um *host* (nodo) remoto como se ela fosse implementada localmente. Todavia, no contexto deste trabalho existem algumas restrições significativas ao mecanismo RMI de Java padrão: (1) ele não permite expressar restrições de tempo, e (2) o RMI depende de um mecanismo de alocação dinâmica de memória, dois problemas para aplicações de tempo-real. Paralelamente, o (3) RMI trabalha sobre TCP/IP, que não oferece garantias de tempo de entrega de mensagens.

Em (BORG, 2003) é apresentado um *framework* para RMI tempo-real que dá suporte à invocação de métodos com tempo controlado. O trabalho não oferece uma implementação completa, mas discute como o *framework* poderia ser usado para desenvolver uma. Os autores consideram o modelo de memória de RTSJ um desafio para a implementação do RMI.

Este trabalho propõe uma especificação de RMI utilizando o modelo de memória da plataforma SASHIMI-FemtoJava e permitindo expressar propriedades de tempo-real da RTSJ. *Threads* RT-Java podem invocar objetos remotos e elas podem contar com uma entrega de mensagens com tempo controlado e parâmetros de escalonamento. A invocação de métodos remotos deste trabalho foi desenvolvida sobre a API de comunicação apresentada na Seção 4.2.

De uma maneira simplificada, existem três entidades principais: o cliente (aquele que deseja acessar um objeto remoto), o servidor (a implementação remota do objeto) e o registrador de nomes (serviço que informa a localização dos objetos cadastrados). Inicialmente o serviço de registro de nomes, chamado `Registry`, deve ser ativado. A partir daí o objeto remoto já pode ser publicado pelo lado servidor da aplicação, ficando disponível para que o lado cliente faça a consulta junto ao `Registry` e obtenha uma referência da localização remota do objeto servidor. A Figura 4.5 mostra o fluxo do registrador de nomes e a sua interação com o servidor e o cliente. No processo chamado `bind` o servidor cadastra um objeto, enquanto que o cliente usa o `lookup` para consultar o registro à procura de um objeto.

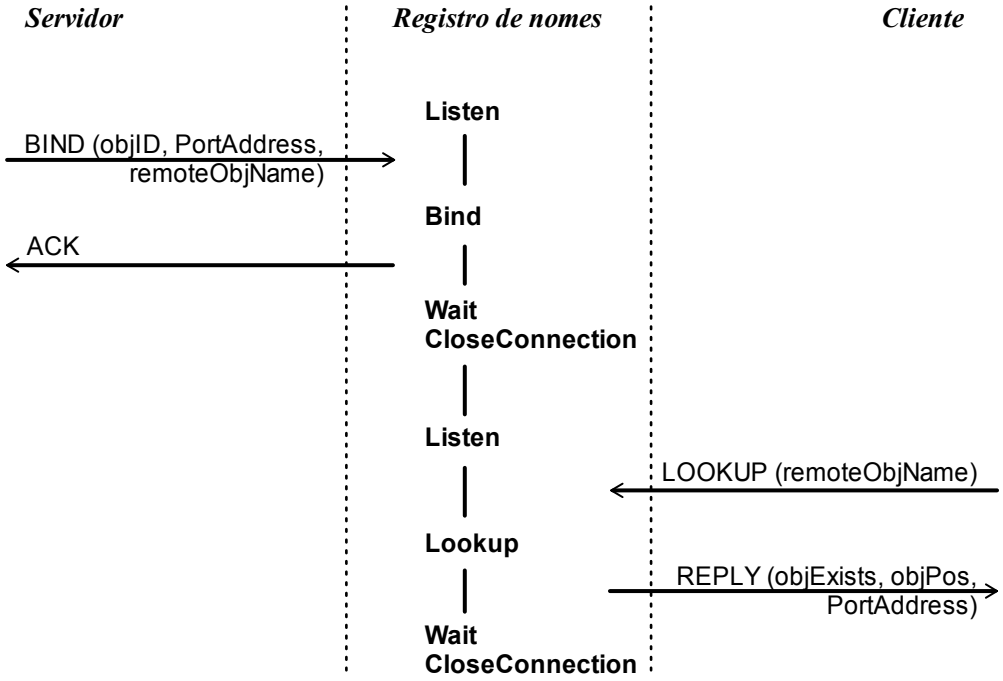

Figura 4.5: Protocolo do registrador de nomes para métodos remotos

A Figura 4.6 mostra a arquitetura do mecanismo de invocação remota de métodos. Nele está presente uma interface que isola o cliente e o servidor das camadas inferiores de software e hardware e seus requisitos de comunicação. No lado cliente, a interface é

chamada Stub e no lado do servidor ela é chamada de Skeleton. Em termos práticos, o Stub oferece uma referência da operação remota que, ao invés de implementá-la de fato, efetua a comunicação (empacotamento e envio de dados) com o lado servidor, onde existe a implementação do objeto.

No lado servidor, o Skeleton recebe os dados vindos do Stub, organiza e efetua a chamada à implementação real do objeto. No final, Skeleton e Stub enviam o resultado na direção contrária. Esta comunicação é escondida da aplicação pelas classes que compõem o mecanismo nos dois lados.

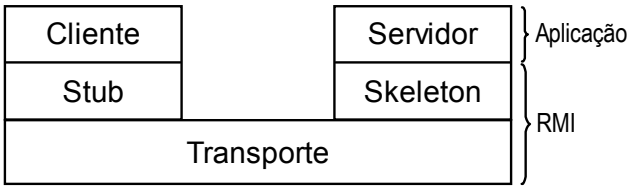

Figura 4.6: Modelo de camadas de RMI

Ao mecanismo original de RMI foi introduzida uma modificação conceitual, ao se usar limitadores de tempo para as operações utilizando recursos da RTSJ. Uma classe específica para encapsular propriedades de tempo-real foi incluída (`RealTimeParameters`) tanto no lado cliente quanto no lado servidor, permitindo à aplicação expressar suas restrições de tempo. A *thread* (`ConnectionHandler`) que trata as conexões no lado servidor é outro mecanismo que traz previsibilidade ao RMI. Esta *thread* é de tempo-real e tem custo máximo definido *a priori*. Assim, as operações de comunicação não irão violar os tempos destinados às demais tarefas da aplicação.

4.3.1 O Cliente de método remoto

A Figura 4.7 mostra o diagrama de classes do lado cliente da invocação remota de método. O mecanismo consta de duas fases distintas. A primeira se dá na geração do código, quando a classe `MyClass_Stub` é gerada. Ela implementa os métodos do objeto remoto, mas o seu código empacota o nome do método e seus argumentos em uma mensagem e os envia para o servidor, cuja localização é informada somente em tempo de execução.

A segunda fase do mecanismo ocorre em tempo de execução. O cliente invoca o método `lookup` da classe `ClientNaming`, responsável por localizar o objeto remoto (através do `Registry`). O `lookup`, por sua vez, devolve uma referência de um objeto local, tipo Stub, implementado pela classe `MyClass_Stub`. A classe `ClientNamingSOCIN` é a implementação do `Naming` que utiliza a rede SoCIN, usada para os testes. Dentro desta classe está o acesso à API de comunicação, invocada através do nível de transporte.

O método `lookup` estabelece conexão com o `Registry`, envia o identificador do objeto procurado e recebe endereço e porta da sua localização. Em seguida, instrumenta um `Stub` local com os dados do objeto remoto e passa a referência deste `Stub` para a aplicação. Em uma JVM convencional, o objeto `Stub` poderia ser construído neste

momento. Aqui se usa o termo instrumentar porque um conjunto de Stubs já está alocado, bastando preencher seus atributos com as informações recebidas do `Registry` e da aplicação.

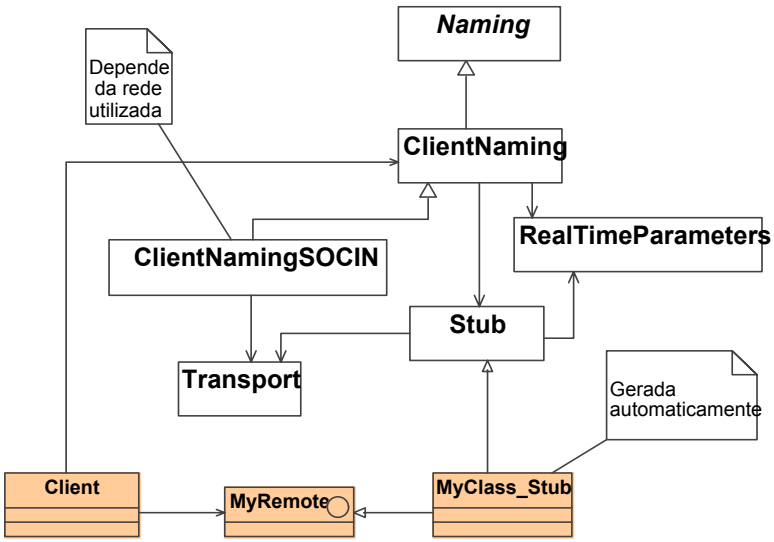

Figura 4.7: Diagrama de Classes (lado cliente) do serviço de acesso a objeto remoto

Caso o projetista opte por outros protocolos de rede, ele usará uma outra classe `ClientNaming` que encapsule a rede escolhida. Neste caso, basta que a implementação do `DataLink` seja compatível com a nova rede que todos os serviços da API de comunicação e de invocação de métodos remotos serão reusados. Nos testes foi usada também uma rede CAN, para a qual existem o `DataLinkCAN` e um `ClientNamingCAN`, que serviram para validar a flexibilidade do modelo.

A classe `RealTimeParameters` contém as restrições de tempo que serão usadas desde o estabelecimento de conexões até o envio de mensagens e espera de confirmações. Esta classe foi introduzida para que a invocação de métodos remotos possa se utilizar dos mecanismos de restrição temporal disponíveis na RTSJ e na API de comunicação. A implementação atual oferece apenas mecanismos de *time out*, ficando a implementação de mecanismos mais sofisticados de garantia de qualidade de serviço para trabalhos futuros.

4.3.2 O Servidor de método remoto

Da mesma forma que no cliente, existe uma classe que deve ser gerada em tempo de projeto, na Figura 4.8 chamada `MyClass_Skeleton`, e que é responsável pelo desempacotamento da mensagem com os dados da invocação do método remoto. O padrão Factory Method (GAMMA, 2000) foi usado (classe `SkelFactory`) com uma pequena modificação, para que a aplicação não precise saber o nome desta classe e para preservar o uso de objetos estáticos do SASHIMI-FemtoJava.

Em tempo de execução o servidor tem uma complexidade maior que o cliente. Como se vê na Figura 4.8, a aplicação (classe `Application`) contém a classe que será acessada remotamente, aqui chamada `MyClass`, que deve ser do tipo `Remote_object`. Esta classe traz consigo as propriedades que vão permitir que uma *thread* tempo-real (`ConnectionHandler`) seja informada quando chegar uma chamada ao objeto que foi publicado. O mecanismo de publicação do objeto (vide Figura 4.5) consiste em uma chamada ao método `bind()`, que é redirecionado para o `Naming`. Este faz o registro do objeto no `Registry`, reserva e instrumenta um `Skeleton` para o objeto. Ele ainda informa ao `ConnectionHandler` que o objeto está publicado e quem é o seu `Skeleton`, passando a referência do `MyClass_Skeleton`.

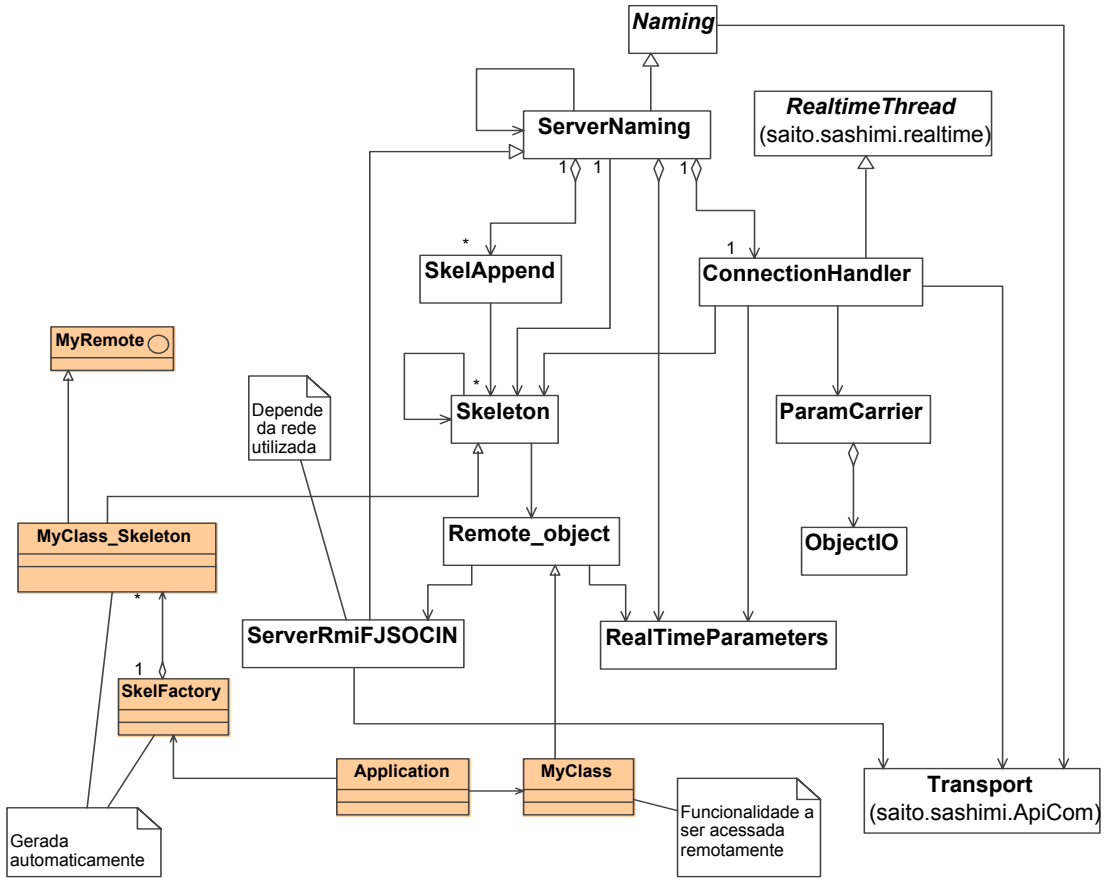

Figura 4.8: Diagrama de classes (lado servidor) do serviço de acesso a objeto remoto

A Figura 4.9 mostra o que acontece no lado servidor para que o método remoto seja executado. Quando uma chamada remota chega ao servidor ela deve ser tratada pela *thread* `ConnectionHandler`. Esta *thread* identifica o Skeleton que possui conexão válida e recupera a referência para esta conexão (2). Em seguida ela recebe a mensagem enviada pelo Stub (3) e recupera o código identificador do método e seus argumentos de dentro da mensagem. Os argumentos são colocados dentro de um objeto tipo `ParamCarrier`, (4) no campo `InputStream`, destinado aos parâmetros de entrada do método. (5) O método `dispatch()` permite 'invocar' o método no Skeleton, informando qual dos métodos do objeto está sendo invocado e o objeto que contém os

seus argumentos. O `Skeleton` então chama a implementação do objeto local (`MyClass`, neste caso) e devolve o retorno do método para o `ConnectionHandler`, no mesmo objeto `ParamCarrier`. O valor de retorno do método (se houver), juntamente com a sinalização de método executado, é enviado como uma mensagem para o Stub (6).

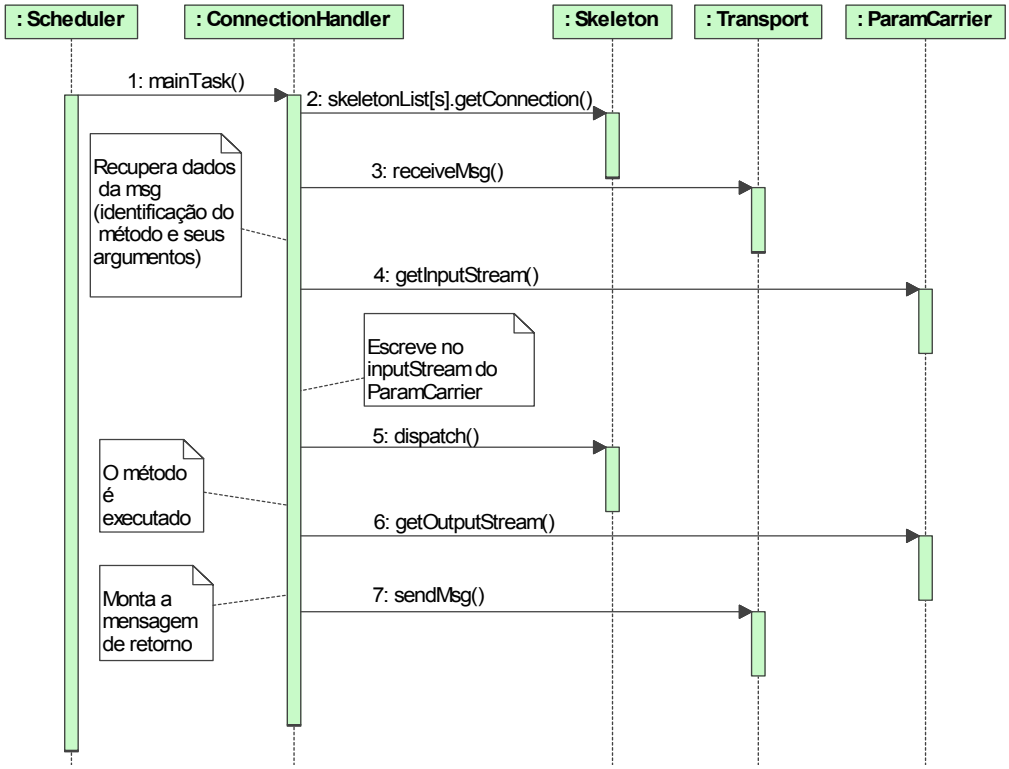

Figura 4.9: Diagrama de seqüência do suporte à execução do método remoto (lado servidor)

O mecanismo para atender restrições de tempo-real está associado à *thread* `ConnectionHandler`, uma *thread* periódica e de tempo-real. Assim como no lado cliente, um objeto `RealTimeParameters` é usado para que a aplicação possa definir restrições temporais para as operações de comunicação realizadas no processo de comunicação com o `Registry` e com o usuário remoto.

Do ponto de vista do cliente é possível definir a latência para receber a resposta de um método executado remotamente pela expressão abaixo:

$$L_C = t_{Stub} + t_{RS} + t_{WH} + t_{Sched} + t_{Skeleton} + t_{Rmote} + t_{Rply} + t_{RC} \quad (4.1)$$

onde,

t_{Stub} – Tempo para o `Stub` empacotar os dados de chamada do método e enviar

t_{RS} – Tempo gasto no receptor para processar a mensagem enviada pelo `Stub`

t_{WH} – Tempo de espera para que a *thread* `ConnectionHandler` seja ativada

t_{Sched} – Tempo de execução do escalonador de tarefas (nodo servidor) – pior caso

$t_{Skeleton}$ – Tempo de execução do `ConnectionHandler` e do `Skeleton`

t_{Rmote} – Tempo de execução do método remoto

t_{Rply} – Tempo gasto pelo nodo servidor processando e enviando o retorno do método

t_{RC} – Tempo gasto pelo nodo cliente processando a mensagem de retorno

O tempo t_{WH} depende do escalonamento de tarefas no servidor e facilmente se torna o elemento mais importante nesta latência. Para assegurar o atendimento às restrições de tempo-real, é preciso que as tarefas aperiódicas e tratadoras de eventos executadas no servidor sejam resolvidas por um tratador periódico e previsível. Assim, o período de ativação do `ConnectionHandler` define o pior caso de t_{WH}. O problema com a escolha deste período é que, se ele for elevado, prejudica a latência na execução de métodos remotos, ou mesmo no tratamento de eventos ligados a outras tarefas no servidor, o que pode resultar em mau funcionamento da aplicação. Por outro lado, se este período for muito reduzido, este tratador consumirá um elevado percentual da capacidade de processamento da CPU podendo ainda ocasionar perda de *deadline* para alguma tarefa de menor prioridade.

O aplicativo que gera as classes stub e skeleton automaticamente não faz parte do escopo deste trabalho. Uma investigação em geração de código está sendo conduzida no contexto do trabalho (WEHRMEISTER, 2008).

A versão atual da plataforma utilizada SASHIMI-FemtoJava não prevê o mecanismo de interface de Java, o que limita o grau de abstração possível na construção de aplicações que usem o serviço aqui descrito. No capítulo de exemplos e aplicações será mostrado como a implementação contornou esta limitação.

4.4 Objetos implementados em Hardware

O particionamento de aplicações em hardware e software é um problema que, pela sua importância, tem ocupado bastante espaço na literatura. Busca-se uma abordagem de mais alto nível (BALARIN, 1997) na especificação dos sistemas procurando abrir espaço para explorações de diferentes estratégias.

A fronteira entre o hardware e o software tem um papel importante no atendimento de requisitos de um sistema embarcado, como desempenho, previsibilidade e energia. Frequentemente este limite é definido nos estágios iniciais do projeto, o que obriga a decisões por vezes prematuras e inadequadas. Além disso, mover esta fronteira vai ficando cada vez mais difícil à medida que o projeto vai avançando em seus estágios. Contraditoriamente, as melhores decisões podem ser tomadas nos estágios finais do projeto, quando se tem uma melhor compreensão do impacto de cada implementação. Este conflito é amenizado quando as ferramentas de projeto simplificam a migração de comportamento do hardware para o software e vice-versa, pela definição de um modelo uniforme de programação para as duas implementações.

Nesta seção é apresentada uma arquitetura para especificar objetos tanto em hardware quanto em software no contexto do padrão RTSJ. A estratégia utilizada oferece flexibilidade na escolha de ambas as implementações, hardware ou software, permitindo prorrogar o particionamento hardware/software para o final do projeto do sistema.

Duas abordagens podem ser consideradas para implementar um objeto em hardware. Na primeira considera-se que qualquer objeto poderia ser implementado em hardware. Esta é a abordagem seguida pelo JavaMen Framework (BORG, 2006). Uma interface em hardware é especificada para funcionar como uma camada de abstração que faz com que um dispositivo se comunique com os outros e com o software da mesma maneira que um objeto Java qualquer.

Uma alternativa seria restringir o objeto implementado em hardware, definindo que ele é do tipo *thread* (ou tarefa), conforme o conceito da API-RTSJ (WEHRMEISTER, 2004) usada no *middleware*. Neste caso, uma *thread* (classe `RealtimeThread`) não é somente um fluxo de execução simples (SILBERSCHATZ, 2004), mas inclui os dados usados por este fluxo. As tarefas aqui utilizadas são tipicamente independentes, carregam considerável informação de estado, têm endereço de memória próprio e interagem somente através de chamada de métodos.

Limitar o objeto hardware a uma tarefa não é uma restrição séria, já que o que se pretende ao implementar tal objeto é melhorar desempenho e energia, em troca de um custo maior em área. Um objeto ativo (tarefa) tem a execução como sua propriedade essencial e seria a principal utilização para objetos implementados em hardware, se não a única.

A plataforma HybridThreads (ANDREWS, 2004) segue uma linha similar. Embora não seja orientada a objetos e nem tenha propriedades de tempo-real, HybridThreads permite definir *threads* em hardware (VHDL ou VERILOG) e ativá-las a partir de uma aplicação Linux usando o padrão pthreads.

A vantagem da segunda abordagem, baseada em objeto tarefa, é a simplicidade. Um componente de hardware para fazer a interface com um hardware que implemente uma `RealtimeThread` seria mais simples que uma interface genérica para invocar qualquer método de um objeto qualquer.

Este trabalho segue a abordagem do HybridThreads, definindo um componente de hardware como interface entre o processador e a *thread* em hardware, chamado HwTI – *Hardware Thread Interface*. Outro componente hardware deve implementar o comportamento da *thread* e é chamado HwTB – *Hardware Thread Behavior*. A Figura 4.10 mostra os dois componentes e a sua ligação ao barramento local do processador. O HwTI faz parte da plataforma disponível para o projetista, enquanto que o HwTB faz parte da aplicação e deve ser escrito pelo projetista usando uma linguagem de descrição de hardware.

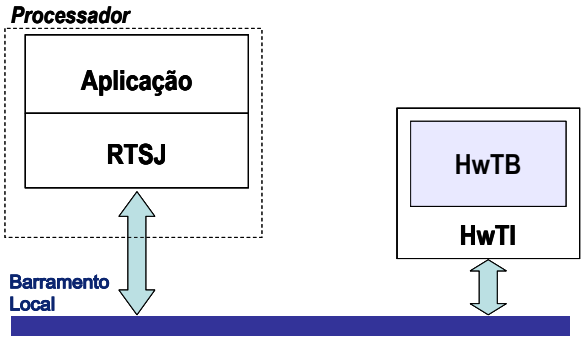

Figura 4.10: Arquitetura simplificada da *thread* em hardware

Algumas pesquisas recentes têm abordado a geração automática de hardware a partir da linguagem Java (MACKETANZ, 1998) (ANDERSSON, 2005). Este trabalho não dispensa o uso destas ferramentas, mas o *middleware* não depende da maneira como o hardware de HwTB é gerado (seja usando uma HDL tradicional ou uma linguagem de alto nível como Java).

4.4.1 Arquitetura de hardware

O componente HwTI define operações de comunicação entre a *thread* e o resto do sistema. Ele possui uma interface com o barramento do processador e outra com a *thread* hardware. O lado do processador oferece registradores mapeados em memória, acessíveis pelo software. O outro lado, conectado ao HwTB, oferece os sinais da Tabela 4.3.

A HwTI acessa a memória RAM através do barramento local do processador. Quando uma *thread* hardware requer uma operação na memória, a HwTI armazena o endereço (e o dado, se for uma escrita) e espera pela próxima janela de acesso à RAM, sinalizada pelo barramento local. No FemtoJava multiciclo esta espera pode ser de até 14 ciclos de relógio (*clock*), que é o tempo de execução da instrução mais longa.

Tabela 4.3: Sinais de comunicação entre HwTB e HwTI

Signal	Description
intrfc2thrd_value	Interface retorna para a *thread* o valor lido da memória
intrfc2thrd_function	Interface informa à *thread* a função a executar
intrfc2thrd_rdy2recv	Interface informa que está pronta para receber o próximo opcode
thrd2intrfc_address	A *thread* hardware informa o endereço de memória que a *thread* pretende ler ou escrever
thrd2intrfc_value	A *thread* hardware informa o dado a ser escrito na memória, ou o identificador do método a ser invocado
thrd2intrfc_function	A *thread* hardware informa a função a ser executada pelo processador
thrd2intrfc_opcode	A *thread* hardware informa a operação

4.4.2 Arquitetura de software

São definidos alguns opcodes, que servem para que a *thread* selecione operações oferecidas pela HwTI, e funções, que são serviços realizados pelo processador ou outro hardware e disponibilizados pela HwTI. Algumas funções também podem ser solicitadas pelo processador à *thread* hardware. A Tabela 4.4 mostra uma lista de funções e opcodes e seus significados. Usando os opcode GETDATA e PUTDATA a *thread* pede à HwTI para acessar a RAM, fazendo uma operação de leitura ou de escrita. Os métodos RTSJ podem ser invocados usando o opcode CALL. Métodos da aplicação (implementados em software) podem também ser invocados pela *thread* hardware usando o opcode RUN_METHOD.

Tabela 4.4: Funções e opcodes para comunicação com a *thread* hardware

Opcode (*Thread* to Interface)	**Descrição**
GETDATA	A *thread* hardware pede para ler uma posição na memória
PUTDATA	A *thread* hardware pede para escrever uma posição na memória
CALL	A *thread* hardware pede para executar um serviço (função da RTSJ) no processador
Functions (Interface to *Thread*)	
START	Inicia a execução da *thread* hardware
RESET	Reinicializa os registradores da *thread* hardware
CONTINUE	Continua a execução da *thread* hardware após um pedido de WAIT
Functions (*Thread* to Interface)	
WAIT_FOR_NEXT_PERIOD	Executa o método RealtimeThread.waitForNextPeriod()
WAIT_FOR_EVENT	Executa o método RealtimeThread.waitForEvent()
RUN_METHOD	Executa o método selecionado pelo sinal thrd2intrfc_value
EXIT	Informa que a *thread* hardware finalizou a sua execução

Do ponto de vista do software, a *thread* em hardware é encapsulada por um objeto que estende a classe `HwRealtimeThread`, que por sua vez estende a classe `RealtimeThread` da RTSJ. Assim, esta *thread* hardware é controlada da mesma forma que uma *thread* qualquer implementada em software, reusando os escalonadores já disponíveis na implementação RTSJ disponível (API-RTSJ).

Sendo uma classe filha da `RealtimeThread`, a classe `HwRealtimeThread` implementa o método `mainTask()`, equivalente ao `run()` em uma *thread* Java padrão. Este método é executado quando o escalonador ativa a *thread* e encapsula o protocolo de comunicação com a HwTI.

O método `HwRealtimeThread.mainTask()` segue os seguintes passos:

1. Carrega o Registrador 'ponteiro de atributos' – Este registrador deve conter o endereço na memória onde os atributos do objeto iniciam. O componente HwTB usa este valor para acessar os atributos do objeto.
2. Envia um comando RESET para a *thread* hardware.
3. Envia um comando START para a *thread* hardware.
4. Espera por um comando CALL vindo do HwTI. Isto permite que a *thread* hardware se sincronize com o processador, se necessário.

Se a *thread* (`HwRealtimeThread`) sofrer preempção, o HwTB pode continuar executando. Se, neste meio tempo, o HwTB faz um CALL ele vai ficar bloqueado até que o `mainTask()` recebe o opcode e retorne a resposta, isto é, quando o escalonador der o processador de volta para aquela *thread*.

O método `mainTask()` é implementado na classe `HwRealtimeThread` como um método final e não pode ser sobrescrito.

Se a *thread* é periódica, ela vai chamar o WAIT_FOR_NEXT_PERIOD (vide Tabela 4.4) depois da execução. Quando a função `mainTask()` recebe o CALL, ele

executa o método `waitForNextPeriod()`, fazendo com que o escalonador entregue o processador para outra tarefa e agende a *thread* hardware para seu próximo período de ativação.

Uma *thread* hardware também pode ser do tipo esporádica ou não-periódica. Neste caso, ela irá invocar o método `waitForEvent()` para suspender a execução. Uma *thread* sempre pode finalizar usando a função EXIT. O método `mainTask()`, então, invoca o `ReatimeThread.finish()`, informando o escalonador para não mais ativar aquela *thread*.

A título de exemplo, a Figura 4.11 mostra a máquina de estados (FSM) para uma implementação de uma *thread* hardware periódica. Quando iniciada, a *thread* (HwTB) precisa esperar por um START, vindo do processador, através da HwTI. Depois do START, a *thread* lê a RAM para pegar o endereço inicial dos atributos do objeto `HwRealtimeThread`. A partir deste ponto, a hardware *thread* pode ler e escrever na memória, usando os opcodes GETDATA e PUTDATA. Uma vez que este exemplo é de uma *thread* periódica ao final do laço de repetição o método `waitForNextPeriod()` é executado. Para aguardar o próximo momento de ativação, comandado pelo escalonador, a *thread* hardware espera pela função CONTINUE.

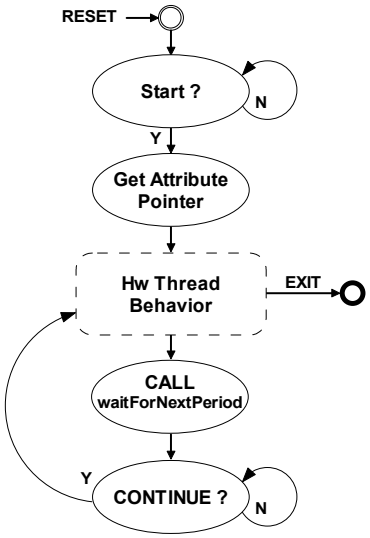

Figura 4.11: FSM para uma *thread* periódica em hardware

4.5 Migração de tarefas

Alocação dinâmica de tarefas (descrito na seção seguinte) tem se mostrado uma técnica promissora para obter balanço de carga entre as unidades de processamento de um MPSoC (BRIÃO, 2008) (ACQUAVIVA, 2008), permitindo minimizar algumas métricas como tempo total de execução, potência ou mesmo temperatura. Para permitir alocação dinâmica, é necessário um mecanismo de migração de tarefas. Alocação e migração de tarefas são serviços relacionados ao balanceamento de carga. Uma tarefa é alocada antes que ele comece a executar e pode migrar durante a sua execução.

Em sistemas distribuídos tradicionais a migração de tarefas usualmente é implementada no nível do *middleware*, funcionando como um serviço totalmente transparente para as aplicações. Em um sistema embarcado, todavia, o serviço de migração de tarefas tem um compromisso entre transparência e implementação leve e eficiente. Além disso, em aplicações de tempo-real, o *overhead* da migração deve ser considerado para evitar perdas de *deadlines* quando uma tarefa está migrando de um processador para outro.

A base para o serviço de migração de tarefas do *middleware* foi desenvolvida como parte do trabalho de Barcelos (2008). Naquele trabalho a migração é feita dentro do espaço do usuário, como em (BERTOZZI, 2006). O modelo de migração de tarefas proposto por Barcelos (2008) utiliza comunicação por troca de mensagens e é baseado na plataforma FemtoJava e na API RTSJ. Toda sua implementação é em software, não recorrendo a nenhuma função adicional em hardware. Para efetuar a comunicação Barcelos utiliza diretamente as primitivas de acesso ao hardware de comunicação da NoC SoCIN.

Como já foi dito anteriormente, a *thread* (ou tarefa) na API RTSJ não é construída em tempo de execução, mas é definida em tempo de projeto, juntamente com os objetos acessados por ela. Portanto, o seu espaço de endereçamento é conhecido *a priori*. Se estes dados são compartilhados por outros fluxos de execução, cabe à aplicação garantir a coerência destes dados após a migração. Esta característica foi mantida na implementação dentro do *middleware*.

Para tornar a migração de tarefas um serviço do *middleware* todas as operações de comunicação foram isoladas e submetidas à API de comunicação. Além disso, toda a comunicação entre os dois processadores envolvidos na migração foi submetida à disciplina de uma *thread* periódica de custo fixo. Com estas propriedades, o serviço de migração torna-se independente da rede utilizada e adquire a previsibilidade necessária para ser usado em aplicações de tempo-real.

A Figura 4.12 mostra o diagrama de classes do serviço. O serviço de migração é acionado por outro serviço, alocação de tarefas, que toma a decisão de que tarefa mover e para onde. O usuário deste serviço tanto pode ser a aplicação quanto outro serviço, como a alocação de tarefas, por exemplo.

A classe `MoveThread` contém os métodos públicos `sendThread()` e `receiveThread()`. Eles são utilizados pelo usuário do serviço de migração para ativar o envio ou a recepção de uma *thread*. Ambos retornam `FALSE` se o serviço estiver indisponível, cabendo ao solicitante efetuar nova tentativa. Quando o método `sendThread()` é executado, um tratador de eventos (`MvThreadHandler`) é configurado para efetuar o envio da *thread* informada na chamada. O primeiro trecho da *thread* é enviado e a partir daí cada vez que o receptor envia uma confirmação de recebimento um evento é gerado, fazendo o próximo trecho ser enviado. O tratador de eventos, conforme a RTSJ, é executado segundo a política de tratamento de eventos assíncronos definida por `AsyncEventMechanism`. Embora este procedimento aumente a latência na migração de uma tarefa, ele assegura o equilíbrio no uso do processador, não interferindo no funcionamento das demais *threads* de tempo-real sendo executadas.

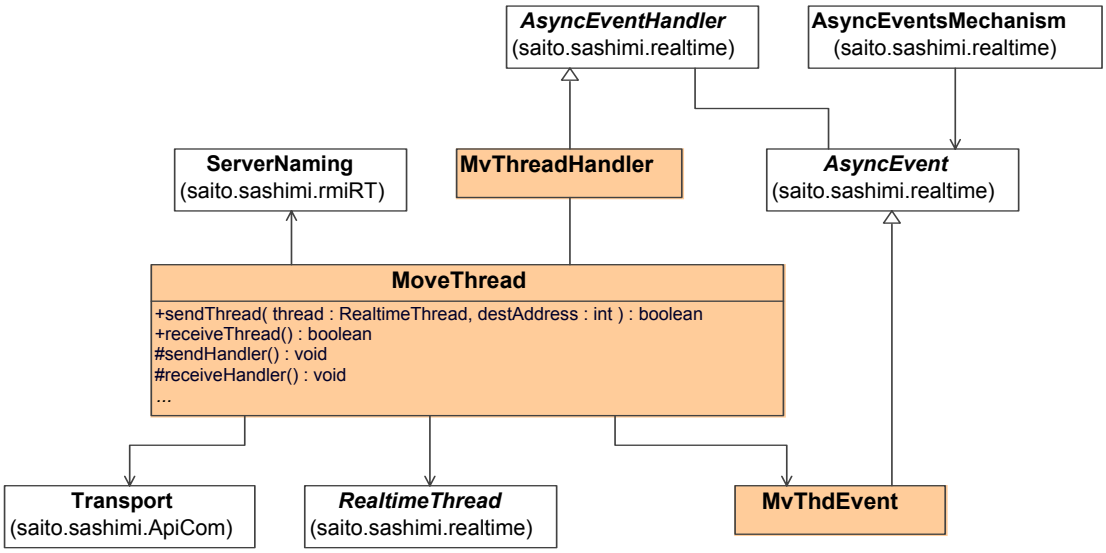

Figura 4.12: Diagrama de classes do serviço de migração de tarefas

Enviar uma *thread* consiste em mandar o código (métodos) e atributos da *thread* e dos objetos contidos nela, bem como a pilha utilizada pela *thread*. Sendo uma máquina de pilha, as variáveis de Java estão contidas na pilha, de modo que todo o contexto da tarefa é replicado quando a pilha é enviada. Esta propriedade simplifica o envio do contexto, tornando o uso de *checkpoints* desnecessário. Em outras palavras, toda a memória utilizada pela *thread* está confinada aos atributos de suas classes e à pilha, que contém as variáveis de métodos. No caso do SASHIMI-FemtoJava, a localização dos atributos de classe é conhecida em tempo de pós-compilação, o que permite instrumentar a classe `RealtimeThread` com estas informações antes mesmo da aplicação iniciar sua execução. Fica, então, a pilha para ser resolvida em tempo de execução. Esta tarefa é simplificada porque a classe `RealtimeThread` conhece a base e o tamanho da sua pilha, permitindo que a classe `MoveThread` consulte e use estes valores na hora de mover a *thread*. Assim, a modificação importante a ser feita na plataforma é dotar o SASHIMI da capacidade de coletar as referências para os atributos da *thread* e de seus objetos e instrumentar a instância da classe `RealtimeThread`, em atributos previamente definidos para esta função, com estes valores. Igualmente, a localização do código da *thread* e seus objetos deverá ser informada à *thread* segundo o mesmo mecanismo.

A Figura 4.13 mostra, de uma maneira simplificada, a seqüência de ações no processo de migrar uma *thread*. Cada ação de envio indicada na figura, na verdade é dividida em tantas mensagens quantas forem necessárias em função da área de memória envolvida. Para cada mensagem o receptor envia uma confirmação (ACK). Antes de migrar, uma *thread* precisa ser removida do escalonador de tarefas. Isto só precisa ser feito antes de iniciar o envio dos atributos e da pilha. Uma vez no destino a *thread* pode iniciar a execução assim que a sua pilha for restaurada. No próximo instante de ativação da *thread*, definido em um de seus atributos, o escalonador irá restaurar a pilha da *thread* que foi recebida no processo de migração. A *thread*, então, continuará executando do ponto onde foi interrompida no processador anterior.

No processador destino o usuário do serviço deve ativar a recepção invocando o método `receiveThread()` que irá preparar um tratador de eventos, também do tipo `MvThreadHandler`, para receber a *thread*.

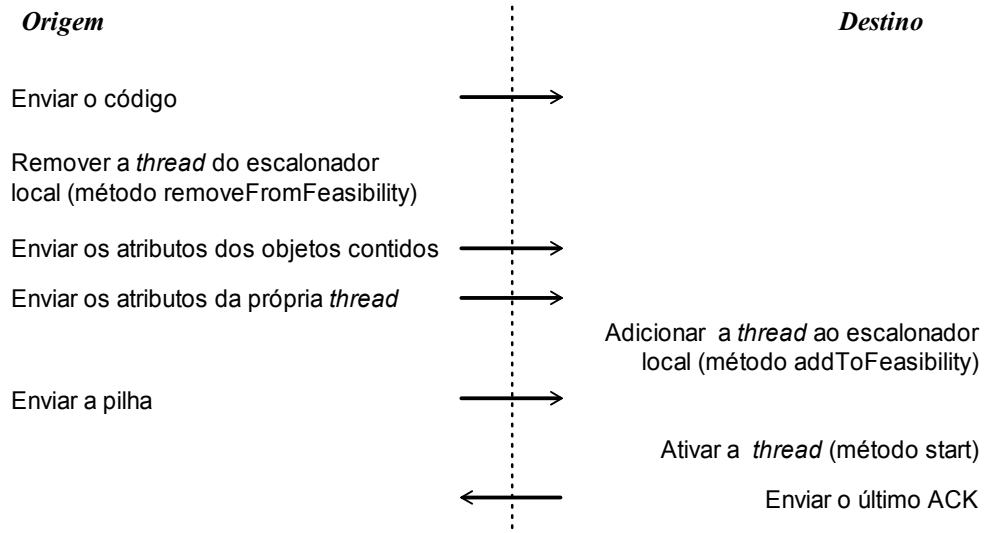

Figura 4.13: Fluxo simplificado da migração de uma tarefa

O tratador de eventos `MvThreadHandler` funciona como uma casca, que é ativada pela API-RTSJ. A implementação do comportamento de receber ou enviar a *thread* está dentro da classe `MoveThread` nos métodos protegidos `receiveHandler()` e `sendHandler()`.

Se a *thread* migrada possui canais de comunicação estabelecidos, eles terão que ser tratados pelo serviço de migração. Na verdade, a API de comunicação precisa ser informada para que as conexões sejam restabelecidas a partir da nova localização da *thread*. Ao receber um pacote de `MIGRATE` em uma conexão, a APICOM desfaz a conexão internamente e efetua um outro `openConnection()` com o novo endereço informado, mantendo para a aplicação o mesmo numero de conexão. Assim, do ponto de vista do usuário do serviço de comunicação, a conexão anterior com a *thread* que migrou é mantida sem alterações. Se a *thread* que migra contém um objeto tipo `Skeleton`, significando que ela publicou um objeto para invocação remota, um `rebind()` terá que ser feito junto ao `Registry`. Desta forma, os próximos a procurar por este objeto terão a referência correta para ele. Este tratamento não chegou a ser validado em código para a plataforma FemtoJava, ou seja, o serviço de migração implementado deixa para a aplicação o tratamento de problemas com conexão em caso de migração de tarefas.

4.6 Alocação de tarefas

O serviço de alocação de tarefas consiste em distribuir *threads* utilizando uma função de distribuição. O problema da escolha da função de distribuição é investigado em (WRONSKI, 2006) e em (BRIÃO, 2008), além de outras pesquisas na literatura e

não faz parte do escopo deste trabalho. Ao *middleware* cabe oferecer uma interface para que o serviço seja utilizado pela aplicação, facilitando a escolha do algoritmo que efetua a distribuição/alocação.

A Figura 4.14 mostra o diagrama de classes do serviço. Em primeiro lugar, a `RealtimeThread` da RTSJ é estendida dando origem à `XtdRealtimeThread`. Esta nova classe incorpora as propriedades que serão necessárias para que a *thread* possa informar ao serviço os seus requisitos de alocação, tais como memória necessária e ocupação do processador. Na verdade, a ocupação já é uma informação possível de calcular com os parâmetros da RTSJ, uma vez que uma `RealtimeThread` periódica sabe o seu tempo máximo de execução (WCET) e o período. A ocupação é a razão entre WCET e o período. Assim, ficam os demais requisitos armazenados em `AllocRequirements`.

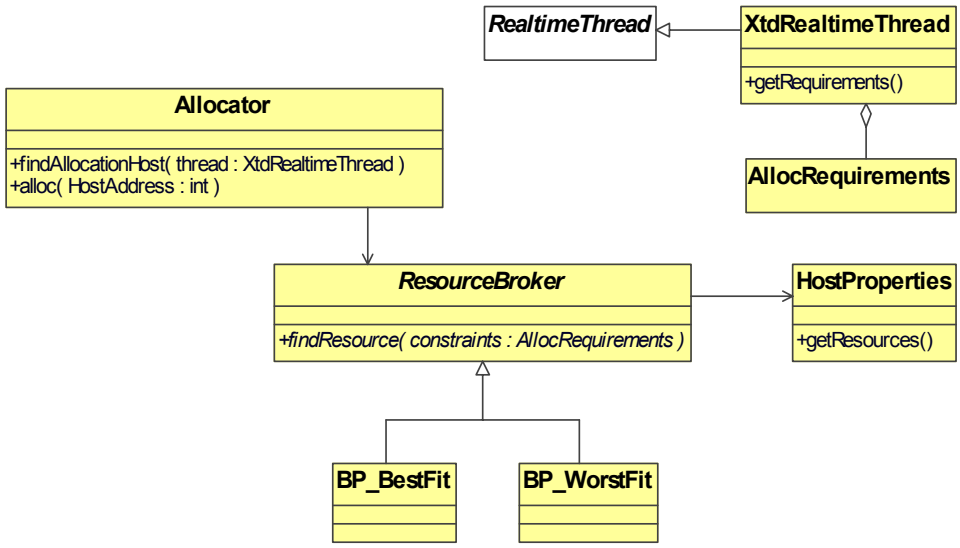

Figura 4.14: Diagrama de Classes para serviço de alocação de tarefas

O serviço de alocação propriamente dito está implementado nas classes `Allocator` e `ResourceBroker`. Estas classes devem estar instanciadas no *core* mestre, processador que centraliza a função de distribuir as aplicações pela rede (BRIÃO, 2008). O padrão de projeto Strategy (GAMMA, 2000) é usado para que o `ResourceBroker` possa incorporar diferentes algoritmos de alocação. No diagrama da figura foram incluídos o *Bin-Packing Best Fit* (`BP_BestFit`) e o *Bin-Packing Worst Fit* (`BP_WorstFit`). O método `findResource()`, abstrato na classe `ResourceBroker`, é implementado nas classes concretas que efetivamente realizam a busca pelo local de alocação da tarefa.

Na literatura, o problema chamado de *bin-packing*, ou BP, (JONHSON, 1973) é formado por um conjunto de itens que devem ser alocados em um conjunto de recipientes. O BP é um problema NP-Completo e por isso várias heurísticas são propostas na literatura para a sua solução. Segundo Brião (2008) as heurísticas *Best Fit* (BF) e *Worst Fit* (WF) são de particular interesse para NoCs. O BF tende a concentrar as tarefas em alguns nodos da rede, permitindo desligar os demais que estiverem sem uso. Já o WF tende a distribuir as tarefas uniformemente na rede, possibilitando que uma estratégia de DVS/DFS minimize a freqüência de operação dos processadores.

Finalmente, a classe `HostProperties` encapsula as propriedades do nodo local da rede e pode informar a disponibilidade de recursos. Esta classe tem acesso ao serviço de Supervisão (nível de estrutura do *middleware*) e deve haver uma instância dela em cada nodo da rede.

A Figura 4.15 complementa a descrição do serviço com o diagrama de seqüência para um procedimento de alocação de uma tarefa. O serviço tem início (1) com a invocação do método `findAllocationHost()`, que passa uma referência da tarefa a ser alocada como parâmetro. Esta invocação pode ser feita por um serviço de inicialização do sistema, que faria a solicitação de alocação para todas as tarefas. À medida que outras tarefas forem sendo adicionadas, a sua inclusão passará pela invocação deste método.

Um serviço adicional, não descrito neste trabalho, deve ser acionado quando uma aplicação for removida. Neste caso, além de remover as tarefas desativadas, o serviço deve fazer uma realocação das tarefas restantes recorrendo, eventualmente, à migração de algumas tarefas ativas. Brião (2008) fez experimentos utilizando os mesmos algoritmos para esta realocação, mostrando que eles são eficientes tanto para alocar inicialmente as tarefas quanto para fazer o reposicionamento posterior.

Dentro do `findAllocationHost()` os requisitos de alocação da *thread* são obtidos (2) e usados para orientar a busca (3) pelo processador alvo para a alocação. Na implementação da função de alocação (4) cada nodo da rede pode vir a ser consultado para avaliar o melhor destino para a tarefa, dependendo da função. No final da busca, o endereço do nodo selecionado é informado (6) para que a alocação seja finalizada (7).

O serviço de alocação de tarefas está especificado em modelos UML (diagramas de classes e de seqüência), mas não chegou a ser codificado para a plataforma FemtoJava. Além dos algoritmos BF e WF citados aqui, outras funções de alocação são avaliadas em (BRIÃO, 2008).

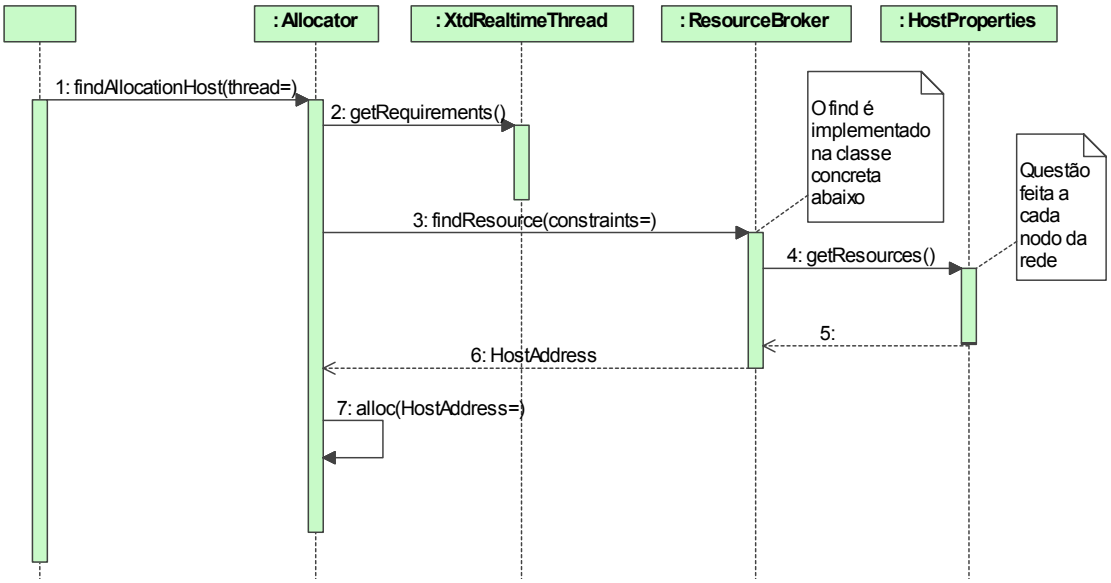

Figura 4.15: Diagrama de Seqüência para serviço de alocação de tarefas

4.7 Gerenciamento de energia

Esta seção mostra como o DVS/DFS (*Dynamic Voltage Scaling / Dynamic Frequency Scaling*) é disponibilizado e como os serviços e recursos do *middleware* se beneficiam dele. O mecanismo de DVS atua apenas nos processadores da rede e é somente neles que se pode gerenciar energia. Esta decisão decorre da análise de trabalhos anteriores (ATITALLAH, 2006) (BARCELOS, 2008) que mostram que uma componente significativa da energia em um sistema MPSoC tipo NoC está concentrada em seus processadores.

DVS tem sido alvo de um esforço significativo de pesquisa nos últimos anos. DVS tenta atacar o conflito entre desempenho e economia de baterias, considerando duas importantes características: (1) a taxa de computação de pico necessária é muito maior que o *throughput* médio que precisa ser mantido; e (2) os processadores são baseados em tecnologia CMOS. A primeira característica significa que alto desempenho só é necessário por uma pequena fração de tempo, enquanto no resto do tempo um processador de baixo desempenho e baixo consumo de energia seria suficiente. Pode-se obter baixo desempenho simplesmente reduzindo a freqüência de operação do processador quando o desempenho máximo não é necessário. DVS vai além e reduz a tensão do processador juntamente com a freqüência. Isto é possível porque a lógica CMOS, usada na maioria dos processadores atuais, tem uma freqüência máxima de operação que é dependente da tensão de alimentação. Assim, quanto o processador é usado em uma freqüência reduzida ele pode operar com uma tensão de alimentação também menor. Uma vez que a energia dissipada por ciclo em circuitos CMOS varia quadraticamente com a tensão de alimentação ($E \alpha V^2$), DVS potencialmente pode prover uma boa economia de energia em uma rede que pode fazer variar a freqüência e a tensão de alimentação de seus processadores.

Para introduzir adaptação automática na freqüência do processador na plataforma utilizada foram feitas modificações em dois pontos. Primeiramente a arquitetura do processador foi alterada introduzindo um registrador de freqüência de operação, além de modificar o relógio de tempo-real para lidar com esta nova possibilidade. Em seguida a API RTSJ foi modificada da maneira que os parâmetros e operações dependentes da freqüência passassem a monitorar a freqüência atual do processador, como programação de *timers*, por exemplo.

O mecanismo de DVS é disponibilizado para o desenvolvedor através do escalonador de tarefas. A API RTSJ permite que a aplicação escolha o algoritmo de escalonamento de tarefas a ser utilizado, através do padrão de projeto Strategy (GAMMA, 2000). Com isso, a efetiva utilização do recurso de DVS/DFS no *middleware* se dá através da escolha de um escalonador de tarefas capaz de fazê-lo. Nada mais precisa ser feito pela aplicação para que o serviço de DVS funcione; a não ser especificar corretamente as tarefas da aplicação, definindo os valores de WCET (*Worst Case Execution Time*) e de *deadline*. O algoritmo mostrado adiante se encarrega de definir a mínima freqüência de operação para o processador que atenda os *deadlines* da aplicação.

Mesmo nos casos em que a plataforma não permita reduzir a tensão de operação do processador, reduzir a freqüência de operação levará à redução da potência dinâmica (e da energia), se os processadores apresentam ocupação inferior a 100%. Em outras palavras, no contexto em que as tarefas em execução no processador são periódicas e

apresentam uma ocupação baixa, durante algum tempo o processador fica em *idle*, ou seja, executa instruções de espera, dissipando potência sem executar nenhuma tarefa. Neste caso, ao baixar a freqüência de operação do processador não se prolonga a execução das tarefas, mas somente este tempo ocioso será reduzido ou deixará de existir, permitindo reduzir também a energia do sistema.

Em (PILLAI, 2001) são apresentados alguns algoritmos que incorporam o DVS no escalonador de tarefas, obtendo a economia de energia propiciada pelo DVS e atendendo aos *deadlines* das tarefas. Alguns destes foram introduzidos no escalonador de tarefas para que o processador pudesse automaticamente operar em modo de energia mínima, atendidas as restrições temporais. O DVS integra o mecanismo do *middleware* de auto-adaptação de hardware para atender as restrições dadas pela aplicação.

Para este trabalho foram avaliados os algoritmos *Static Voltage Scaling* e *Cycle-Conserving* DVS. De uma forma geral eles analisam o grau de ociosidade do processador e regulam a freqüência em função disso. Para a implementação dos dois algoritmos o escalonador EDF (*Earliest Deadline First*) foi tomado como referência.

O *Static Voltage Scaling* (S-DVS) consiste em recalcular a freqüência mínima que ainda atende aos *deadlines* cada vez que uma tarefa é incluída no escalonador, seguindo o teste de escalonabilidade para cada tipo de escalonador (BURNS, 1997). Esta abordagem é mais simples de implementar e tem impacto praticamente nulo sobre o tempo de execução do escalonador. Sua limitação é não considerar que eventualmente alguma tarefa use um tempo menor que o seu WCET.

O algoritmo *Cycle-Conserving* (CC-DVS) procura superar a limitação do estático. Inicialmente, ele assume o WCET para todas as tarefas e a freqüência é selecionada como no algoritmo estático. Ao término de cada tarefa, a sua utilização é substituída por uma utilização real e a freqüência é recalculada. Quando uma nova tarefa é lançada, novamente o pior caso de utilização é adotado e a freqüência, então, é escolhida de acordo.

Diferentemente das seções anteriores, aqui não serão usados diagramas de classe ou de seqüência para explicar o serviço, que consiste basicamente na inclusão de novas classes que implementam o escalonador (`Scheduler`) na API RTSJ. Foram apresentados os algoritmos usados no DVS e onde eles atuam dentro da estrutura do *middleware*. A seguir, três aplicações são mostradas como ilustração do uso do conceito de escalonadores DVS/DFS. Em todos os casos é usado um escalonador EDF acrescido do algoritmo CC-DVS. O tempo gasto para a transição no valor da freqüência quando o DVS atua não foi considerado.

A Figura 4.16 mostra um diagrama de tempo para a primeira delas, que é uma migração de tarefa. O eixo superior mostra o que está sendo executado no processador, conforme legenda na própria figura. O eixo inferior mostra a freqüência de operação do processador normalizada em relação à freqüência máxima. No primeiro período de execução apenas a tarefa Task1 está em execução e o processador opera a 60% da freqüência máxima. Segue-se um intervalo em que a tarefa Task2 conclui a sua migração e é adicionada ao escalonador (1). Em (2) o escalonador é executado pela primeira vez após a inclusão de Task2. Neste momento é detectado o aumento na ocupação (utilização) do processador e a freqüência é elevada para assegurar que o sistema seja escalonável.

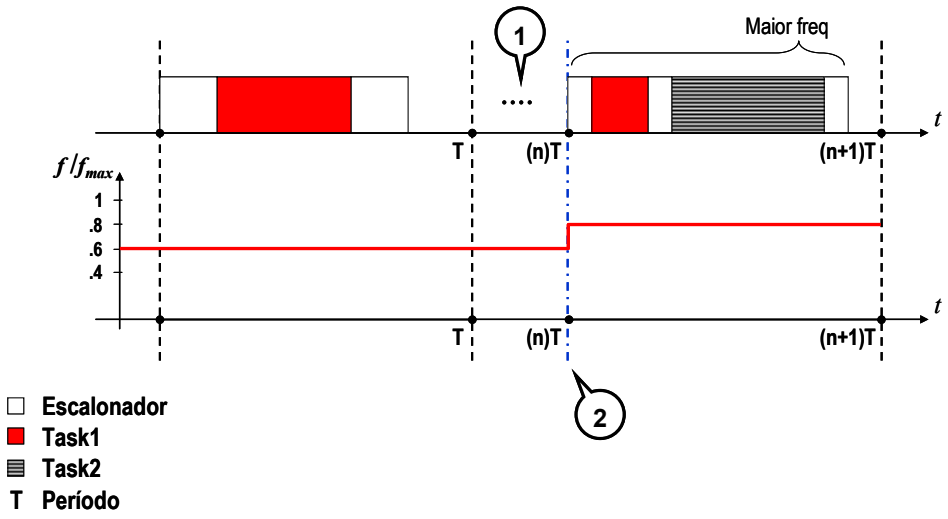

Figura 4.16: Escalonador DVS/DFS - migração de tarefa

A Figura 4.17 ilustra o caso em que uma tarefa implementada em software, um filtro FIR, por exemplo, é substituída por uma implementação equivalente em hardware. O exemplo se inicia com a freqüência do processador em seu valor máximo, onde estão sendo executadas uma tarefa Task1 e a versão software do FIR. No intervalo (1) a *thread* software é removida e a sua versão hardware é ativada. Em (2) o escalonador, pela análise do WCET de todas as tarefas, detecta que a utilização do processador reduziu e determina a nova freqüência de operação, 80% da freqüência máxima. O tempo de execução do FIR hardware, indicado na figura, se refere à comunicação com o componente hardware, executada pelo processador.

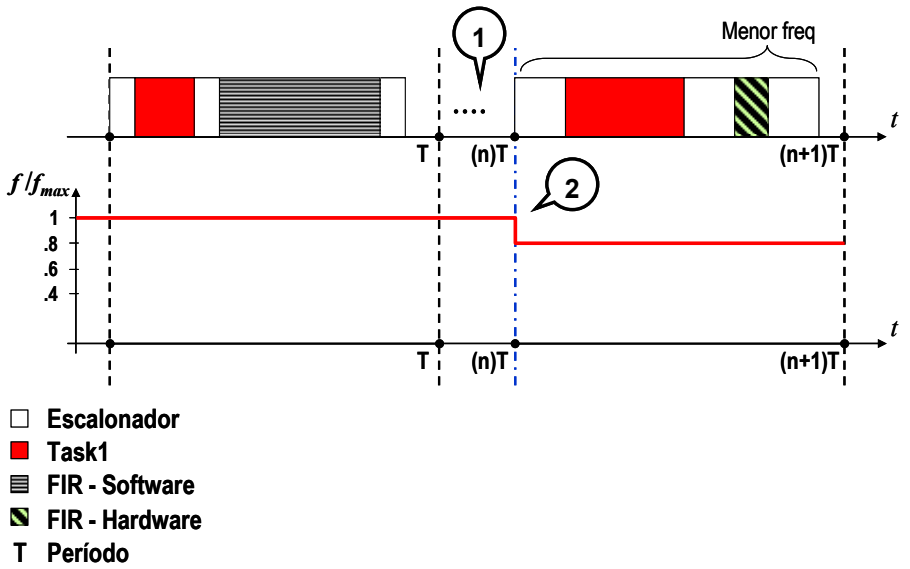

Figura 4.17: Escalonador DVS/DFS - tarefa em hardware

Nos dois exemplos anteriores as tarefas consomem todo o tempo previsto no seu WCET, de modo que o algoritmo CC-DVS apresenta o mesmo resultado que o S-DVS. O exemplo a seguir, visto na Figura 4.18, mostra o caso em que a chegada de uma mensagem pode alterar a ocupação do processador.

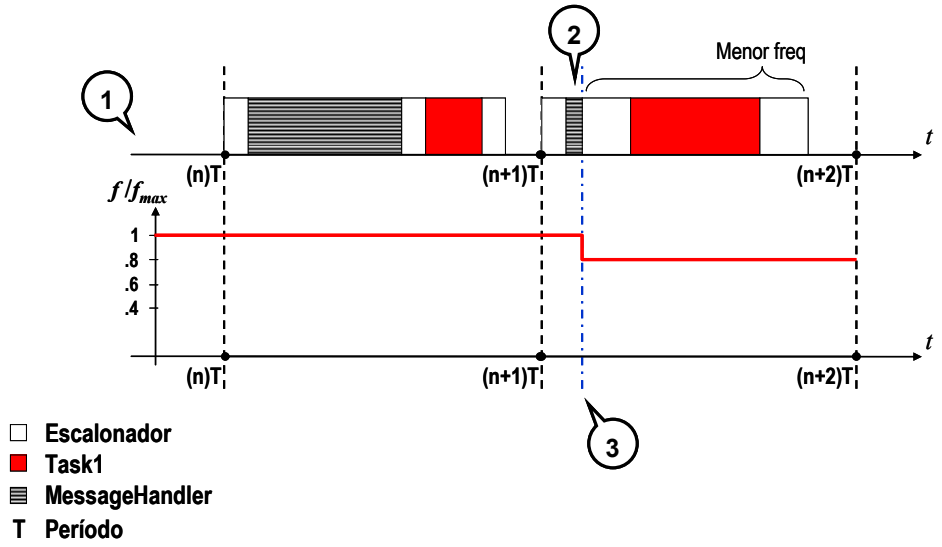

Figura 4.18: Escalonador DVS/DFS - comunicação

Antes da execução do primeiro ciclo (1) uma mensagem chega ao *buffer* do serviço de comunicação. O `MessageHandler`, encarregado de processar a mensagem recebida e entrega-la à aplicação destino, durante o primeiro ciclo consome todo o tempo previsto para sua execução. Para o segundo ciclo não há mensagens a serem processadas e o `MessageHandler` consome um tempo mínimo (2). Na próxima execução do escalonador, ele detecta que o `MessageHandler` não utilizou todo o seu WCET e, ao avaliar a utilização do processador, reduz a freqüência (3) para um valor menor (80 % do máximo) que ainda satisfaz as condições de escalonabilidade. Neste exemplo se pode ver como a troca de mensagem gerenciada pelo mecanismo de tratamento de eventos assíncronos da API de comunicação, associado a um escalonador de tarefas DVS/DFS, pode operar com o mínimo de energia sem perder os *deadlines* das demais tarefas em execução.

4.8 Serviços implementados em Hardware

Assim como na Seção 4.4, onde os objetos em hardware são parte da aplicação, serviços do *middleware* em hardware podem ser usados como alternativa de projeto para atender requisitos, como desempenho e energia.

Alguns trabalhos anteriores têm proposto a implementação de serviços do sistema operacional em hardware, particularmente aqueles ligados ao gerenciamento de tarefas (BURLESON, 1999) (LINDH, 1992) (KUACHAROEN, 2003) (AGRON, 2004) (KOHOUT, 2003). Todos têm o objetivo comum de melhorar o desempenho do sistema, tirando proveito do paralelismo natural de uma implementação em hardware.

Nesta seção será apresentada uma extensão desta idéia, encapsulando as implementações hardware de dois serviços essenciais do sistema operacional, gerenciamento de tarefas e comunicação. Uma vez que eles são encapsulados pelo *middleware*, o projetista pode facilmente explorar as diferentes implementações dos serviços, procurando pela configuração que melhor atenda os requisitos da aplicação, sem comprometer o tempo desenvolvimento.

O papel do *middleware* neste caso é encapsular serviços implementados em hardware em objetos, de modo que fique transparente para a aplicação se o serviço é realizado em software ou por um hardware com função equivalente. Assim, o *middleware* proporciona redução no tempo de desenvolvimento, agregando melhor desempenho e menor consumo de energia, sem perder a previsibilidade requerida por aplicações de tempo-real.

A API RTSJ inclui a implementação hardware do escalonador de tarefas tempo-real. Da mesma forma, os serviços de comunicação, quando implementados em hardware, são encapsulados pela API de comunicação.

4.8.1 Escalonador de tarefas de tempo-real implementado em hardware

A estrutura do escalonador em software consiste em um processo adicional que se encarrega da alocação da CPU para aqueles processos de aplicação que estão prontos para executar, exatamente como ocorre em um RTOS.

O escalonador em hardware é encapsulado por uma classe chamada `HardwareScheduler`, que interage com o hardware e realiza as ações de troca de contexto e despacho. Estas ações têm um baixo custo relativamente à ação de escalonar as tarefas, especialmente quando se trata de algoritmos de escalonamento complexos.

O desenvolvedor pode facilmente optar por uma das versões, pois ambas são compatíveis do ponto de vista da sua interface, embora elas se apliquem a diferentes requisitos de tempo e energia. Detalhes de baixo nível do hardware são escondidos do desenvolvedor da aplicação que escolhe qual versão do escalonador utilizar apenas trocando a classe do objeto.

Ao mover o escalonamento de tarefas do software para o hardware, esta função do sistema operacional não mais compete com as aplicações pelo uso do processador. A função de escalonamento tem sua própria e dedicada unidade de hardware, que é capaz de executar algoritmos de escalonamento mais complexos e fazer um escalonamento realmente não-invasivo, melhorando a previsibilidade temporal das tarefas.

4.8.1.1 Arquitetura do escalonador em hardware

A arquitetura do escalonador em hardware e seus componentes principais: General Register, Scheduler, SyncEvent e AsyncEvent são apresentados na Figura 4.19.

O bloco General Register contém os descritores de todas as tarefas independentemente de seu estado (*running*, *blocked* e *ready*). A qualquer momento, o processador pode incluir ou remover tarefas neste bloco.

O bloco AsyncEvent recebe tarefas que serão disparadas por eventos e monitora eventos externos de hardware cujos instantes de ocorrência não podem ser definidos *a priori*. Estes eventos ou sinais são tipicamente associados a sensores, tais como ocorrência de alarmes. Na ocorrência de um evento associado a uma tarefa, o identificador da tarefa é enviado para o bloco Scheduler.

O bloco SyncEvent é similar ao bloco AsyncEvent, porém monitora tarefas da aplicação sensíveis a eventos do relógio de tempo-real do sistema. A execução deste tipo de tarefa depende do relógio e pode ocorrer periodicamente ou em instantes específicos.

O bloco Scheduler recebe tarefas que estão prontas para executar, enviadas pelos blocos SyncEvent ou AsyncEvent, e as coloca em sua tabela, classificando-as de acordo com a ordem estabelecida pela política de escalonamento. O bloco Scheduler envia tarefas para o processador em duas situações:

1) A tarefa atual concluiu a sua execução. O processador executa uma operação de leitura e recebe a próxima tarefa informada pelo escalonador.

2) A tarefa recentemente inserida na primeira posição da tabela do bloco Scheduler tem prioridade sobre a tarefa atualmente em execução no processador. O escalonador interrompe o processador, provocando a preempção da tarefa que estava em execução.

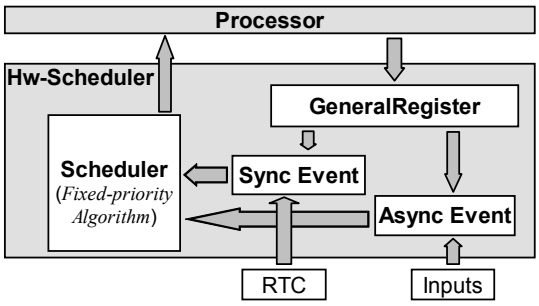

Figura 4.19: Arquitetura do escalonador em Hardware

Uma classe Java (`FixedPriorityHWScheduler`) encapsula a comunicação entre o processador e o escalonador em hardware. Esta classe, ao ser incluída na implementação da RTSJ, permite que o desenvolvedor possa facilmente optar por uma das versões do escalonador da API RTSJ, seja ela hardware ou software, pois todas são compatíveis, do ponto de vista de sua interface.

Mais detalhes da implementação, bem como do objeto que faz o encapsulamento do escalonador em hardware, podem ser encontrados em (SILVA JÚNIOR, 2005a) e (SILVA JÚNIOR, 2005b).

4.8.2 Comunicação (transporte) implementada em hardware

Na estratégia adotada neste trabalho o componente hardware que faz o serviço de comunicação é encapsulado pela classe `HwTransport` e pode ser usado da mesma maneira que a versão implementada em software (classe `Transport`). O processador

interage com o bloco de comunicação da mesma forma que com outros dispositivos de entrada/saída.

Quando uma mensagem deve ser enviada, a classe `HwTransport` lê o objeto `Message` passado como parâmetro e o entrega para o hardware. Da mesma forma, quando está recebendo uma mensagem, a classe preenche com os dados vindo do hardware um objeto `Message` passado pela aplicação. Estas operações são transparentes para o programador da aplicação, uma vez que estas são internas à classe `HwTransport`.

4.8.2.1 Arquitetura da camada de Transporte implementada em hardware

A Figura 4.20 mostra a arquitetura geral do hardware que implementa a camada de transporte. O bloco *Network Interface* (interface de rede) é responsável por entregar os pacotes ao nível físico. De fato, o *hardware communication block* implementa as camadas de Transporte e de Enlace de Dados. Esta opção, adotada para demonstrar o conceito, pode não ser a melhor do ponto de vista da flexibilidade de projeto. Ela foi escolhida por ser a mais agressiva, levando todas as funções de comunicação para o hardware e oferecendo a maior redução no tempo de execução.

O bloco OP_READER recebe e interpreta comandos vindos do processador e despacha comandos e dados para os blocos OUTPUT_MESSAGE_STORAGE ou CONNECTION_MANAGER.

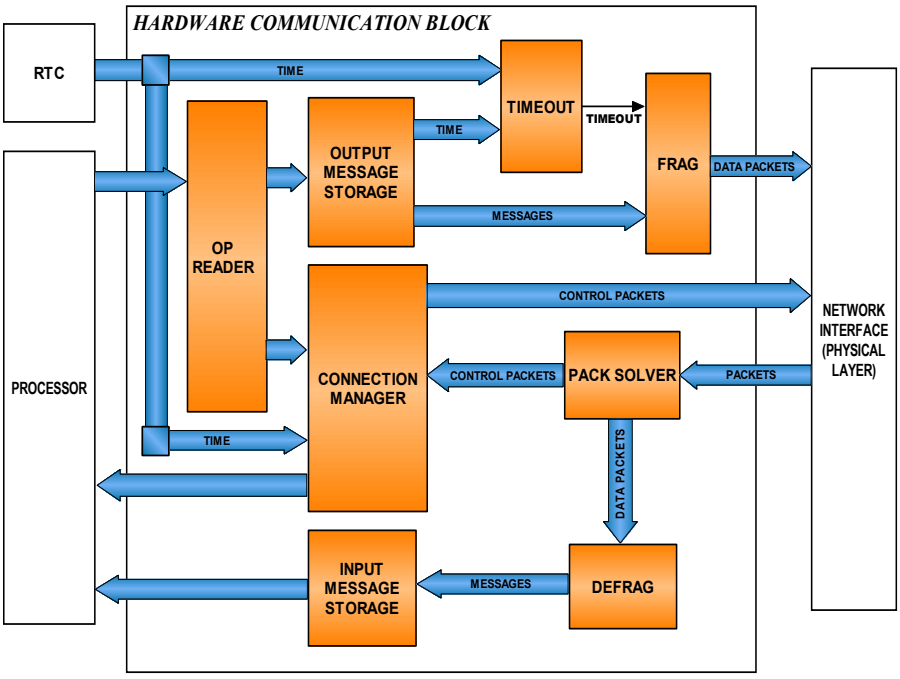

Figura 4.20: Arquitetura do Transporte em Hardware

O bloco OUTPUT_MESSAGE_STORAGE é encarregado do armazenamento das mensagens que devem ser enviadas. Logo que ele recebe uma mensagem completa, inicia a interação com o bloco FRAG, o qual irá preencher o número de pacotes

necessários e enviar para a interface de rede. Depois de enviar cada pacote o FRAG espera por uma confirmação vinda da interface de rede antes de enviar o próximo pacote.

O bloco TIMEOUT monitora as operações do FRAG, acompanhando a evolução do relógio de tempo-real (RTC). Se o prazo estabelecido para envio da mensagem termina, uma exceção é enviada ao processador, permitindo que a aplicação recupere o controle por falha na tentativa de envio de uma mensagem.

O bloco PACK_SOLVER é capaz de identificar o tipo de pacote que chega do nível físico. Se for um pacote de dados, esse é enviado para o bloco DEFRAG; caso contrário, é um pacote de controle e deve ser enviado ao bloco CONNECTION_MANAGER.

O bloco DEFRAG recebe e acumula os pacotes de dados vindos da interface de rede. Quando uma mensagem está pronta, é enviada para o bloco INPUT_MESSAGE_STORAGE, o qual sinaliza para o processador que há uma mensagem pronta. Em seguida, INPUT_MESSAGE_STORAGE entrega a mensagem para o processador em uma seqüência pré-definida de bytes.

O bloco CONNECTION_MANAGER é responsável por abrir e fechar conexões. Ele também interage com o bloco TIMEOUT para garantir a previsibilidade de suas operações.

4.9 Discussão

Uma visão geral do *middleware* e seus requisitos foi apresentada neste capítulo. Cada serviço foi apresentado, seguido de sua descrição funcional. O *middleware* tem na sua base (camada de estrutura) um serviço de comunicação e um mecanismo de gerenciamento *multithread* tempo-real (RTSJ). Acima destes, quatro serviços são descritos por serem considerados essenciais para aplicações em MPSoCs. Todavia, outros serviços poderão ser descritos e implementados sobre a camada de estrutura.

Uma lista completa das funcionalidades do *middleware* é dada abaixo:

- Implementação de objetos da aplicação em hardware
- Serviços implementados em hardware
- Invocação de objeto remoto
- Migração de tarefa
- Alocação de tarefa
- Comunicação por troca de mensagens
- Escalonamento de tarefas de tempo-real com mecanismo de DVS

Das propriedades citadas na Tabela 2.1, apenas a adaptação no software não é incluída no *middleware* aqui proposto. As demais estão contempladas nos serviços acima, para citar: nodos com limitada capacidade computacional, limitada memória de dados e código, paradigma de orientação a objetos, expressão de requisitos tempo-real, mecanismo de controle/limitação de energia, implementações hardware na aplicação e nos serviços, adaptação no hardware.

O porte do *middleware* para outro tipo de plataforma onde a rede não seja intra-chip é viável. Os serviços aqui descritos podem ser reusados e/ou ampliados em função dos requisitos provenientes do novo domínio de aplicação.

Exemplos de utilização dos serviços descritos neste capítulo foram propositalmente omitidos, pois serão descritos no capítulo a seguir.

5 EXEMPLOS DE UTILIZAÇÃO DO *MIDDLEWARE*

Este capítulo apresenta casos de uso do *middleware* que foram implementados e simulados. Duas seções compõem o capítulo: a primeira, mostra exemplos simples de utilização de cada serviço individualmente e a segunda, apresenta exemplos que integram mais de um serviço.

As implementações do *middleware* foram avaliadas utilizando um simulador ciclo-a-ciclo do processador FemtoJava. Para a Seção 5.1.6 foi usado o simulador CACO-PS. Para as demais seções foi empregado o SERPENS, que utiliza um modelo SystemC do processador conectado à NoC SoCIN.

5.1 Casos de uso simples para cada serviço implementado

Esta seção está dividida em uma subseção para cada serviço implementado do *middleware*. Todas as subseções contêm um exemplo de utilização comentado seguido de resultados de simulação, avaliando desempenho, memória e previsibilidade.

5.1.1 Troca de mensagens usando a API de comunicação

Esta seção apresenta um exemplo simples e completo da utilização das funcionalidades disponibilizadas pela API de comunicação. A partir deste exemplo é possível vislumbrar aplicações mais elaboradas, como os serviços de migração de tarefas e de acesso a objetos remotos, mostrados a seguir.

Este exemplo de aplicação consta de dois nodos, um encarregado de enviar uma mensagem e o outro de receber. O código fonte do receptor é mostrado na Figura 5.1. Nas linhas 8-10 são construídos e inicializados os componentes da API de comunicação. Nas linhas 19-21 o receptor fica à espera de um pedido de conexão na porta número 2. Ao receber este pedido ele o aceitará e passará a esperar por uma mensagem através daquela conexão estabelecida (linha 22). Após receber a mensagem, o receptor pede o fechamento da conexão (linha 24). O método `initSystem()` é uma exigência do SASHIMI-FemtoJava e será usado como o ponto de entrada para execução da aplicação (equivalente ao método `main`).

A Figura 5.2 mostra o código fonte do transmissor, que, do outro lado, realiza tentativas de conexão com o endereço lógico 1 na porta de número 2 até obter sucesso

(linhas 26-28). Ao estabelecer a conexão, uma mensagem é enviada com o conteúdo "Hello World" (linhas 29-31). Se o tempo limite, definido na linha 15 (5 milissegundos), for atingido, o método `sendMsg()` retorna `FALSE` e a aplicação invoca um método de tratamento. Nesta aplicação a prioridade das mensagens foi estabelecida na conexão pela passagem do valor PRIORITY (linha 7). Este valor é atribuído a todas as mensagens nesta conexão, podendo ser sobrescrito por um valor passado diretamente na chamada ao `sendMsg()`.

```
 1 import saito.sashimi.*;
 2 import saito.sashimi.realtime.*;
 3 import saito.sashimi.ApiCom.*;
 4
 5 public class Receiver {
 6     private static final int PORT = 2;
 7     private static final int MY_ADDRESS = 1;
 8     public static DataLinkSOCIN dlink = new DataLinkSOCIN();
 9     public static Network net = new Network();
10     public static Transport tp = new Transport(net, dlink, MY_ADDRESS);
11
12     public static int conectionNumber;
13     public static Message myMsg = new Message();
14
15     public static RelativeTime receiverTimeOut = new RelativeTime(0,5,0);
16
17     public static void initSystem() {
18         tp.setupListen(PORT);
19         do {
20              conectionNumber = tp.listen(PORT, receiverTimeOut);
21         }  while (conectionNumber < 0);
22         while (tp.messageReady(conectionNumber) == false) {}
23         tp.receiveMsg(myMsg, conectionNumber);
24         tp.closeConnection(conectionNumber);
25         idleTask();
26     }
27 };
```

Figura 5.1: Código fonte do receptor

```
 1 import saito.sashimi.*;
 2 import saito.sashimi.realtime.*;
 3 import saito.sashimi.ApiCom.*;
 4
 5 public class Sender {
 6     private static final int MY_ADDRESS = 0;
 7     private static final int PORT = 2;
 8     private static final int ADDRESS = 1;
 9     private static final int PRIORITY = 7;
10
11     public static DataLinkSOCIN dlink = new DataLinkSOCIN();
12     public static Network net = new Network();
13     public static Transport tp = new Transport(net, dlink, MY_ADDRESS);
14
15     private static RelativeTime sendTimeOut = new RelativeTime(0,5,0);
16
17     public static int conectionNumber;
18     public static Message myMsg = new Message();
19     public static byte[] hello = {'H','e','l','l','o',' ','W','o','r','l','d'};
20
21     public static void initSystem() {
22        myMsg.setNrBytes(0);
23        for (int i=0; i<hello.length; i++)  {
24           myMsg.addByte(hello[i]);
25        }
26        do {
27           conectionNumber = tp.openConnection(ADDRESS,PORT,sendTimeOut,PRIORITY);
```

```
28          } while (conectionNumber < 0);
29           if ( tp.sendMsg(conectionNumber, myMsg, sendTimeOut) == false) {
30             error();
31          }
32          idleTask();
33      }
34  };
```

Figura 5.2: Código fonte do transmissor

5.1.1.1 Uso de eventos na troca de mensagens

Para experimentar o uso do conceito de eventos na APICOM foi elaborada e implementada uma aplicação onde mensagens são enviadas de um nodo da rede para outro. O transmissor e o receptor estão localizados em processadores diferentes, ligados pela rede. O código fonte do transmissor é o mesmo do exemplo anterior, visto na Figura 5.2.

O receptor usa eventos tanto para reconhecer o pedido de conexão quanto para receber as suas mensagens. A Figura 5.3 mostra a classe principal do receptor, que faz a inicialização da aplicação. A complexidade do código aumenta em relação ao exemplo anterior, uma vez que os parâmetros de multitarefa devem ser fornecidos. Com o uso de eventos uma *thread* vai cuidar da comunicação para a aplicação. Na linha 6 o escalonador é declarado e nas linhas 7-14 são declarados o mecanismo de tratamento de eventos e seus parâmetros. O tratador de eventos é definido na linha 16 e o objeto evento, na linha 17. Quando a aplicação se inicia, o método `initSystem()` é invocado. O escalonador é definido na linha 30 e ativado na 34. A linha 31 declara o mecanismo que vai gerenciar os eventos assíncronos. O método `addHandler()`, invocado na linha 32, informa ao evento quem é o seu tratador e na seguinte o tratador de eventos é inicializado. Mais adiante o método `init()` será explicado.

```
 1  import saito.sashimi.*;
 2  import saito.sashimi.realtime.*;
 3  import saito.sashimi.ApiCom.*;
 4
 5  public class Receiver {
 6      public static EDFScheduler mySched = new EDFScheduler();
 7      private static PeriodicParameters asmReleaseParam =
 8                          new PeriodicParameters(TimeObjects._2_ms, // start
 9                                                 null, // end
10                                                 TimeObjects._1_ms, // period
11                                                 TimeObjects._500_us, // cost
12                                                 TimeObjects._500_us); // deadline
13      public static AsyncEventsMechanism asyncEventMechanism =
14                          new AsyncEventsMechanism(null, asmReleaseParam);
15
16      public static Consumer consumidor = new Consumer();
17      public static MsgEvent msgEvent = new MsgEvent();
18
19      public static void initSystem(){
20          Scheduler.setDefaultScheduler(mySched);
21          mySched.setPoolingServer(asyncEventMechanism);
22          msgEvent.addHandler(consumidor);
23          consumidor.init(msgEvent);
24          mySched.setupTimer();
25          idleTask();
26      }
27  };
```

Figura 5.3: Código fonte do receptor com eventos

A Figura 5.4 mostra o código da classe que implementa o tratador de eventos, chamada `Consumer`, a qual é filha da classe `AsyncEventHandler`, da API RTSJ. Esta classe encapsula os mecanismos de comunicação através da APICOM. O método `init()` é fornecido para configuração dos eventos. Uma instância de `AsyncEvent` deve ser passada, a qual será usada pela APICOM para informar quando mensagens de dado ou de requisição de conexão chegarem. Nas linhas 27 e 28 o objeto evento é passado à APICOM junto com a porta que deve ser associada ao evento. O método `handleAsyncEvent()`, que é ativado quando o evento ocorre, classifica o evento (dado ou requisição de conexão) verificando se o número da conexão é válido. Nesta aplicação o receptor espera por um pedido de conexão e em seguida passa a esperar por mensagens, usando o mesmo evento tanto para requisição de conexão quanto para recebimento de mensagens.

Este exemplo serve ainda para mostrar como a APICOM usa a idéia de eventos internamente para tratar a chegada de pacotes, no nível de enlace de dados. O método `setEventStatus()` permite ativar um serviço de eventos no `DataLink`, de modo que a chegada de pacotes vai disparar um tratador específico, como foi mencionado na Seção 4.2.1.

```
 1 import saito.sashimi.*;
 2 import saito.sashimi.realtime.*;
 3 import saito.sashimi.ApiCom.*;
 4
 5 public class Consumer extends AsyncEventHandler {
 6     private static final int PORT = 3;
 7     private static final int MY_ADDRESS = 1;
 8     public static DataLinkSOCIN dlink = new DataLinkSOCIN();
 9     public static Network net = new Network();
10     public static Transport tp = new Transport(net, dlink, MY_ADDRESS);
11
12     public static int conectionNumber = -1;
13     public static Message userMsg = new Message();
14     public static RelativeTime serverTimeOut = new RelativeTime(0,5,0);
15
16     public void handleAsyncEvent() {
17         if (conectionNumber < 0) {
18                 conectionNumber = tp.listen(PORT, serverTimeOut);
19         } else
20         if ( (conectionNumber >= 0) && (tp.messageReady(conectionNumber)) ) {
21                 tp.receiveMsg(userMsg, conectionNumber);
22         }
23     }
24
25     public void init(AsyncEvent ev) {
26         tp.setEventStatus(true); // datalink com eventos
27         tp.setMsgRdyEvent(ev, PORT);
28         tp.setResquestEvent(ev, PORT);
29         tp.setupListen(PORT); // prepara para escutar a porta
30     }
31 }
```

Figura 5.4: Código fonte para o tratador de eventos do receptor

5.1.1.2 Resultados de simulação

Para demonstração do serviço de comunicação foi usado um produtor-consumidor simples onde um processador envia mensagens para outro processador. Para conexão entre os processadores foi usada a NoC SoCIN e o simulador é o SERPENS.

O tempo gasto enviando e recebendo 20 mensagens com comprimentos variando de 01 a 20 bytes é mostrado na Figura 5.5, onde o eixo x indica o comprimento das mensagens. A freqüência de operação dos dois processadores é de 100MHz. Na figura se observa um crescimento gradual da latência, com um pequeno salto entre 07 e 08 bytes e entre 14 e 17 bytes, exatamente quando o número de pacotes usados pela mensagem muda. Para este exemplo o tamanho dos pacotes foi definido como 07 bytes. Portanto, a partir de 08 bytes a API tem que lidar com dois pacotes. O custo adicional está relacionado com a fragmentação/desfragmentação. Também se nota que a latência no destino é mais sensível à variação do tamanho da mensagem. Isto ocorre porque o recebimento envolve copiar um objeto mensagem do transporte para a aplicação, o que depende do tamanho da mensagem. No envio, somente uma referência do objeto mensagem é entregue à API.

Vale observar que a soma dos tempos dados no gráfico da Figura 5.5 para transmissor e receptor não coincide com a latência total desde o instante em que a transmissão se inicia até o final da recepção. Existe uma superposição entre estes dois tempos de modo que a latência total observada por uma base tempo comum é mais favorável, como mostrado na Figura 5.6.

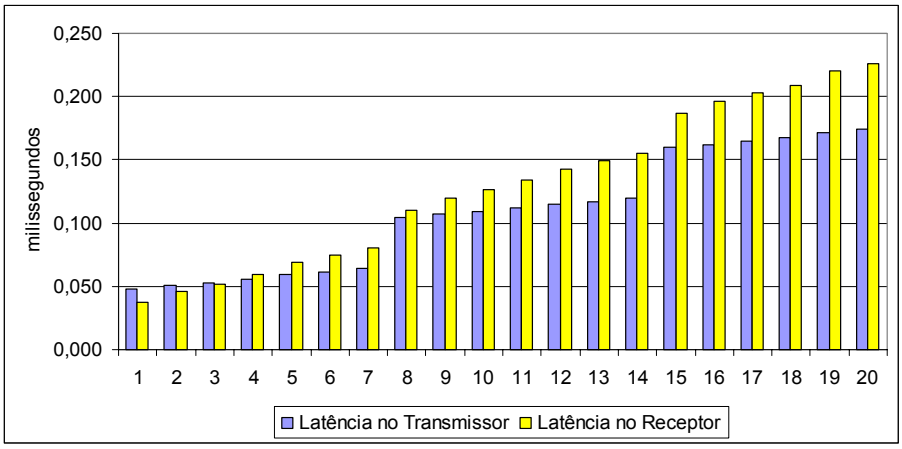

Figura 5.5: Latências no produtor e no consumidor

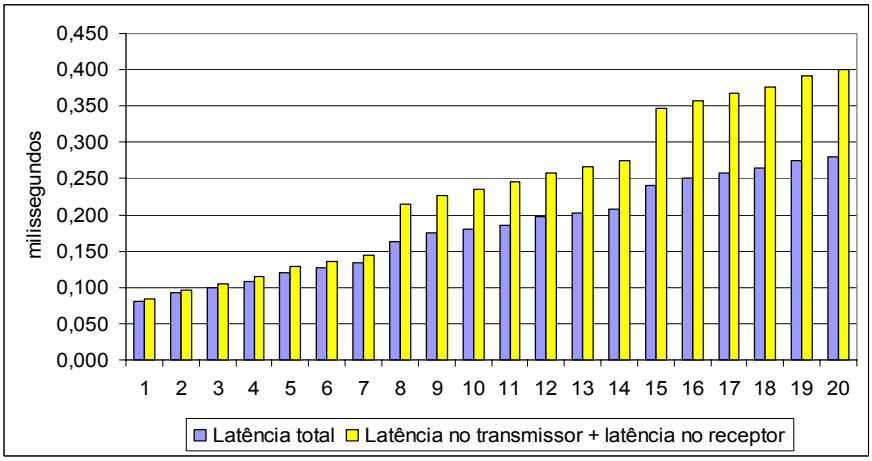

Figura 5.6: Latência total versus custos nos dois lados da comunicação

O desenvolvedor pode alterar parâmetros como tamanho da mensagem, usada pelo transporte, e tamanho do pacote, usado pelo enlace de dados. Esta possibilidade abre espaço para diferentes combinações de uso de memória e desempenho. Para mostrar isso, um outro experimento similar foi montado, desta vez configurando a APICOM para usar pacotes de 49 bytes e mensagens de 500 bytes. São avaliados os impactos no desempenho e na memória.

Os resultados de desempenho são semelhantes ao experimento inicial, mostrando uma variação mais brusca nas latências quando a quantidade de pacotes muda. Usar pacotes maiores contribui para minimizar estes saltos em mensagens menores, já que o primeiro salto por aumento no número de pacotes, neste caso, só vai ocorrer para uma mensagem de 50 bytes. Além disso, como a mensagem é bem maior, esta variação fica quase imperceptível para mensagens próximas de 500 bytes, como se vê na Figura 5.7. A partir de 491 bytes a mensagem passa de 10 para 11 pacotes, mas o salto é proporcionalmente muito pequeno.

A rede SoCIN é capaz de enviar pacotes de qualquer tamanho, ou seja, não existem tamanhos pré-definidos. Mais ainda, mesmo que o pacote tenha um tamanho definido pela API, é possível que no nível físico seja usado um pacote menor se a mensagem for pequena o suficiente. A classe que implementa o nível de enlace (`DataLinkSOCIN`) explora esta propriedade. Assim, o desperdício pelo aumento do tamanho do pacote é mínimo em termos de desempenho.

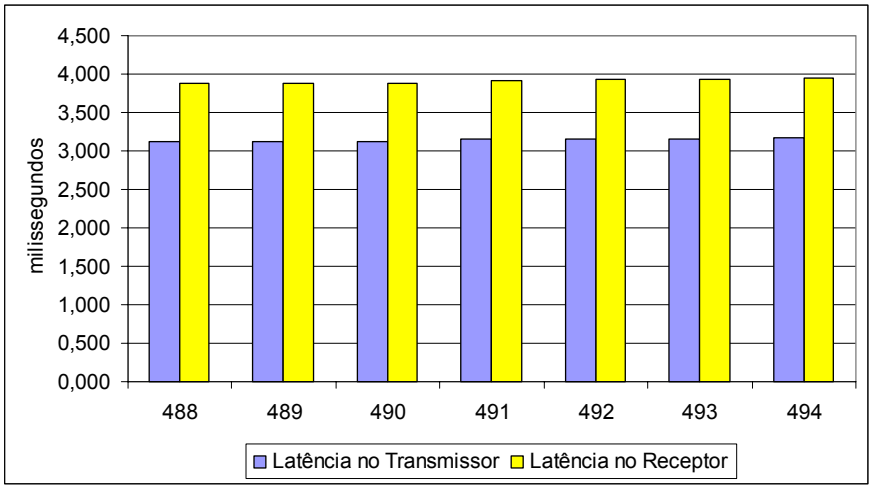

Figura 5.7: Latências no produtor e no consumidor (mensagem grande)

O volume de memória utilizado pela API de comunicação foi medido e está indicado na Tabela 5.1. Estes dados são relativos às classes da API, excluindo-se todos os custos da aplicação. Os resultados para as duas versões testadas estão indicados em linhas separadas, conforme indicação na primeira coluna. Comparando os custos de memória de código (ROM) observa-se que há uma diferença entre o valor para o transmissor e para o receptor. Esta diferença decorre da otimização feita pelo SASHIMI, retirando o método `Transport.sendMsg()` do receptor e o `Transport.receiveMsg()` do transmissor, já que estes métodos não são alcançados (acessados) pela aplicação usada

para esta avaliação. O gasto de memória de dados (RAM) é igual para os dois processadores, já que todos os atributos são alcançados pelos métodos carregados pelo SASHIMI. Na comparação entre os resultados para os diferentes tamanhos de pacotes e mensagens, a memória de dados aumenta e a de código não se altera. Embora o tamanho da mensagem tenha crescido 25 vezes e o dos pacotes 07 vezes, a área de dados cresce 07 vezes. Cabe destacar que existem vários objetos mensagem (`Message`) e pacote (`Pack`) dentro da API e que todos eles são construídos estaticamente e estão incluídos nos números da Tabela 5.1.

Tabela 5.1: Ocupação de memória pela APICOM

	Nodo Transmissor		**Nodo Receptor**	
	ROM	RAM	ROM	RAM
Pacote 7 bytes – Mensagem 20 bytes	4223 Bytes	913 Bytes	4084 Bytes	913 Bytes
Pacote 49 bytes – Mensagem 500 bytes	4223 Bytes	6345 Bytes	4084 Bytes	6345 Bytes

A Tabela 5.2 foi construída para oferecer uma referência de uso de memória quando aplicações são construídas sobre a plataforma SASHIMI-FemtoJava. Foram coletados os dados de memória de código (ROM) e de atributos (RAM). As informações procuram cobrir aplicações bem diferentes. A mais simples delas é o ordenador HeapSort para 10 valores. O Crane é um controlador de um guindaste, uma aplicação tipo *control-flow*. O IMDCT (*Inverse Modified Discrete Cosine Transform*) é uma transformação linear particularmente útil em aplicações de tratamento de sinais e foi escolhida por ser *data-flow*. Finalmente, o Mp3player é uma aplicação completa, que inclui o IMDCT e outras funções. Comparando o uso de memória da API de comunicação com as aplicações da Tabela 5.2 conclui-se que o seu custo é aceitável para aplicações completas, como é o caso do Mp3Player.

Tabela 5.2: Ocupação de memória para aplicações *benchmark*

Aplicação	**ROM**	**RAM**
HeapSort 10	352 Bytes	43 Bytes
Crane	4842 Bytes	423 Bytes
IMDCT	394 Bytes	1796 Bytes
Mp3Player	48548 Bytes	63702 Bytes

Para avaliação do uso de eventos na API de comunicação, o exemplo da Seção 5.1.1.1 foi implementado do lado receptor, usando como transmissor o mesmo produtor dos exemplos anteriores.

O custo em memória pelo uso de eventos é mostrado na Tabela 5.3. Nestes números são consideradas classes da API-RTSJ e da API de comunicação. Na prática, o *overhead* tende a ser menor do que o mostrado na tabela, já que a aplicação pode estar usando o serviço de eventos da API-RTSJ e este custo não seria adicionado. O custo do

escalonador de tarefas não foi considerado, admitindo-se que a aplicação seria *multithread*. Também há que se considerar que o custo com os objetos eventos (`AsyncEvent`) e seus tratadores (`AsyncEventHandler`) depende da utilização dada pela aplicação. Nestas medidas considerou-se um uso mínimo que seria um par para a aplicação e outro para o nível de enlace de dados da APICOM. A avaliação foi feita no receptor das mensagens, onde o conceito de eventos foi introduzido.

Tabela 5.3: Ocupação adicional de memória com uso de eventos

API-RTSJ		**APICOM**	
ROM	RAM	ROM	RAM
1136 Bytes	44 Bytes	33 Bytes	8 Bytes

A Figura 5.8 mostra o tempo usado para cada ação em um ciclo onde uma mensagem (evento) chega. Para o experimento o período do `AsyncEventMechanism` foi definido em 1ms e a mensagem recebida tem 08 bytes. O tratador (`MessageHandler`), executado dentro do `AsyncEventMechanism`, tem custo proporcional ao tamanho da mensagem recebida, neste caso 8% do tempo total. A figura destaca o *overhead* do `AsyncEventMechanism` (4%), separado do tratador. É importante notar o custo relativo do `AsyncEventMechanism` em relação ao do escalonador (21%), que neste caso é o mais simples possível – prioridade fixa. Pode-se dizer que o custo adicionado pelo mecanismo de tratamento de eventos é de 20% do custo do escalonador de tarefas.

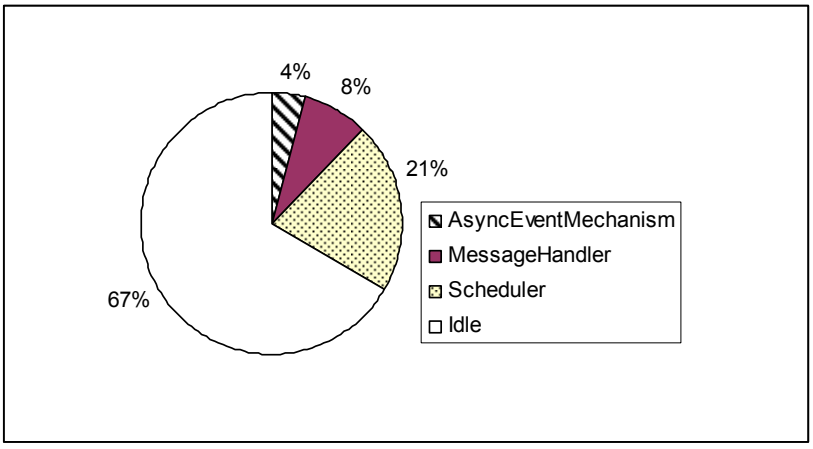

Figura 5.8: Ocupação da CPU em um ciclo

Se o recurso de eventos não fosse utilizado, o custo do `AsyncEventMechanism` não existiria, mas o tratamento da mensagem teria que existir. O prejuízo decorrente de não usar eventos é que uma tarefa esperando por uma mensagem poderia reduzir o tempo *idle* (67%), executando um laço de espera testando se a APICOM teria mensagens prontas. Este tempo *idle* pode ser explorado por um escalonador DVS/DFS para fazer o processador operar a uma freqüência menor que a máxima.

A vantagem do uso do mecanismo de eventos RTSJ está em atribuir à comunicação uma característica periódica, confinada em uma tarefa com custo de execução limitado. Assim, esta implementação evita que estes eventos assíncronos perturbem as tarefas periódicas.

5.1.2 Invocação de método remoto

Em um dos tutoriais da SUN para RMI (SUN, 1996) é construído um exemplo simples de uma calculadora que pode ser acessada por um cliente remoto. Nesta seção é usado o mesmo exemplo para explicar o uso do serviço de invocação de método remoto desenvolvido para o *middleware*.

Para este exemplo três nodos são utilizados. Em um deles é executado o serviço de nomes (`Registry`), encarregado de receber a referência do(s) objeto(s) remoto(s), localizado(s) no(s) servidor(es), e informar posteriormente ao(s) cliente(s). Os dois outros nodos executam o cliente e o servidor.

A Figura 5.9 apresenta o código do objeto a ser disponibilizado remotamente. No exemplo foram implementadas as quatro operações aritméticas básicas. O objeto remoto herda de `Remote_object` e implementa uma `interface`, que contém os métodos que serão vistos pelo cliente.

```
 1 import saito.sashimi.*;
 2 import saito.math.*;
 3 import saito.sashimi.realtime.*;
 4 import saito.sashimi.rmiFJ.*;
 5
 6 public class CalculatorImpl extends Remote_object implements CalculatorInterf  {
 7
 8     public CalculatorImpl() {
 9     }
10
11     public CalculatorImpl(RealTimeParameters t, PeriodicParameters p) {
12         super(t, p);
13     }
14
15     public int add(int a, int b) {
16         return a + b;
17     }
18
19     public int sub(int a, int b) {
20         return a - b;
21     }
22
23     public int mul(int a, int b) {
24         return IntMath.imul(a, b);
25     }
26
27     public int div(int a, int b) {
28         return IntMath.idiv(a, b);
29     }
30 }
```

Figura 5.9: Código do objeto a ser acessado remotamente

A classe `Application`, vista na Figura 5.10, é encarregada de lançar o lado remoto da aplicação, aqui chamado de servidor. As linhas 7, 15 e 19 são necessidades de

qualquer aplicação multitarefa. O objeto servidor é criado nas linhas 9-10 e a sua publicação (junto ao `Registry`) ocorre na linha 17.

A Figura 5.11 mostra um cliente que utiliza o serviço de invocação de método remoto. O cliente obtém uma referência (`calc`) para o objeto remoto quando o `lookup` é chamado (linha 17), e ela passa a apontar para o Stub. Daí em diante o cliente chama os métodos do objeto remoto como se ele estivesse local.

O *cast* na linha 17 permite que o retorno do `lookup`, que é tipo `Object`, tome a forma da classe remota da aplicação. O modelo apresentado na seção 4.3 prevê o uso de interface para este *cast*. Como a versão utilizada do SASHIMI-FemtoJava não trata interfaces Java, a interface (no lado cliente) foi transformada em uma classe.

```
 1 import saito.sashimi.*;
 2 import saito.sashimi.realtime.*;
 3 import saito.sashimi.rmiFJ.*;
 4
 5 public class Application  {
 6     // ......
 7     public static EDFScheduler mySched = new EDFScheduler();
 8
 9     public static CalculatorImpl calc =
10                                  new CalculatorImpl(_rtParam, threadParameter);
11     // SkelFactory deve ser gerada automaticamente
12     public static SkelFactory skelFactory = new SkelFactory(calc);
13
14     public static void initSystem() {
15         Scheduler.setDefaultScheduler(mySched);
16
17         calc.bind(CALCULADORA); // publica o objeto
18
19         mySched.setupTimer();
20
21         idleTask();
22     }
23 }
```

Figura 5.10: Código que publica objeto para acesso remoto

```
 1 import saito.sashimi.*;
 2 import saito.sashimi.realtime.*;
 3 import saito.sashimi.rmiFJ.*;
 4
 5 public class Client  {
 6     // ......
 7     public static ClientNamingSOCIN naming = new ClientNamingSOCIN(_realtimeParam);
 8
 9     public static Calculator calc; // classe q substitui a interface
10     public static final int CALCULATOR = 9; // código_nome do objeto remoto
11
12     public static int result_add;
13     public static int result_sub;
14     // .....
15
16     public static void initSystem() {
17         calc = (Calculator)naming.lookup(CALCULATOR);
18
19         result_add = calc.add(10,20);
20         result_sub = calc.sub(11,2);
21     // ..........
22     }
23 }
```

Figura 5.11: Cliente que acessa objeto remoto

5.1.2.1 Resultados de simulação

O exemplo da seção anterior foi simulado no SERPENS usando três processadores (cliente, servidor e registro de nomes) operando a 100MHz conectados pela NoC SoCIN. Para iniciar a execução da aplicação o registro de nomes deve estar ativo. O servidor é ativado para registrar o objeto que poderá ser acessado remotamente, e em seguida o cliente pode buscar a referência do objeto remoto junto ao registrador de nomes. Neste exemplo, foram usados temporizadores para que cada etapa do processo fosse ativada conforme a seqüência descrita acima.

A Tabela 5.4 traz a ocupação em memória tanto no lado cliente quanto no lado servidor. A primeira linha mostra exclusivamente os gastos do serviço, enquanto a segunda linha mostra os custos com Stub, no lado cliente, e com Skeleton, no lado servidor.

Tabela 5.4: Ocupação de memória pelo acesso a objeto remoto

	Nodo Cliente		**Nodo Servidor**	
	ROM	RAM	ROM	RAM
Custo na API	637 Bytes	47 Bytes	2076 Bytes	112 Bytes
Custo adicional por método remoto	155 Bytes	0 Bytes	63 Bytes	6 Bytes

O custo do Stub e do Skeleton é dependente do número de métodos remotos e da quantidade de seus parâmetros. Por isso, foi incluído na tabela o custo para cada método de dois parâmetros de entrada.

Aos custos da Tabela 5.4 devem ser acrescentados os custos da API de comunicação, uma vez o serviço de objeto remoto funciona sobre o serviço de comunicação.

Tabela 5.5: Latências no processo de invocação de método remoto

Latências (µs)	
t_{Stub} – Tempo para o `Stub` empacotar os dados de chamada do método e enviar	121,0
t_{RS} – Tempo gasto no receptor para processar a mensagem enviada pelo `Stub`	64,0
t_{WH} – Tempo de espera para que a *thread* `ConnectionHandler` seja ativada	2559,0
t_{Sched} – Tempo de execução do escalonador de tarefas (nodo servidor) – pior caso	59,0
$t_{Skeleton}$ – Tempo de execução do `ConnectionHandler` e do `Skeleton`	30,7
t_{Rmote} – Tempo de execução do método remoto	0,4
t_{Rply} – Tempo gasto pelo nodo servidor processando e enviando o retorno do método	113,0
t_{RC} – Tempo gasto pelo nodo cliente processando a mensagem de retorno	74,0

As latências dadas na Equação (4.1), Seção 4.3, foram medidas para este exemplo e são mostradas na Tabela 5.5. O período da *thread* tratadora de conexões (`ConnectionHandler`) no servidor foi definido em 3ms, sendo este o valor máximo

esperado para o t_{WH}. Nota-se o quanto este valor é elevado em relação aos demais custos. Este é o preço pago pelo determinismo no comportamento do sistema. Os tempos $t_{Skeleton}$, t_{Rmote} e t_{Rply} ocorrem dentro da *thread* `ConnectionHandler` e vão sempre respeitar o escalonamento de tarefas no servidor. Do ponto de vista do cliente, a latência para resposta da invocação remota fica previamente conhecida também. Neste exemplo o t_{Rmote} (custo do método remoto executado no servidor) foi muito pequeno, por ser uma operação simples de soma.

5.1.3 Objeto implementado em hardware

Como exemplo de utilização de um componente hardware foi implementado um filtro FIR (*Finite Impulse Response*). Duas *threads* são lançadas, uma implementando o filtro em software e a outra em hardware. As *threads* executam periodicamente para processar um novo valor de entrada, aplicando este novo valor na linha de retardo do filtro, processando-o e saindo para aguardar o próximo período. Neste caso o filtro FIR implementado possui profundidade 10, ou seja, o número de taps do filtro é 10.

A classe principal, mostrada na Figura 5.12, é encarregada de dar partida no sistema. Nesta classe não há diferença no tratamento dado a uma *thread* em hardware ou em software. Nas linhas 5 e 6, as *threads* software e hardware são construídas da mesma maneira, enquanto que nas linhas 14-17 elas são adicionadas ao escalonador e iniciadas.

```
 1 import saito.sashimi.*;
 2 import saito.sashimi.realtime.*;
 3 public class Taskset  {
 4   public static PriorityScheduler sched = new PriorityScheduler();
 5   public static Taskc t1 = new Taskc(); // sw
 6   public static FIR10hw t2 = new FIR10hw(); // hw
 7 // .....
 8
 9   public static void initSystem()  {
10       // initialize the attributes for the threads
11       // .....
12
13       Scheduler.setDefaultScheduler(sched);
14       t1.addToFeasibility();
15       t1.start();
16       t2.addToFeasibility();
17       t2.start();
18       sched.setupTimer();
19       idleTask();
20   }
21 }
```

Figura 5.12: Classe principal do filtro FIR

Uma descrição software do filtro FIR é mostrada na Figura 5.13. Trata-se de uma *thread* periódica que faz passar um valor de entrada através do filtro a cada execução. O método `mainTask()` (linhas 21-43) é o corpo da *thread*. Sendo uma *thread* periódica, o laço do método principal termina com uma chamada ao `waitForNextPeriod()` (linha 41), quando a *thread* renuncia ao processador até o seu próximo período de ativação. Para fins de demonstração o número de execuções da *thread* foi definido em três vezes a profundidade do filtro.

```
 1 import saito.sashimi.*;
 2 import saito.sashimi.realtime.*;
 3
 4 public class Taskc extends RealtimeThread {
 5     // Application Attributes
 6     public final int NTAPS = 10;
 7     public int[] coefs;
 8     public int[] input;
 9     public int[] output;
10     public static Arguments arg = new Arguments();
11     // software version specific attributes
12     private int[] z = new int[NTAPS]; // delay line
13     private int accum;
14     private int dataIn;
15     // ReleaseParameters definition
16     // Defines period, start-time, deadline and other RTSJ params.
17     // .....
18     // SchedulingParameters definition
19     // .....
20
21     public void mainTask() {    // equivalent to run()
22       coefs = arg.coefs;
23       input = arg.input;
24       output = arg.output;
25       for(int ii = 0; ii < 3*NTAPS; ii++)     {
26         dataIn = input[ii];
27         /* store current input at the beginning of the delay line */
28         z[state] = dataIn;
29         if (++state >= NTAPS) {
30             state = 0;
31         }
32         /* calc FIR and shift data */
33         accum = 0;
34         for (int jj = NTAPS - 1; jj >= 0; jj--)  {
35             accum += coefs[jj] * z[state];
36             if (++state >= NTAPS) {
37                 state = 0;
38             }
39         }
40         output[ii] = accum;
41         this.waitForNextPeriod();
42       }
43     }
44     public void exceptionTask() {}
45     protected void initializeStack() {}
46 }
```

Figura 5.13: Filtro FIR implementado em software

A Figura 5.14 apresenta o código que encapsula a *thread* hardware. A classe estende a classe `HwRealtimeThread`, que foi incluída na API RTSJ. A classe `FIR10hw`, que é a parte Java da *thread* tempo-real, é descrita de forma idêntica à sua equivalente em software. Ela deve construir objetos `SchedulingParameters` e `ReleaseParameters` (linhas 13-17), que são passados no construtor (linha 20). A primeira diferença visível entre as classes software e hardware do FIR aparece no construtor, já que a `HwRealtimeThread` contém um número para definir qual componente de hardware ele encapsula. Esta constante é chamada de `myHWTI` no código (linha 6). Este número será usado pelo sistema para selecionar o componente de hardware correto. O uso deste parâmetro permite que mais de uma *thread* em hardware seja executada em uma mesma aplicação. A classe da Figura 5.14 também não contém o método `mainTask()` porque ele é implementado na classe mãe e é `final`. Nas linhas 28 e 29 estão os métodos que podem ser invocados pela *thread* hardware, que não foram usados nesta aplicação.

```
 1 import saito.sashimi.*;
 2 import saito.sashimi.realtime.*;
 3
 4 public class FIR10hw extends HwRealtimeThread  {
 5     // Tell the number of this hardware thread
 6     private static final int myHWTI = 0;
 7     // Application Attributes
 8     public final int NTAPS = 10;
 9     public int[] coefs;
10     public int[] input;
11     public int[] output;
12     public static Arguments arg = new Arguments();
13     // ReleaseParameters definition
14     // Defines period, start-time, deadline and other RTSJ params.
15     // .....
16     // SchedulingParameters definition
17     // .....
18
19     public FIR10hw()  {
20       super(myHWTI, schedParam, periodicParam);
21       coefs = arg.coefs;
22       input = arg.input;
23       output = arg.output;
24     }
25
26     public void exceptionTask() {}
27     protected void initializeStack() {}
28     protected void method01() {}
29     protected void method02() {}
30 }
```

Figura 5.14: Classe que encapsula o FIR em hardware

5.1.3.1 Resultados de simulação

Para verificação experimental o simulador SERPENS foi usado, colocando toda a aplicação em um único processador. Um exemplo envolvendo *thread* hardware e multiprocessador será mostrado na seção seguinte.

Dois experimentos foram realizados: o primeiro demonstra a redução no tempo de execução propiciado pela implementação em hardware; o segundo experimento mostra a regularidade (previsibilidade) do sistema quando o número de tarefas cresce.

Para o primeiro experimento, filtros FIR com 10, 30 e 100 taps foram implementados em Java, para a versão software. A versão hardware foi descrita em SystemC. As *threads* foram configuradas para executar a cada 3ms, processando a cada período um novo valor de entrada. Este período foi usado para garantir a escalonabilidade de um conjunto de tarefas que inclui FIR com 100 taps, versões hardware e software, e uma task sintética, com o sistema operando a 50 MHz.

A Tabela 5.6 mostra os valores de tempo para execução em software e em hardware. Um escalonador de prioridade fixa foi usado e o processador tem freqüência de 50MHz. Como era de se esperar, os resultados de desempenho mostram ganhos na implementação em hardware. O período de execução das *threads*, definido como 3ms por construção, foi estritamente respeitado.

É importante comentar que o tempo gasto na execução do opcode CALL é de 12,5µs (625 ciclos de relógio) e representa o principal custo na execução da *thread* hardware (para um filtro de 10 taps este valor é quase 100% do custo total). Uma vez que o método `waitForNextPeriod()` é executado cada vez que um valor de entrada é

processado, uma *thread* em hardware não será vantajosa se a implementação em software não tiver um custo bastante superior a 625 ciclos de relógio.

Tabela 5.6: Medidas de tempo para filtro FIR em hardware e em software

	Tempo de execução (µs)	
Nº de taps	**Filtro FIR (software)**	**Filtro FIR (hardware)**
10	46,7	16,1
30	125,1	22,4
100	399,6	39,8

O segundo experimento foi feito colocando a implementação hardware do filtro com 100 taps e outras três *threads* periódicas (T1, T2 e T3) implementadas em software para serem executadas concorrentemente. A freqüência do processador foi mantida em 50MHz e o escalonador de tarefas usado foi um EDF. A Tabela 5.7 mostra o tempo de execução, o *deadline* e o período para as tarefas utilizadas no experimento. Os valores de tempo de execução foram extraídos dos próprios resultados de simulação, enquanto que o período e o *deadline* são definidos no código através dos `ReleaseParamenters`. Se uma *thread* não executa antes do seu tempo de ativação acrescido do seu valor de *deadline* o escalonador de tarefas irá executar o método `exceptionTask()`.

Tabela 5.7: Caracterização das *threads* do experimento 02 com filtro FIR

	T1	**T2**	**T3**	**FIR (hw)**
Período (ms)	2,0	2,0	2,0	2,0
***Deadline* (ms)**	1,0	1,0	1,0	1,0
Tempo médio de execução (µs)	135,0	135,0	135,0	39,8

A Tabela 5.8 mostra os resultados extraídos da simulação do segundo experimento. A primeira coluna traz o número de tarefas em execução, incluindo o filtro em hardware. A coluna seguinte (ocupação da CPU) indica o tempo total no qual a CPU está executando *threads* ou o escalonador em um dado período. Se nenhuma tarefa está pronta para ser executada, a CPU está em uma condição chamada *idle*. Assim, a ocupação da CPU é igual a (*Período – idle*)/*Período*. O tempo de ativação é o instante em que uma *thread* periódica é ativada. Foram coletadas 10 amostras, ou seja, as 10 primeiras ativações de cada tarefa. O *jitter*, neste caso, foi obtido usando o atraso na ativação da *thread* em cada período. O valor do *jitter* reflete o máximo afastamento em relação ao valor médio. A coluna seguinte mostra o mesmo *jitter*, desta vez percentual em relação ao período de ativação da tarefa.

Tabela 5.8: *Jitter* no tempo de ativação das *threads*

Número de *Threads*	Ocupação da CPU	*Jitter* no tempo de ativação (µs)				*Jitter* no tempo de ativação (%)			
		FIR (hw)	T1	T2	T3	FIR (hw)	T1	T2	T3
02	48%	0,08	1,17	-	-	0,004			
03	71%	0,27	0,27	0,27	-	0,014	0,014	0,014	
04	96%	0,49	0,49	0,62	0,50	0,025	0,025	0,031	0,025

Os resultados mostram um sistema previsível e estável. À medida que o número de tarefas cresce, o tempo de execução do escalonador também cresce. Entretanto, o tempo de ativação não sofre um *jitter* significativo.

5.1.4 Migração de tarefas

Para demonstrar o uso do serviço de migração de tarefas é apresentado um exemplo onde três delas são executadas em um processador e uma é executada em outro. Após certo tempo, uma das tarefas do primeiro processador migra para o segundo processador.

Na implementação do serviço de migração foi usado o conceito de 'replicação de *thread*', onde uma cópia da(s) *thread*(s) está presente na memória local de cada processador. Apenas um processador de cada vez pode executar a *thread*. Esta limitação decorre do fato do SASHIMI-FemtoJava não possuir Unidade de Gerenciamento de Memória (MMU) ou outro mecanismo que permita tradução de endereços para realocação do código da tarefa no processador destino. A replicação é necessária apenas para as *threads* que poderão ser migradas, mas, para os testes, todo o código foi replicado. Assim, o código a ser usado nos dois processadores é igual, mas tem comportamento diferente.

Embora esta técnica leve a um desperdício de memória para as *threads* que poderão migrar, ela tem a vantagem de ser mais rápida por dispensar o tempo de alocação de memória demandado no processo de construção do(s) objeto(s) no destino.

A Figura 5.15 mostra o código Java utilizado para os dois lados do processo. A diferença fica na linha 21, onde será determinado o comportamento de enviar a *thread* ou de estar pronto para receber a *thread*. O atributo `_sender` será `TRUE` ou `FALSE`, dependendo de em qual processador o código for lançado. Como o serviço de migração usa o tratamento de eventos da RTSJ, a classe principal da aplicação deve lançar o serviço (linhas 12-19 e 25). O serviço de migração propriamente está na classe `MoveThread`, construída na linha 20. O código executado na origem está nas linhas 27-38 e o código executado no destino da migração, nas linhas 40-46. As quatro *threads* são construídas na memória dos dois processadores (linhas 8-11), mas no processador de onde a *thread* migra são lançadas somente as *thread* A, B e C (linhas 28-33), enquanto no destino somente a *thread* D é lançada (linhas 41-42).

A decisão de migrar uma *thread* deve ser tomada por um outro serviço, baseado em restrições, ocupação da CPU e/ou da memória, e em objetivos, como distribuição uniforme na rede, por exemplo. Para efeito de demonstração, neste exemplo a decisão de migrar a *thread* (`TaskC`) é tomada com base no seu número de execuções, no caso, após a segunda execução (linha 35). Na invocação do serviço de migração (linha 36) é

informada a *thread* que vai migrar e o endereço lógico de destino na rede, informações que deverão estar disponíveis para quem pede a migração.

Assim como na origem, no destino o serviço de migração também deve ser ativado por alguma outra tarefa. Este serviço, que toma a decisão de migrar uma tarefa, o faz com base em informações oriundas do destino. O módulo deste serviço em execução no destino deve solicitar o serviço de migração executando uma chamada ao método `receiveThread()` (linha 44).

```
 1 import saito.sashimi.*;
 2 import saito.sashimi.realtime.*;
 3 import saito.sashimi.moveThrd.*;
 4
 5 public class TaskSet {
 6     //.....
 7     public static EDFScheduler sched = new EDFScheduler();
 8     public static TaskA tA = new TaskA(1);
 9     public static TaskB tB = new TaskB(2);
10     public static TaskC tC = new TaskC(3);
11     public static TaskD tD = new TaskD(4);
12     private static PeriodicParameters asmReleaseParam =
13                               new PeriodicParameters(TimeObjects._3_ms, // start
14                                                      null, // end
15                                                      TimeObjects._3_ms, // period
16                                                      TimeObjects._1_5_ms, // cost
17                                                      TimeObjects._3_ms); // deadline
18     public static AsyncEventsMechanism asyncEventMechanism =
19                               new AsyncEventsMechanism(null,asmReleaseParam);
20     public static MoveThread mvthd = new MoveThread();
21     public static boolean _sender = true; // no receiver é feito FALSE
22
23     public static void initSystem() {
24         Scheduler.setDefaultScheduler(sched);
25         sched.setPoolingServer(asyncEventMechanism);
26
27         if (_sender) {
28             tA.addToFeasibility();
29             tB.addToFeasibility();
30             tC.addToFeasibility();
31             tA.start();
32             tB.start();
33             tC.start();
34             sched.setupTimer();
35             while(tC.var < 2);
36             while (!mvthd.sendThread(tC, DEST_ADDRESS));
37             idleTask();
38         }
39
40         else {
41             tD.addToFeasibility();
42             tD.start();
43             sched.setupTimer();
44             mvthd.receiveThread();
45             idleTask();
46         }
47     }
48     // .....
49 }
```

Figura 5.15: Classe aplicação para migração de *thread*

As *threads* A, B, C e D são periódicas e estendem a classe `RealtimeThread` da API RTSJ. O código de uma delas é apresentado na Figura 5.16 apenas a título de ilustração. Para os testes do serviço de migração, as informações que caracterizam a *thread*, posição do código na ROM e dos atributos na RAM, foram inseridas manualmente. Em

(BARCELOS, 2008) é feita uma proposta de mecanismo para que o SASHIMI possa instrumentar a própria classe com estas informações, tornando a consulta a estes dados possível através de métodos da própria classe do tipo 'migrável'.

```
 1 import saito.sashimi.*;
 2 import saito.sashimi.realtime.*;
 3
 4 public class TaskC extends RealtimeThread {
 5     private static AbsoluteTime m_taskActiveTime = new AbsoluteTime(0,0,0);
 6     private static RelativeTime m_taskPrevProcTime = new RelativeTime(0,0,0);
 7     protected static RelativeTime period = TimeObjects._30_ms; // periodo
 8     protected static RelativeTime cost = TimeObjects._200_us; // wcet
 9     protected static RelativeTime dline = TimeObjects._30_ms ; // deadline
10     protected static AbsoluteTime m_taskRelease = new AbsoluteTime(0,3,00);
11     protected static PeriodicParameters perpar =
12                             new PeriodicParameters(null, null, period, cost, dline);
13
14     public int var = 0;
15     public int m_TaskID;
16
17     public TaskC(int TaskID)  {
18         super(null,perpar);
19         m_ResumeTime = m_taskReleaseTime;
20         m_ActiveTime = m_taskActiveTime;
21         m_PreviousProcessTime = m_taskPrevProcTime;
22         m_TaskID = TaskID;
23     }
24
25     public void exceptionTask(){}
26     protected void initializeStack(){}
27
28     public void mainTask() {
29         int local=0;
30         while (true) {
31             local++;  // Counts how many times this Task was activated
32             var = local;
33             FemtoJavaIO.write(m_TaskID,10); // Writes the task ID in one
34                                             //      port for CPU time analysis
35             for (int j=0; j<200; j++) {}
36             this.waitForNextPeriod(); // Go to sleep until next timeslice
37         }
38     }
39 }
```

Figura 5.16: *Thread* usada para migração

5.1.4.1 Resultados de simulação

O serviço de migração foi testado usando o exemplo da seção anterior. O período do mecanismo de eventos assíncronos foi escolhido de tal maneira que a migração de `TaskC` conclui dentro de um período de execução. A Figura 5.17 mostra os instantes de ativação da *thread* `TaskC`, sendo o eixo x a representação do tempo em milissegundos. As duas primeiras execuções ocorrem no processador de origem e as três seguintes já ocorrem no destino. A terceira execução ocorre com retardo, em função do tempo de migração. Nota-se que a tarefa retoma seu período original de 30 ms, conforme iniciado na origem. O instante de ativação da *thread* faz parte de seus atributos, que são levados na migração. Assim, o escalonador de tarefas no destino consegue manter a *thread* dentro do seu padrão.

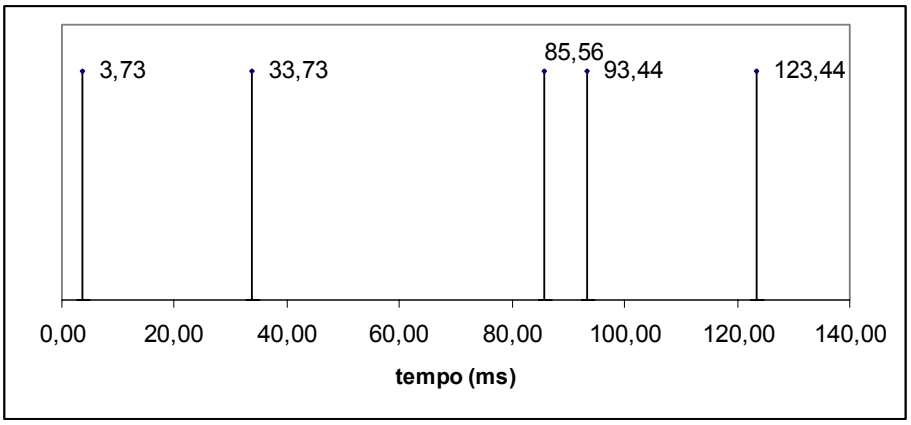

Figura 5.17: Instantes de ativação da *thread* que migra (`TaskC`)

O tempo necessário para migração de uma *thread* pode ser medido sob dois aspectos, como mostrado na Tabela 5.9. Na primeira linha é dado o custo computacional para migração da *thread* `TaskC`, ou seja, o tempo efetivamente gasto no processamento do serviço de migração. A migração não se dá de maneira ininterrupta, o que poderia comprometer o *deadline* das outras tarefas da aplicação. Ao invés disso, a transmissão é feita em blocos controlados pelo `AsyncEventMechanism` da RTSJ. Isso vai permitir que outras tarefas possam ser executadas enquanto uma tarefa migra. Por outro lado, a latência do processo vista pelo usuário do serviço fica mais elevada. E será tão maior quanto maior for o período de execução da *thread* que implementa o `AsyncEventMechanism`. A segunda linha da Tabela 5.9, então, traz a medida da latência total do serviço. Em outras palavras, a latência total na origem é o tempo desde que o método `sendThread()` é invocado até a conclusão do serviço; no destino, é o tempo desde a chegada da primeira mensagem iniciando a transmissão até o `start()` na *thread*, já no processador destino. Nos dois lados (origem e destino), o fim do serviço é transparente para o usuário, ou seja, o método de ativação do serviço não bloqueia o fluxo do código que o invoca. Como o serviço foi implementado de maneira a migrar uma *thread* de cada vez, na prática, esta latência define o tempo que outro solicitante do serviço terá que aguardar para usá-lo.

Tabela 5.9: Medidas de tempo para a migração de tarefa

	Nodo Origem	**Nodo Destino**
Custo efetivo (ms)	3,27	3,93
Latência (ms)	53,30	51,42

O tamanho de `TaskC` é de 322 bytes, incluindo código, atributos e a pilha. A taxa efetiva de migração (tamanho da tarefa dividido pelo custo de envio) da tarefa é 96KB/s. Este valor depende das latências impostas pelo serviço de comunicação, oferecido pelo nível de estrutura do *middleware*. Esta taxa pode ser melhorada usando serviço de comunicação implementado em hardware (SILVA JÚNIOR, 2007b) ou um processador com maior desempenho (BECK FILHO, 2003).

Um estudo da energia gasta para a migração é apresentado por Barcelos (2008), razão pela qual foi omitido aqui. Sabe-se que a rede representa um percentual baixo do custo total, tanto em latência quanto em energia. Os processadores são responsáveis por quase todo o custo. Isso decorre do fato da comunicação ser intensiva em uso dos processadores (software). O uso de versões hardware para implementar os serviços de comunicação pode modificar este balanço significativamente.

Em um sistema de tempo-real espera-se que, apesar dos custos de processamento da migração tanto na origem quanto no destino, as demais tarefas do sistema mantenham a sua regularidade. A *thread* `TaskB`, executada no processador de origem, tem período de 3ms e sua execução seria prejudicada pela migração de `TaskC`, se o envio não fosse fragmentado.

A Tabela 5.10 traz o uso de memória tanto no nodo origem quanto no destino. Embora o código para simulação dos processadores origem e destino tenha sido o mesmo, para avaliação do uso de memória eles foram separados. A estes custos devem ser acrescidos aqueles trazidos pela API de comunicação, já que o serviço de migração é executado sobre ela.

Tabela 5.10: Ocupação de memória pela migração de tarefa

Nodo Origem		**Nodo Destino**	
ROM	RAM	ROM	RAM
2343 Bytes	81 Bytes	1251 Bytes	45 Bytes

5.1.5 Escalonamento dinâmico de freqüência

O escalonador de tarefas EDF da API RTSJ foi modificado para inclusão do algoritmo *Cycle-Conserving*, descrito na Seção 4.7. Internamente, o escalonador de tarefas tem acesso à classe `FemtoJava`, onde foram introduzidos os métodos `getDFSFreqFactor()` e `setDFSFreqFactor()`, que permitem consultar e alterar o valor da freqüência do processador. Do ponto de vista do desenvolvedor estes dois métodos são irrelevantes, já que ele deverá usar apenas o escalonador de tarefas. Assim, o uso do DVS/DFS tal como proposto neste trabalho é extremante simples. Uma vez que o escalonamento de tensão/freqüência está inserido no escalonador de tarefas, é bastante para o desenvolvedor usar o escalonador de tarefas que contenha o DVS/DFS.

A Figura 5.18 mostra a classe principal de uma aplicação onde três tarefas são lançadas (linhas 6-8 e 12-17). Na linha 5 o escalonador de tarefas é construído, no caso o EDF com *Cycle-Conserving*. A descrição das tarefas não precisa de nenhuma modificação específica para usar o DVS/DFS.

```
 1 import saito.sashimi.*;
 2 import saito.sashimi.realtime.*;
 3
 4 public class Dvstest {
 5   public static CC_EDFScheduler sched = new CC_EDFScheduler();
 6   public static Task1 t1 = new Task1(1);
 7   public static Task2 t2 = new Task2(2);
 8   public static Task3 t3 = new Task3(3);
 9
10   public static void initSystem()   {
```

```
11      Scheduler.setDefaultScheduler(sched);
12      t1.addToFeasibility();
13      t1.start();
14      t2.addToFeasibility();
15      t2.start();
16      t3.addToFeasibility();
17      t3.start();
18      sched.setupTimer();
19      idleTask();
20    }
21
22    public static void idleTask()  {
23      while (true)    {       FemtoJava.sleep();       }
24    }
25  }
```

Figura 5.18: Aplicação multithread com DVS/DFS

5.1.5.1 Resultados de simulação

Para experimentação do DVS/DFS foi usado o simulador SystemC SERPENS, que foi modificado para que cada processador da rede possa operar a uma freqüência própria, que pode ser definida pela aplicação através de um método apropriado. O tempo gasto para a transição no valor da freqüência quando o DVS atua não foi considerado nos experimentos.

Com o simulador é possível monitorar a freqüência do processador a cada instante, bem como verificar tempos de execução das tarefas e se os seus *deadlines* são atendidos. A partir dos valores de freqüência obtidos a cada instante pode-se calcular a energia. A Figura 5.19 foi montada com resultados da simulação do exemplo acima, onde três tarefas periódicas são escalonadas usando um EDF com o algoritmo *Cycle-Conserving* DVS. A parte inferior da figura mostra o que está ocupando o processador a cada instante (tarefas, escalonador), enquanto que a parte superior mostra a freqüência de operação do processador. A freqüência máxima de operação do processador é de 100 MHz e o escalonador pode escolher entre 100, 80, 60 e 40 MHz.

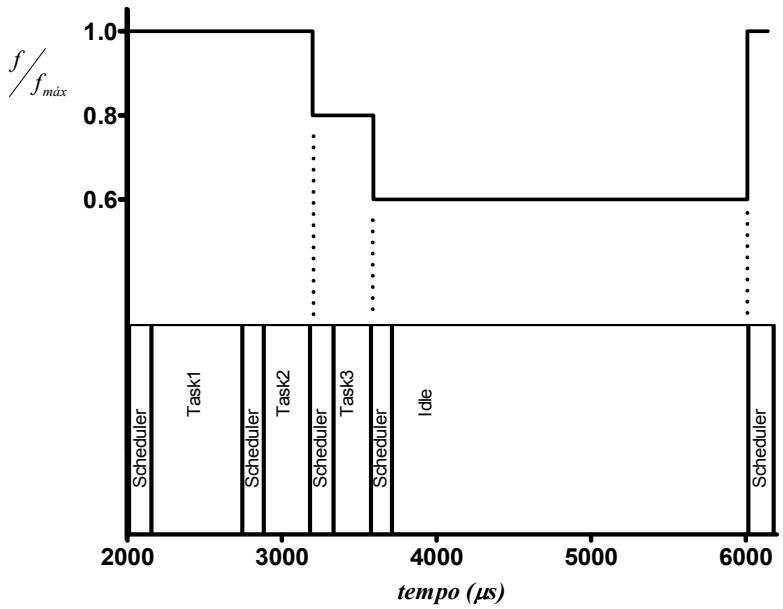

Figura 5.19: Ocupação + freqüência do processador (DVS/DFS)

A Tabela 5.11 mostra os valores de WCET e o tempo de execução medido para o mesmo ciclo de execução visto na Figura 5.19. Dado que a soma dos WCET é muito próxima do período das tarefas (3 ms), a ocupação prevista para o processador é próxima de 100%. Assim, o escalonador, em sua primeira execução, mantém a freqüência do processador no valor máximo.

Pela figura pode-se ver que após a conclusão da tarefa `Task2` o escalonador reduz a freqüência para 80 MHz. Isto ocorre porque o tempo de execução de `Task2` foi inferior ao seu WCET (*Worst Case Execution Time*) de tal forma que mesmo com o processador a 80 MHz será possível executar as demais tarefas sem perder seus *deadlines*. O mesmo não ocorre quando o escalonador executa após a finalização de `Task1` porque o tempo não consumido pela tarefa não foi suficiente para redução da freqüência. Outra redução na freqüência vai ocorrer ao final de `Task3`.

O algoritmo *Cycle-Conserving* é baseado na utilização (ocupação) do processador. Ao final de `Task3` ele soma a utilização efetiva de cada tarefa dividida pelo seu período. Se este número fosse inferior a 25% a freqüência seria ajustada para 40MHz. Como neste caso o resultado é 43%, a freqüência é levada para 60 MHz.

Tabela 5.11: Tempos de execução para tarefas do exemplo DVS/DFS

	WCET (µs)	**Custo real (µs)**
Task1	1300	587
Task2	1300	299
Task3	1300	239

O custo adicional de memória é mostrado na Tabela 5.12. Estes números refletem a diferença no tamanho do código da classe `CC_EDFScheduler` e a classe `EDFScheduler` e correspondem ao tamanho do código que implementa o algoritmo *Cycle-Conserving* DVS.

Tabela 5.12: Incremento de memória com uso do DVS *Cycle-Conserving*

	Absoluto	**percentual**
ROM	358 Bytes	5,4 %
RAM	65 Bytes	20,0 %

A Tabela 5.13 traz o incremento no tempo de processamento comparando as mesmas duas classes da tabela anterior, desta vez em outro experimento envolvendo 08 tarefas. O número de *threads* adicionadas ao escalonador afeta o desempenho, por isso foram simulados três casos para 02, 04 e 08 *threads*.

Tabela 5.13: Incremento no custo do escalonador com uso do DVS *Cycle-Conserving*

	Absoluto (µs)	**percentual**
2 threads	29	37,8%
4 threads	46	36,5%
8 threads	58	26,2%

O maior componente no custo adicional pelo uso do *Cycle-Conserving* é devido ao cálculo que determina a ocupação do processador e a freqüência mais adequada, que é realizado (*n+1*) vezes, onde *n* é o numero de tarefas adicionadas ao escalonador. Além disso, o FemtoJava não implementa divisão em hardware, requerendo um método em software para fazer isso, que é invocado dentro do cálculo de ocupação. Os resultados mostram que o custo absoluto cresce com o número de tarefas, mas o seu impacto no custo total vai caindo.

5.1.6 Escalonador de tarefas implementado em hardware

Um escalonador de prioridade fixa foi implementado em uma linguagem de descrição de hardware, para avaliação de sua área. Um simulador ciclo-a-ciclo do FemtoJava foi usado para verificar do comportamento. Foram feitos estudos de caso de implementação do escalonador de tarefas, tanto em hardware quanto em software, avaliando as latências e a área no *chip*.

Na Figura 5.20 a classe `TaskTest` é responsável por inicializar uma aplicação que utiliza o escalonador em hardware. A escolha do escalonador ocorre na linha 06, da mesma forma como ocorre para qualquer outro escalonador na API RTSJ.

```
 1 import saito.sashimi.*;
 2 import saito.sashimi.realtime.*;
 3
 4 public class TaskTest  {
 5    // static allocations    //
 6    public static FixedPriorityHWScheduler mySched
 7                                          = new FixedPriorityHWScheduler();
 8
 9    public static void initSystem() {
10       Scheduler.setDefaultScheduler(mySched);
11       t1.addToFeasibility();
12       // .....
13       t1.start();
14       // .....
15    }
16 }
17 // .....
```

Figura 5.20: Classe principal de aplicação usando escalonador em hardware

5.1.6.1 Resultados de simulação

Uma descrição VHDL do escalonador foi compilada usando a ferramenta Quartus II v.5.1 da Altera. A área total do componente é extremamente dependente do número de tarefas gerenciadas. Uma área de 3305 células lógicas é necessária para 04 tarefas e 15181 células são necessárias para 08 tarefas. O escalonador em hardware tem uma área relativamente elevada, uma vez que o processador FemtoJava demanda cerca de 3500 células lógicas (embora se trate de um processador pequeno e o custo do escalonador em hardware não seria um custo adicional significativo em um processador mais complexo). Mesmo que sejam feitas otimizações no projeto para reduzir a área do escalonador, um custo adicional em área é o preço a ser pago por um ganho em desempenho, previsibilidade e energia. De fato, cerca de 70% da área do escalonador

em hardware se deve ao bloco SyncEvent, o qual não é exatamente o escalonador, mas um bloco detector de eventos que foi projetado para máximo paralelismo.

Um conjunto de 08 tarefas sintéticas foi usado para avaliar o escalonador. A simulação foi feita no CACO-PS, sendo 20MHz a freqüência do processador FemtoJava. A Tabela 5.14 mostra os custos relacionados aos escalonadores nas versões software e hardware, para algumas execuções de 02, 04 e 08 tarefas.

Para compreender a tabela é preciso separar a ação do escalonador em dois momentos: (1) a decisão de qual será a próxima tarefa a entrar em execução, e (2) a troca do contexto da tarefa atual para a tarefa seguinte. O uso do escalonador em hardware leva o momento (1) para o hardware, deixando o momento (2) em software. Usando a API RTSJ sobre o processador FemtoJava, este momento (2) não depende da quantidade de tarefas em execução no sistema, ou do limite máximo de tarefas. Portanto, o que diferencia os dois escalonadores é o momento (1), que depende do número máximo de tarefas admitido pelo sistema, definido em projeto. O tempo de execução do escalonador (Tabela 5.14), seja ele hardware ou software, mede o tempo gasto com ele em processamento (no FemtoJava). Ou seja, no caso do escalonador em software o tempo de processamento inclui os dois momentos e na versão hardware, apenas o momento (2). Para o escalonador em hardware a latência que depende do tamanho do sistema não afeta o seu tempo de execução 'visível', mas o tempo que ele leva para decidir qual tarefa está pronta para executar. Entretanto, este custo pode afetar a granularidade de tempo que ele é capaz de detectar. Ou seja, se ele gasta 16 ciclos de relógio, por exemplo, para verificar o RTC (*Real-Time Clock*) e avaliar se há alguma tarefa para ser ativada naquele instante, os tempos de ativação de tarefa informados ao bloco SyncEvent (Seção 4.8.1.1) do escalonador devem ser arredondados de maneira a desconsiderar tempos menores que 16 ciclos de relógio.

Em resumo, para o escalonador em hardware o tempo de execução não depende do número de tarefas, uma vez que este custo não corresponde ao processamento do algoritmo de escalonamento. Por isso, na Tabela 5.14, ao aumentar o número de tarefas a latência não se altera quando o escalonador em hardware é usado. A versão software, entretanto, percorre uma tabela verificando que tarefas estão prontas para execução, de modo que o seu desempenho é sensível ao número de tarefas e à posição na tabela em que se encontra a tarefa selecionada para execução. O escalonador em software também tem o seu custo elevado por ter sido desenvolvido em uma linguagem de alto nível (Java) e que usa o conceito de objetos.

Tabela 5.14: Tempo usado pelo escalonador

Num. de Tarefas	Tempo de Execução (ms)			
	Escalonador em Hardware	Escalonador em Software		
		Méd	Min	Máx
02	0,0575	0,335	0,229	0,364
04	0,0575	0,514	0,373	0,598
08	0,0575	0,778	0,585	0,978

5.1.7 Comunicação implementada em hardware

A exemplo do escalonador de tarefas, esta função é encapsulada em uma classe (chamada `HwTransport`) e é usada da mesma maneira que a implementação em software (chamada `Transport`).

5.1.7.1 Resultados de simulação

Para os testes do serviço de comunicação em hardware, a camada de enlace de dados implementada é compatível com o barramento CAN (*Controller Area Network*) (BOSCH, 1991), que é síncrono e utiliza uma estratégia de acesso ao meio do tipo CSMA/AMP (*Carrier Sense Multiple Access with Arbitration on Message Priority*) (WOLF, 2001). Neste protocolo não há colisão, pois o pacote de maior prioridade sempre ganha acesso ao barramento. Esta estratégia foi escolhida para atender aos requisitos de tempo-real e pode ser usada tanto em barramentos inter-dispositivo (como no CAN-bus) quanto em barramentos intra-chip.

As simulações foram feitas no simulador ciclo-a-ciclo CACO-PS. Dois processadores operando a 20MHz foram conectados, sendo que um deles envia 20 mensagens, cujos tamanhos variam de 01 a 20 bytes, e o outro as recebe.

Para efeito de comparação, a Figura 5.21 mostra o tempo gasto enviando e recebendo as mensagens na implementação em software, onde o eixo x indica o comprimento das mensagens. Usando a implementação em hardware, as latências para envio e recepção de mensagens se reduzem muito, como se vê na Figura 5.22. Para uma mensagem com 07 bytes, por exemplo, o custo de transmissão cai de 0,201 ms, na implementação em software, para 0,073 ms. Este custo é devido às operações executadas pela classe `HwTransport`. O tempo gasto pelo hardware para construir o pacote e enviar para a camada física da rede é desprezível. Na verdade o hardware usa apenas 06 ciclos de relógio para enviar uma mensagem completa, que significa menos de 1 μs considerando que a freqüência de operação é 20MHz.

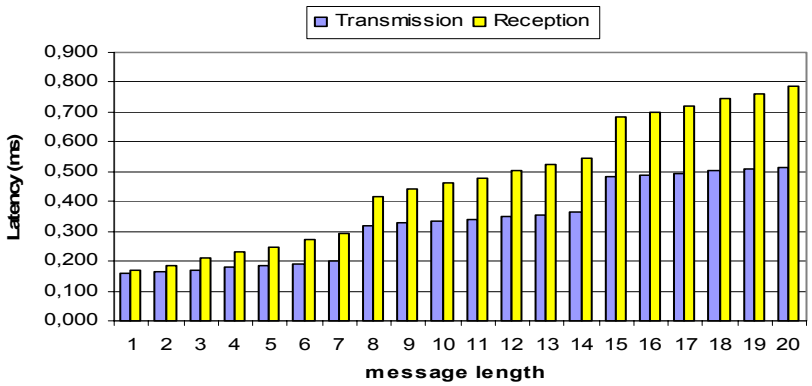

Figura 5.21: Latências de comunicação (implementação software)

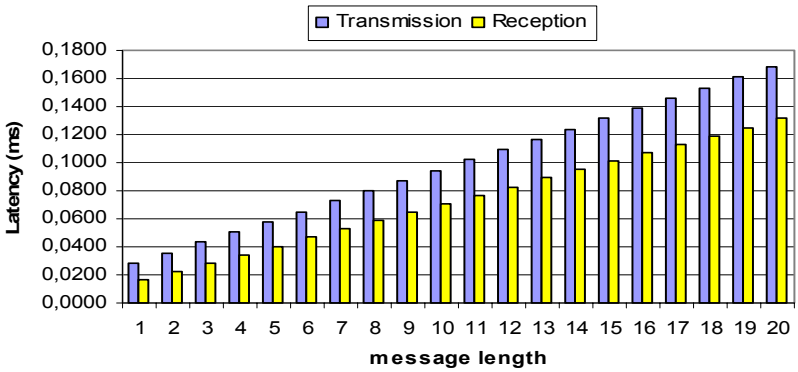

Figura 5.22: Latências de comunicação (implementação hardware)

Para a implementação hardware, observa-se que a latência aumenta linearmente com o crescimento do tamanho da mensagem. Isto ocorre porque o processador apenas entrega a mensagem para o hardware dedicado, o qual faz a fragmentação quase que instantaneamente. Também nota-se que, quando a implementação hardware é utilizada, o custo de transmissão é maior que o de recepção. Isto acontece porque, no processo de envio, a API precisa entregar informações sobre a mensagem (*time out*, endereço de destino, porta de conexão) para o bloco em hardware, enquanto durante a recepção, o hardware sinaliza para o processador somente quando a mensagem está pronta e a API (processador) apenas lê a mensagem.

Em (SILVA JÚNIOR, 2006a) é apresentado um estudo de caso envolvendo 05 processadores para controle automotivo. Naquele exemplo, um dos laços de controle utiliza tanto mensagens por conexão quanto broadcast.

A energia é fortemente relacionada com o tempo gasto pelas classes Java processando as mensagens. Assim, a energia gasta é proporcional à latência e seus gráficos têm formas semelhantes.

A descrição VHDL do hardware para comunicação foi sintetizada no FPGA da Xilinx Virtex-II Pro XC2VP30. A Tabela 5.15 mostra os custos em área necessários para tratar até 04 conexões simultâneas e mensagens de até 20 bytes. Os dados foram obtidos através da ferramenta ISE Project Navigator 7.1i da Xilinx. A área total do bloco de comunicação não é desprezível, se comparada com a área do processador FemtoJava.

Tabela 5.15: Indicadores de área no FPGA para comunicação em hardware

Componente	Slices	Slice FFs	4 input LUTs	BRAMs
Transporte+Enlace em hw	2566	2922	4162	0
FemtoJava Multiciclo	1180	412	2163	0
Interface CAN	486	508	730	2

Detalhes construtivos, bem como medidas de área de cada componente da APICOM em hardware, podem ser encontrados em (KUNZ, 2007).

5.1.8 Resumo do impacto em memória

Para avaliar o gasto de memória introduzido pelo *middleware* num caso envolvendo diversos serviços foi construída a Tabela 5.16. A primeira coluna mostra o componente do *middleware* seguida das memórias de código (ROM) e de dados (RAM). Para o serviço de invocação de método remoto foi tomado o lado servidor, que contém o método a ser invocado. Para o serviço de migração, o lado de onde sai a tarefa. Os dois serviços compartilham a APICOM e a API-RTSJ. A APICOM, neste caso, inclui tanto os métodos para envio quanto para recepção, já que ambos são requisitados pelos serviços de invocação de método remoto e de migração. Também optou-se por usar uma configuração de maior consumo de memória, pacote de 49 bytes e mensagem de 500.

Para efeito de comparação, ao final da tabela foi incluído o custo de uma aplicação típica de sistemas embarcados, um tocador de mp3 (Mp3Player). Os números da tabela mostram que o consumo de memória do *middleware* é aceitável para aplicações reais.

Tabela 5.16: Ocupação de memória por diversos serviços em conjunto

Componente	ROM	RAM
Método remoto (servidor)	2139 Bytes	118 Bytes
Migração de tarefas (origem)	2343 Bytes	81 Bytes
APICOM (Pack49-Msg500)	4493 Bytes	6345 Bytes
API-RTSJ + DVS	4849 Bytes	242 Bytes
TOTAL	13824 Bytes	6786 Bytes
Aplicação	**ROM**	**RAM**
Mp3Player	48548 Bytes	63702 Bytes

5.2 Casos envolvendo o uso de vários serviços

A idéia do *middleware* é dar suporte a aplicações complexas, que envolvam vários dos serviços apresentados nas seções anteriores deste capítulo. Nesta seção são apresentados alguns casos onde diversos serviços do *middleware* são integrados.

A Tabela 5.17 traz uma lista com os serviços implementados e as aplicações que utilizam cada um deles. O FIR_1 é um exemplo onde um filtro FIR implementado em hardware obtém os dados de entrada em um objeto remoto. No caso FIR_2 ocorre uma substituição do objeto software por um equivalente hardware em tempo de execução. O exemplo Migração mostra uma tarefa migrar de um processador para outro, com otimização de energia por DVS/DFS. O exemplo FIR_1 foca na abstração, tanto de localização quanto de implementação. Os exemplos FIR_2 e Migração mostram o mecanismo de otimização de energia (DVS/DFS) aliado à implementação de objeto em hardware (FIR_2) e à comunicação (Migração).

Tabela 5.17: Aplicações exemplo e os serviços do *middleware* utilizados

Aplicação / Serviços	FIR_1	FIR_2	Migração
DVS/DFS		X	X
APICOM	X		X
APICOM + eventos			X
Método remoto	X		
Migração			X
Objeto em Hardware	X	X	

5.2.1 *Thread* hardware obtém dados de entrada em outro processador (FIR_1)

Neste exemplo, esquematizado na Figura 5.23, um filtro FIR implementado em hardware é encapsulado pela classe `FIR10hw` e tem seus dados de entrada fornecidos por outro objeto (`InputDriver`). O componente FIR em hardware não aparece na figura. O objeto `InputDriver` é uma *thread* que obtém os dados usando invocação a um método de um objeto remoto (`DataProvider`).

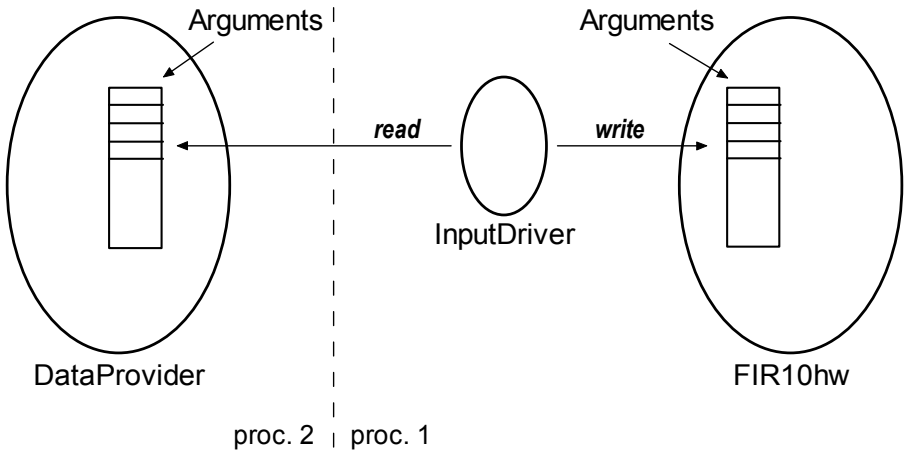

Figura 5.23: FIR hardware usando dados remotos

O diagrama de classes da aplicação no lado servidor, onde estão os dados de entrada, é mostrado na Figura 5.24. As classe em destaque (cor escura) são da aplicação e as demais, das APIs, sendo que somente as que interagem diretamente com a aplicação são mostradas. A classe `ProdTaskset` é responsável pela construção dos objetos da aplicação e pela inicialização. A classe `DataProvider` encapsula os métodos que serão invocados remotamente e contém um objeto `Arguments`, com os dados de entrada para o filtro FIR. A `Task1` é uma tarefa sintética apenas para gerar uma carga adicional no processador.

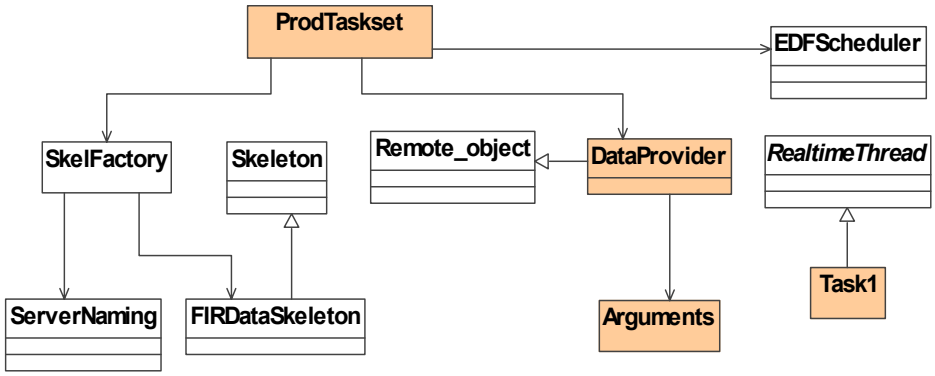

Figura 5.24: Diagrama de classe FIR hw com dados remotos – produtor de dados

Para o lado cliente, onde está o filtro FIR, o diagrama de classe é mostrado na Figura 5.25. Novamente, as classes da aplicação são colocadas em destaque (cor escura). A inicialização dos objetos e da aplicação é feita na classe `FirTaskset`. A classe `Fir10hw` implementa o filtro FIR em hardware e contém um objeto tipo `Arguments`, que vai receber os dados obtidos remotamente pelo `InputDriver`. Novamente, uma tarefa `Task1` sintética foi incluída com o objetivo de gerar uma carga adicional no processador.

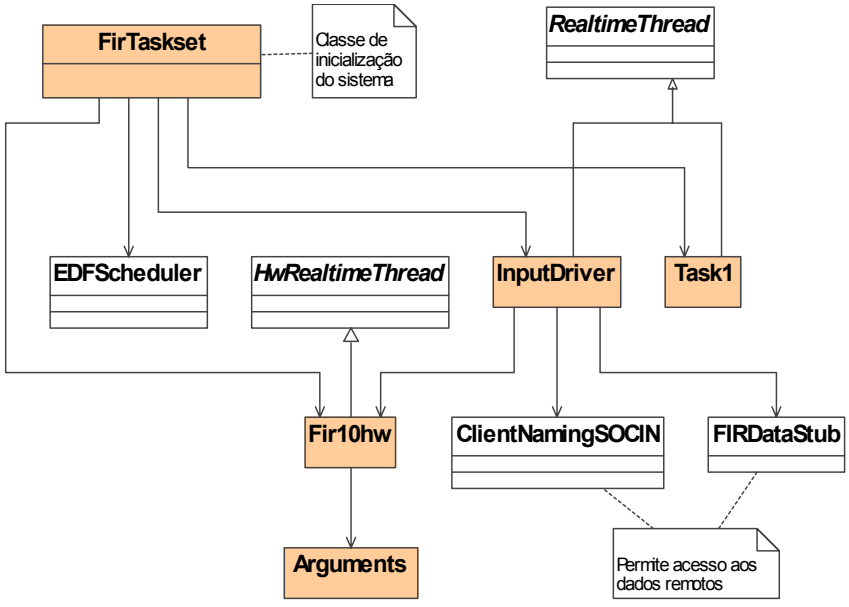

Figura 5.25: Diagrama de classe FIR hw com dados remotos – consumidor de dados

O código Java utilizado para esta aplicação está incluído no apêndice.

5.2.1.1 Resultados de simulação

Para a simulação foi usado o SERPENS, utilizando a NoC SoCIN para conectar os processadores, que funcionam a uma freqüência de 100MHz. A Tabela 5.18 foi

construída com os dados gerados após processamento de vinte valores de entrada pela aplicação. A linha custo mostra o tempo médio de execução das tarefas. O *jitter* foi obtido a partir das medidas de atraso na ativação das tarefas em relação ao tempo previsto para sua ativação. Os baixos valores de *jitter* mostram um comportamento bastante estável em relação às latências. Mais do que um atraso baixo o sistema apresenta uma baixa variação neste atraso. O custo da *thread* `InputDriver` deve-se mais ao tempo de espera pelo remoto do que ao processamento propriamente. Como já foi mencionado, no lado remoto (servidor) há uma *thread* periódica que administra as conexões e o seu período de ativação é responsável pela maior parte do tempo de espera imposto a quem invoca um método remotamente. Todavia, como convém a uma aplicação de tempo-real, este tempo de espera é limitado e conhecido previamente.

Tabela 5.18: Medidas de tempo para exemplo FIR_1

	Thread		
	InputDriver	**Fir10hw**	**Task1**
Custo	1,18 ms	17,75 µs	119,00 µs
Período	2,00 ms	2,00 ms	2,00 ms
***Jitter* máx**	1,00 µs	0,60 µs	4,00 µs

5.2.2 DVS aliado a tarefas implementadas em hardware (FIR_2)

Este exemplo foi proposto para demonstrar o caso mostrado na Seção 4.7, Figura 4.1, onde um componente da aplicação implementado em software é substituído por outro implementado em hardware, permitindo que o escalonador DVS/DFS reduza a freqüência de operação.

O diagrama de classes do exemplo é mostrado na Figura 5.26. A classe `Taskset` constrói e inicializa os objetos da aplicação. Uma implementação software do filtro FIR (`Fir100sw`) inicia executando, junto com duas tarefas sintéticas (`Task1` e `Task2`). Após certo tempo, a *thread* software é removida do escalonador (`CC_EDFScheduler`) e a *thread* hardware (`Fir100hw`) é adicionada.

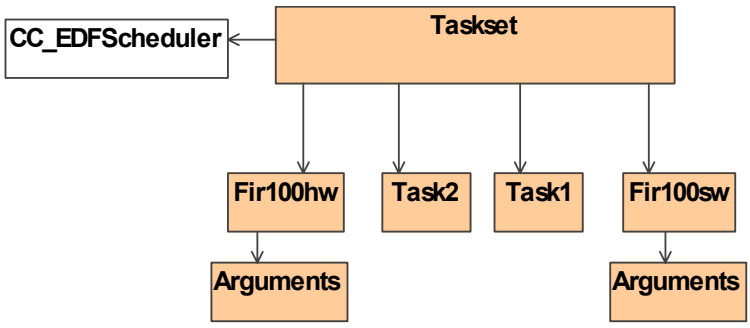

Figura 5.26: Diagrama de classes da aplicação FIR_2

5.2.2.1 Resultados de simulação

Para a simulação a freqüência máxima do processador foi definida em 100MHz, podendo o DVS/DFS alterá-la para 80, 60 ou 40MHz. A `Task1` foi encarregada de fazer a remoção do FIR em software e ativação do FIR em hardware.

A Figura 5.27 mostra dois gráficos no mesmo eixo x, que contém a linha do tempo. Estão mostrados na figura somente os instantes próximos ao momento em que a FIR-software é removida e a FIR-hardware é adicionada. O gráfico inferior mostra o que está sendo executado no processador (escalonador, *threads*), ou se ele está desocupado (*Idle*). O gráfico superior representa a freqüência de operação do processador normalizada em relação à freqüência máxima. No tempo 8000 µs a freqüência está em 60MHz porque a ocupação do processador é tal que não é necessário operar a 100MHz. Este valor foi ajustado pelo escalonador desde a primeira vez que ele executou, em função dos valores de WCET das tarefas adicionadas. Como as tarefas sempre consumiram o seu WCET este valor não se alterou nenhuma vez. Quando a `Task1` é executada a 11,3 ms, ela substitui a FIR-sw pela FIR-hw, as quais possuem WCET diferentes. Na execução seguinte do escalonador (11,5 ms) uma nova ocupação máxima é determinada, o que permite ajustar a freqüência para 40MHz. Finalmente, a ação do DVS ocorre apenas sobre o processador. O componente hardware sempre opera na sua freqüência máxima, que neste caso é 100 MHz.

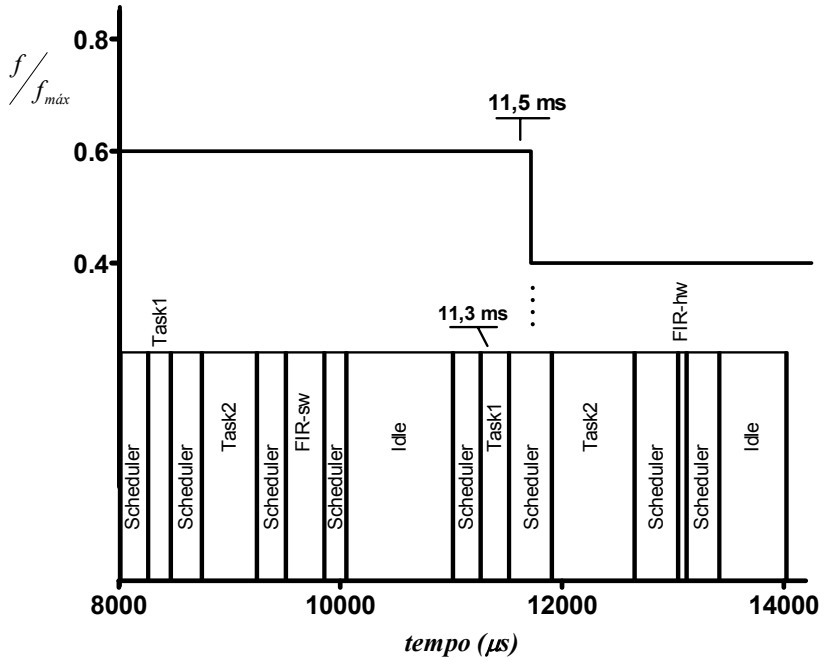

Figura 5.27: Ocupação + freqüência do processador (FIR_2)

Foram feitas medidas adicionais de tempo, mostradas na Tabela 5.19, para demonstrar o determinismo do sistema. São mostrados o custo (tempo de execução) e a variação (*jitter*) no atraso de ativação das tarefas adicionais do sistema, `Task1` e `Task2`. O tempo de execução destas tarefas é menor quando a FIR-sw está em execução porque a freqüência de operação é maior. Isso decorre do maior WCET da FIR-sw, que leva o escalonador (`CC_EDFScheduler`) a requerer um freqüência maior, que atenda a todos os

deadlines. O baixo *jitter* no tempo de ativação mostra que as tarefas mantêm seu comportamento periódico bastante estável. As tarefas `Task1` e `Task2` têm o mesmo período dos filtros FIR, ou seja, 3 ms.

Tabela 5.19: Medidas de tempo para exemplo FIR_2

	Antes de 11,5 ms		*Após 11,5 ms*	
	Tempo de execução (µs)	***Jitter* (µs)**	**Tempo de execução (µs)**	***Jitter* (µs)**
Task1	205	1,00	309	0,0
Task2	498	0,67	748	0,0

5.2.3 Migração de tarefas, DVS e comunicação orientada a eventos

Para este caso é retomado o exemplo de migração de tarefas da Seção 5.1.4, desta vez usando um escalonador de tarefas DVS/DFS. O propósito é observar o efeito do escalonador DVS/DFS quando ocorre comunicação dentro de um serviço de alto nível. O código Java desta aplicação é o mesmo da Seção 5.1.4, sendo substituído o escalonador pelo `CC_EDFScheduler`.

5.2.3.1 Resultados de simulação

A simulação foi feita no SERPENS utilizando a NoC SoCIN para conectar os processadores, que podem operar a 100, 80, 60 ou 40MHz, de acordo com o controle exercido pelo DVS/DFS.

A Figura 5.28 foi montada com os dados de simulação, sendo selecionado um intervalo onde mensagens são tratadas pelo `AsyncEventHandler`. Na parte inferior da figura é mostrado o que está sendo executado no processador (escalonador, *threads*) ou se ele está desocupado (*Idle*). A parte superior da figura representa a freqüência de operação do processador normalizada em relação à freqüência máxima. A primeira execução de `AsyncEventHandler` na figura tem um tempo de execução de aproximadamente 1,1 ms. Este tempo é menor que o WCET de `AsyncEventHandler` e o escalonador, ao determinar a ocupação do processador a seguir, usa este tempo extra para reduzir a freqüência para 60MHz. Na execução seguinte de `AsyncEventHandler` não há mensagens a serem tratadas e o seu tempo de execução é praticamente zero (18 µs). Desta vez o tempo extra é maior do que na execução anterior e o escalonador pode reduzir ainda mais a freqüência do processador, agora para 40MHz. Pode-se notar ainda que a freqüência é elevada para 80MHz a 63,1 ms, após o primeiro *Idle*. Isto ocorre porque o escalonador considera o WCET da tarefa `AsyncEventHandler` antes de colocá-la em execução. Somente quando a tarefa é concluída e o escalonador verifica que houve 'sobra' de tempo é que a freqüência é reduzida. Este procedimento conservador vai assegurar que as tarefas tempo-real não percam seus *deadlines*.

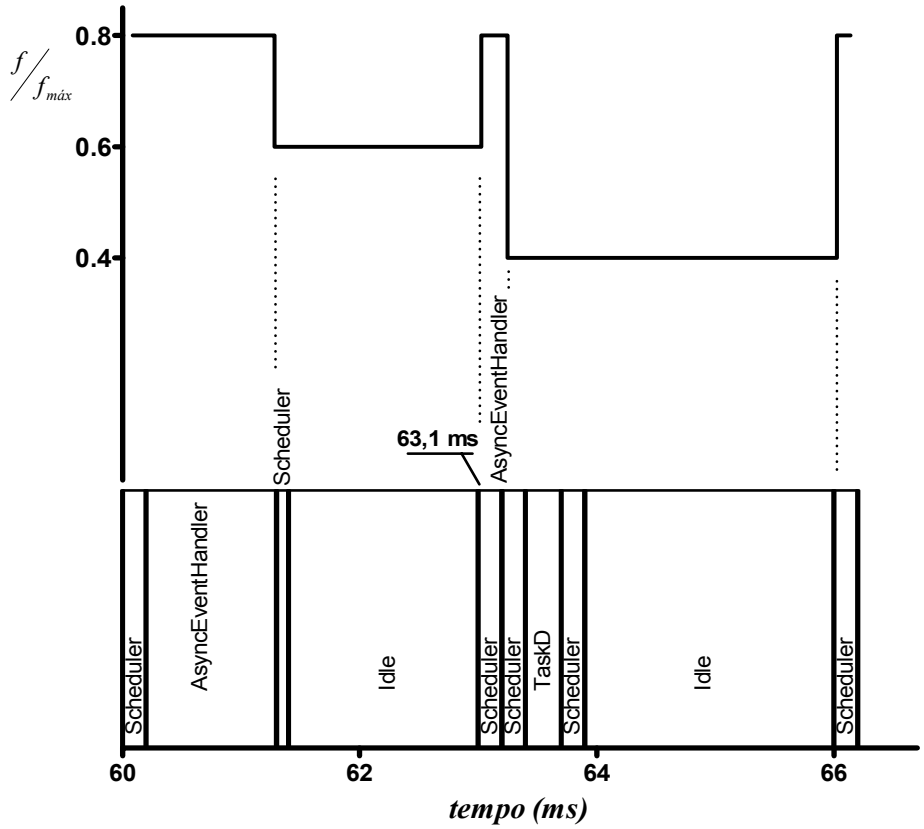

Figura 5.28: Ocupação + freqüência do processador (Migração)

6 CONCLUSÕES E TRABALHOS FUTUROS

Com o aumento na complexidade das aplicações em sistemas embarcados a demanda por capacidade computacional tem aumentado. E, sendo o uso de energia ainda mais importante em sistemas embarcados do que para em sistemas de propósito geral, a melhor maneira de aumentar o desempenho é aumentando o numero de processadores no chip.

Desenvolver sobre uma plataforma MPSoC implica em lidar com as dificuldades inerentes às aplicações distribuídas, relacionadas à concorrência. Por esta razão crescem as pesquisas na busca por soluções para este problema. Paralelamente, plataformas MPSoC são com freqüência propostas para aplicações multimídia, as quais requerem propriedades de tempo-real.

Uma das alternativas para aumentar a produtividade no desenvolvimento de aplicações distribuídas é o uso de um *middleware*, camada intermediária que abstrai a complexidade da distribuição dos componentes da aplicação.

Este trabalho especifica e implementa um *middleware* para desenvolvimento de aplicações embarcadas com restrições de tempo-real sobre plataformas MPSoC onde o processador tem memória e desempenho muito restritos, oferecendo mecanismos de baixo consumo de energia.

Como recurso de projeto de sistemas embarcados, o *middleware* oferece seus serviços em dois momentos do ciclo de vida da aplicação. Na fase de especificação e projeto ele permite abstrair os mecanismos de comunicação utilizados, bem como a localização e forma de implementação (hw/sw) dos objetos invocados pela aplicação. A intenção é abrir espaço para exploração sem que o projetista fique preso a particularidades na interface de acesso aos componentes da aplicação ou dos serviços da plataforma, além de facilitar o reuso de componentes oriundos de projetos anteriores, em casos de mudanças na plataforma. No segundo momento, que é a fase de execução, o *middleware* oferece adaptação dinâmica do hardware para operação em mínima energia (DVS/DFS) e alocação automática dos objetos da aplicação em função de demandas geradas pelo usuário do sistema.

Abordagens que propõem *middlewares* adaptáveis usualmente oferecem dois tipos de adaptação, ambas em software. (1) A aplicação se adapta a partir de informações fornecidas pelo *middleware*, ou (2) o *middleware* se adapta em função de parâmetros previamente carregados pela aplicação. Este trabalho explora adaptações na arquitetura da plataforma, usando serviços em hardware, proporcionando ganhos em energia e em

desempenho e atendendo a restrições de tempo-real. Esta exposição de propriedades do hardware no nível do *middleware* não se encontra em nenhuma das propostas na literatura, até o momento.

São dadas a seguir as contribuições deste trabalho com as respectivas citações do que já foi publicado até o momento.

À implementação da especificação RTSJ, introduzida por Wehrmeister (2004), foi acrescentado um serviço de comunicação entre processadores que utiliza troca de mensagens (SILVA JÚNIOR, 2006a) (SILVA JÚNIOR, 2007a). Um mecanismo de escalonamento dinâmico de freqüência e tensão (DVS - *Dynamic Voltage Scaling*) também foi adicionado ao escalonador de tarefas da implementação RTSJ inicial. Ainda neste nível de base foram propostos e implementados serviços do *middleware* em hardware (SILVA JÚNIOR, 2005b) (SILVA JÚNIOR, 2007b).

Em mais alto nível o presente trabalho oferece mais duas contribuições originais: objetos da aplicação podem ser implementados em hardware (SILVA JÚNIOR, 2008) e tarefas em software podem migrar de um processador para outro. A migração de tarefas toma como base o trabalho de Barcelos (2008) e eleva o nível de abstração daquela proposta. Adicionalmente, um serviço de invocação remota de métodos com restrições tempo-real foi implementado e um serviço de alocação de tarefas foi especificado.

Para minimização da energia o *middleware* utiliza dois mecanismos: escalonador de tarefas com DVS/DFS e implementações em hardware, tanto em seus serviços quanto na aplicação. O escalonador DVS/DFS atua transparentemente na busca pela menor freqüência de operação do sistema e mostrou-se particularmente interessante nos procedimentos de comunicação, evitando gastos desnecessários na ausência de mensagens. O interfaceamento com as versões hardware tanto de serviços quanto de partes de aplicação mostrou-se viável em relação à facilidade de uso e o seu desempenho é satisfatório. Estudos futuros poderão viabilizar a transição automática entre as versões hardware e software, aumentando a capacidade de adaptação da plataforma e do *middleware*.

O serviço básico de comunicação (APICOM) oferece flexibilidade para que diferentes arquiteturas de rede possam ser utilizadas. A experimentação feita utiliza o barramento CAN-bus e a NoC SoCIN, que tem estrutura em malha. Os serviços de mais alto nível do *middleware* operam sobre esta camada e podem ser facilmente portados para outras redes desde que uma classe da APICOM seja reescrita para a rede alvo. Sabe-se que para comunicação em aplicações multimídia a definição de políticas de qualidade de serviço (QoS) é importante. O suporte a QoS nos serviços de comunicação foi deixado como um trabalho futuro. Juntamente com este objetivo está a ampliação dos roteadores da NoC, para que esta QoS tenha suporte na infra-estrutura da rede usada.

Os serviços de alto nível ‘Objeto em hardware’, ‘Objeto remoto’ e ‘Migração de tarefas’ foram implementados e avaliados por simulação. O gerenciamento de memória da plataforma impõe endereçamento absoluto, o que limita a experimentação do serviço de ‘Migração de tarefas’, que deverá ser revisitado em breve. Esta limitação também deverá influenciar a implementação do serviço de ‘Alocação de tarefa’, que foi deixado para trabalho futuro.

De uma maneira mais geral, o *middleware* deste trabalho oferece mecanismos de: (1) adaptação de hardware no atendimento a requisitos não funcionais de aplicações embarcadas, como tempo-real e energia; e (2) programação em alto nível para plataformas MPSoC embarcadas executando aplicações de tempo-real.

A partir do uso do *middleware*, fica facilitado para o projetista da aplicação o reuso dos seus serviços e daqueles oferecidos pela plataforma em diferentes aplicações. No Apêndice C constam tabelas que dão uma idéia do número de linhas de código que pode ser reusado. Da mesma maneira, aplicações escritas sobre o *middleware* podem ser portadas para outras plataformas onde ele possa ser executado.

O *middleware* deste trabalho foi pensado com o objetivo de ser aplicado em MPSoCs. Todavia, o processador usado na plataforma tem algumas propriedades que o tornam uma boa opção para redes de sensores. Avaliar a possibilidade de portar este *middleware* para uma aplicação de redes de sensores é um trabalho futuro promissor.

O *deployment* (colocação em serviço) de aplicações não está incluído nas especificações deste *middleware*. Como um trabalho futuro, este recurso poderia facilitar ainda mais o desenvolvimento de aplicações distribuídas fazendo o mapeamento no código de parâmetros de localização, por exemplo.

REFERÊNCIAS

ACQUAVIVA, A. et al. Assessing Task Migration Impact on Embedded Soft Real-Time Streaming Multimedia Applications. **EURASIP Journal on Embedded Systems**, New York, v. 2008, n.2, p.1-15, Apr. 2008.

AGRON, J. et al. FPGA Implementation of a Priority Scheduler Module. In: IEEE INTERNATIONAL REAL-TIME SYSTEMS SYMPOSIUM, RTSS, 25., 2004, Lisboan, Portugal. **RTSS 2004**: Work- In- Progress: Proceedings... [S.l.:s.n.], 2004. [4p.].

AL ENAWY, T. A.; AYDIN, H. On Energy-Constrained Real-Time Scheduling. In: EUROMICRO CONFERENCE ON REAL-TIME SYSTEMS, ECRTS, 16., 2004, Catania, Italy. **Proceedings...** Los Alamitos, CA: IEEE Computer Society, 2004.

ANDERSSON, P.; KUCHCINSKI, K. Java to Hardware Compilation for non Data Flow Applications. In: EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN, DSD, 8., 2005. **Proceedings...** Washington, DC: IEEE Computer Society, 2005. p. 330-337.

ANDREWS, D. L. et al. Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link. **IEEE Micro**, Los Alamitos, CA, v.24, n.4, p.42-53, 2004.

ANDREWS, D. L. et al. Evaluation of the Hybrid Multithreading Programming Model using Image Processing Transform. In: RECONFIGURABLE ARCHITECTURES WORKSHOP, RAW, 12., 2005, Denver, CO. **Proceedings...** [S.l.]: IEEE Computer Society, 2005.

ATITALLAH, R. B. et al. Estimating Energy Consumption for an MPSoC Architectural Exploration. In: ARCHITECTURE OF COMPUTING SYSTEMS, ARCS, 2006, Frankfurt, Germany. **Proceedings...** Berlin: Springer, 2006. p. 298-310. (Lecture Notes in Computer Science, v. 3894).

AYDIN, H.; YANG, Q. Energy-Aware Partitioning for Multiprocessor Real-time Systems. In: INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING, IPDPS, 17., 2003. **Proceedings...** Washington, DC: IEEE Computer Society, 2003. p.113-121.

BALARIN, F. et al. **Hardware-Software Co-design of Embedded Systems**: the POLIS Approach. Boston: Kluwer Academic Publishers, 1997.

BARCELOS, D. **yAfJS**: yet Another femtoJava Simulator. Instituto de Informática, UFGRS, Porto Alegre. Publicação interna.

BARCELOS, D. **Modelo de migração de tarefas para MPSoCs baseados em redes-em-chip**. 2008. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, UFGRS, Porto Alegre.

BARR, R. et al. On the Need for System-Level Support for Ad Hoc and Sensor Networks. **Operating Systems Review**, New York, v.36, n.2, p.1-5, 2002.

BECK FILHO, A.C.S.; CARRO, L. Low Power Java Processor for Embedded Applications. In: IFIP WG 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, VLSI-SoC, 11., 2003, Darmstadt, Germany. **Proceedings...** Darmstadt, Germany: IFIP, 2003. p. 239-244.

BECK FILHO, A.C.S. et al. CACO-PS: A General-Purpose Cycle-Accurate Configurable Power Simulator. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 16., 2003, São Paulo. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 2003.

BENINI, L.; DEMICHELI, G. Networks on Chip: A New SoC Paradigm. **IEEE Computer**, [S.l.], v.35, n.1, p. 490-504, Jan. 2002.

BERNSTEIN, P.A. Middleware: A Model for Distributed System Services. **Communications of the ACM**, New York, v.3, n.2, p.86-97, Feb. 1996.

BERTOZZI, S. et al. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In: DESIGN, AUTOMATION, AND TEST IN EUROPE, DATE, 9., 2006, Munich, Germany. **Proceedings...** New York: ACM SIGDA, 2006. p. 15-20.

BOLLELLA, G. et al. **The Real-Time Specification for Java**. Massachusetts: Addison Wesley Longman, 2000. 195 p.

BORG, A.; WELLINGS, A. A Real-Time RMI Framework for the RTSJ. In: EUROMICRO CONFERENCE ON REAL-TIME SYSTEMS, ECRTS, 15., Porto, Portugal, 2003. **Proceedings...** [S.l.]: IEEE, 2003. p. 238-246.

BORG, A.; GAO, R.; AUDSLEY, N. A Co-design Strategy for Embedded Java Applications Based on a Hardware Interface with Invocation Semantics. In: INTERNATIONAL WORKSHOP ON JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS, JTRES, 4., 2006, Paris. **Proceedings...** New York: ACM Press, 2006. p. 58-67.

BOSCH. **CAN Specification 2.0, CAN in Automation**. 1991. Disponível em: <http://www.infineon.com/cmc_upload/migrated_files/document_files/Application_Notes/can2spec.pdf>. Acesso em: maio 2005.

BOULIS, A.; HAN, C.C.; SRIVASTAVA; M. B. Design and Implementation of a Framework for Programmable and Efficient Sensor Networks. In: INTERNATIONAL CONFERENCE ON MOBILE SYSTEMS, APPLICATIONS AND SERVICES, MobiSys, 1., 2003, San Franscisco, CA. **Proceedings...** New York: ACM, 2003. p. 187-200.

BRAY, M. **Middleware**. 1997. Disponível em: <http://www.sei.cmu.edu/str/descriptions/middleware_body.html>. Acesso em: fev. de 2003.

BRIÃO, E. W. **Métodos de EEP em tempo de execução para sistemas embarcados baseados em redes-em-chip tempo-real soft**. 2008. Tese (Doutorado em Ciência da Computação) - Instituto de Informática, UFGRS, Porto Alegre.

BURLESON, W. et al. The Spring Scheduling Co-Processor: A Scheduling Accelerator. **IEEE Transactions on VLSI Systems**, New York, v.7, n.1, p. 38-48, Mar. 1999.

BURNS, A.; WEELINGS, A. **Real-time systems and programming languages**. 2nd ed. Harlow: Addison-Wesley, c1997. 611 p.

CARRO, L.; WAGNER, F. Sistemas Computacionais Embarcados. In: JORNADAS DE ATUALIZAÇÃO EM INFORMÁTICA, 22., 2003, Campinas. **Livro Texto**. Campinas: SBC, 2003.

CORSARO, A.; SCHMIDT, D.C. The Design and Performance of the jRate Real-Time Java Implementation. In: MEERSMAN, R. et al. (Ed.). **On the Move to Meaningful Internet Systems 2002**: CoopIS, DOA, and ODBASE. Berlin: Springer- Verlag, 2002. p. 900-921. (Lecture Notes in Computer Science, v.2519).

DALLY, W. J.; TOWLES, B. Route Packets, Not wires: On-chip Interconnection Networks. In: DESIGN AUTOMATION CONFERENCE, DAC, 38., 2001, Las Vegas, USA. **Proceedings...** New York: ACM, 2001. p. 684-689.

ESMERTEC. **Jeode Java Runtime Environment**. Disponível em: <http://www.esmertec.com/products/products_jeode.shtm>. Acesso em: nov. 2003.

GAMMA, E. et al. **Padrões de projeto**: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000. 364 p.

GILL, C.; CYTRON, R.; SCHMIDT, D.C. Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing. **Proceedings of the IEEE**, Piscataway, v.91, n.1, p.183-197, Jan. 2003. Special Issue on Modeling and Design of Embedded Software.

GOOSSENS, K. et al. Networks on Silicon: Combining Best-Effort and Guaranteed Services. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 5., 2002, Paris. **Proceedings...** Washington, DC: IEEE Computer Society, 2002. p. 423-425.

GRÖTKER, T. et al. **System design with SystemC**. Boston: Kluwer Academic Publishers, 2002.

ITO, S.A.; CARRO, L.; JACOBI, R.P. Making Java Work for Microcontroller Applications. **IEEE Design & Test of Computers**, California, v.18, n.5, p. 100-110, Sept./Oct. 2001.

JERRAYA, A.A. et al. Programming models and HW-SW Interfaces Abstraction for Multi-Processor SoC. In: DESIGN AUTOMATION CONFERENCE, DAC, 43., 2006, San Francisco. **Proceedings...** New York: ACM Press, 2006. p. 280-285.

JOHNSON, D.S. Approximation Algorithms for Combinatorial Problems. In: ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING, 5., 1973, Austin, TX, USA. **Proceedings...** New York: ACM, 1973. p.38-49.

KLEFSTAD, R. et al. Towards Highly Configurable Real-Time Object Request Brokers. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, ISORC, 5., 2002, Crystal City, VA, USA. **Proceedings...** Washington, DC: IEEE, 2002. p. 437-447.

KOHOUT, P.; GANESH, B.; JACOB, B. Hardware support for real-time operating systems. In: IEEE/ACM/IFIP INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS, CODES+ISSS, 1., 2003, New Port, CA. **Proceedings...** New York: ACM, 2003. p.45-51.

KON, F. et al. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In: IFIP/ACM INTERNATIONAL CONFERENCE ON DISTRIBUTED SYSTEMS PLATFORMS, MIDDLEWARE, 2., 2000, Palisades, NY. **Proceedings...** Berlin: Springer, 2000. p. 121-143. (Lecture Notes in Computer Science, v.1795).

KRISHNA, A.S. et al. Optimizing the ORB Core to Enhance Real-time CORBA Predictability and Performance. In: DISTRIBUTED OBJECTS AND APPLICATIONS, DOA, 5., 2003, Catania, Italy. **Proceedings...** [S.l.: s.n.], 2003.

KUACHAROEN, P.; SHALAN, M.; MOONEY, V. A configurable hardware scheduler for realtime systems. In: INTERNATIONAL CONFERENCE ON ENGINEERING OF RECONFIGURABLE SYSTEMS AND ALGORITHMS, ERSA, 2003, Las Vegas, USA. **Proceedings...** [S.l.]: CSREA Press, 2003. p. 96-101.

KUNZ, L. **Implementação em hardware de uma API de comunicação para processador FemtoJava**. 2007. 45 f. Trabalho de diplomação (Engenharia de Computação) - Instituto de Informática, UFGRS, Porto Alegre.

LI, S.; SON, S.H.; STANKOVIC, J.A. Event Detection Services Using Data Service Middleware in Distributed Sensor Networks. In: INTERNATIONAL CONFERENCE ON INFORMATION PROCESSING IN SENSOR NETWORKS, IPSN, 2003, Palo Alto, USA. **Proceedings...** [S.l.]: Springer, 2003. p. 502-517. (Lecture Notes in Computer Science, v.2634).

LINDH, L. Fasthard - a fast time deterministic hardware based real-time kernel. In: EUROMICRO WORKSHOP ON REAL-TIME SYSTEMS, 4., 1992. **Proceedings...** [S.l.]: IEEE, 1992. p. 21-25.

MADDEN, S. R. et al. TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks. In: OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, OSDI, 2002, Boston, USA. **Proceedings...** [S.l.: s.n.], 2002.

MACKETANZ, R.; KARL, W. JVX: A Rapid Prototyping System Based on Java and FPGAs. In: HARTENSTEIN, R.W.; KEEVALLIK, A. (Ed.). **Field-Programmable**

Logic And Applications: From FPGAs to Computing Paradigm. Berlin: Springer, 1998. p. 99-108. (Lecture Notes in Computer Science, v.1482).

MARCON, C.M. **Modelos para o mapeamento de aplicações em infra-estruturas de comunicação intrachip**. 2005. 192 f. Tese (Doutorado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

MARTIN, G. Overview of the MPSoC Design Challenge. In: DESIGN AUTOMATION CONFERENCE, DAC, 2006, 43., San Francisco, CA. **Proceedings...** New York, ACM Press, 2006. p.274-279.

MATTOS, J.C.B. et al. Design Space Exploration with Automatic Generation of IP-based Embedded Software. In: IFIP WORKING CONFERENCE ON DISTRIBUTED AND PARALLEL EMBEDDED SYSTEMS, DIPES, 2004, Toulouse. **Proceedings...** Boston: Kluwer Academic Publishers, 2004. p.237-246.

MILLBERG, M. et al. The Nostrum Backbone - a Communication Protocol Stack for Networks on Chip. In: INTERNATIONAL CONFERENCE ON VLSI DESIGN, 17., 2004, Mumbai, India. **Proceedings...** Washington, DC: IEEE Computer Society, 2004. p.693-696.

OLIVEIRA, M. F. S. **Exploração do Espaço de Projeto em Sistemas Embarcados Baseados em Plataforma Através de Estimativas Extraídas de Modelos UML**. 2006. 86 f. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

OMG: Object Management Group. **MinimumCORBA Specification**. 1998. Disponível em: <http://www.omg.org>. Acesso em: nov. 2003.

PAULIN, P.G. et al. Parallel Programming Models for a Multi-Processor SoC Platform Applied to High-Speed Traffic Management. In: IEEE/ACM/IFIP INTERNATIONAL CONFERENCE ON HARDWARE - SOFTWARE CODESIGN AND SYSTEM SYNTHESIS, CODES+ISSS, 2., 2004. **Proceedings...** Washington, DC: IEEE Computer Society, 2004. p. 48-53.

PAULIN, P.G. et al. Distributed Object Models for Multi-Processor SoC's, with Application to Low-power Multimedia Wireless Systems. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 9., 2006, Munich, Germany. **Proceedings...** New York: ACM SIGDA, 2006. v.1, p. 482-487.

PILLAI, P.; SHIN, K.G. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. In: ACM SYMP. ON OPERATING SYSTEMS PRINCIPLES, 18., 2001. **Proceedings...** New York: ACM, 2001. p. 89-102.

ROMAN, M. et al. LegORB and Ubiquitous CORBA. In: IFIP/ACM REFLECTIVE MIDDLEWARE WORKSHOP, 2000, Palisades, NY. **Proceedings...** [S.l.: s.n.], 2000. p. 1-2.

ROMER, K. Programming paradigms and *middleware* for sensor networks. In: GI/ITG WORKSHOP ON SENSOR NETWORKS, 2004, Karlsruhe, Germany. **Proceedings...** [S.l.: s.n.], 2004. p. 49-54.

ROSA JUNIOR, L.S. et al. Scheduling Policy Costs on a Java Microcontroller. In: WORKSHOP ON JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS, JTRES, 2003, Catania, Italy. **Proceedings...** Berlin: Springer, 2003. p. 520-533. (Lecture Notes in Computer Science, v.2889).

SANGIOVANNI-VICENTELLI, A. et al. Benefits and Challenges for Platform Based. In: DESIGN AUTOMATION CONFERENCE, DAC, 41., 2004, San Diego, CA. **Proceedings...** New York: ACM Press, 2004. p. 409-414.

SHEN, C.C. et al. Sensor Information Networking Architecture and Applications. **IEEE Personal Communications**, New York, v.8, n.4, p. 52-59, 2001.

SCHMIDT, D.C. Middleware Techniques and Optimizations for Realtime, Embedded Systems. In: INTERNATIONAL SYMPOSIUM ON SYSTEM SYNTHESIS, 1999, San Jose, CA. **Proceedings...** Washington, DC: IEEE Computer Society, 1999. p. 12-16.

SILBERSCHATZ, A. **Sistemas operacionais com Java**. 6. ed. Rio de Janeiro: Elsevier, 2004. 670 p.

SILVA JÚNIOR, E.T. et al. Design Exploration in Hw/Sw Co-design of Real-Time Object-oriented Embedded Systems: the Scheduler Object. In: INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED REAL-TIME DEPENDABLE SYSTEMS, WORDS, 10., 2005, Sedona, Arizona. **Proceedings...** Los Alamitos: IEEE Computer Society, 2005a. p. 378-385.

SILVA JÚNIOR, E.T. et al. Development of Multithread Real-Time Applications using a Hardware Scheduler. In: IFIP WG 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, VLSI-SoC, 13., 2005, Perth, Australia. **Proceedings...** Perth, Australia: IFIP, 2005b. p. 311-316.

SILVA JÚNIOR, E.T. et al. Java Framework for Distributed Real-Time Embedded Systems. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, ISORC, 9., 2006, Gyeongju, Korea. **Proceedings...** Washington, DC: IEEE Computer Society, 2006a. p. 85-92.

SILVA JÚNIOR, E.T.; WAGNER, F.R.; FREITAS, E.P.; PEREIRA, C.E. Hardware support in a middleware for distributed and real-time embedded applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 19., 2006, Ouro Preto, Brazil. **Proceedings...** New York: ACM Press, 2006b. p. 149-154.

SILVA JÚNIOR, E.T.; WEHRMEISTER, M.A.; WAGNER, F.R.; PEREIRA, C.E. An approach to improve predictability in communication services in distributed real-time embedded systems. In: INTERNATIONAL WORKSHOP ON JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS, JTRES, 5., 2007, Vienna. **Proceedings...** New York: ACM Press, 2007a. p. 121-126.

SILVA JÚNIOR, E.T.; WAGNER, F.R.; FREITAS, E.P.; KUNZ, L.; PEREIRA, C.E. Hardware Support in a Middleware for Distributed and Real-Time Embedded

Applications. **Journal of Integrated Circuits and Systems**, [S.l.], v.2, n.1, p. 37-44, Mar. 2007b.

SILVA JÚNIOR, E.T.; ANDREWS, D.; PEREIRA, C.E.; WAGNER, F.R. **An Infrastructure for Hardware-Software Co-design of Embedded Real-Time Java Applications**. Aceito para o IEEE International Symposium On Object-Oriented Real-Time Distributed Computing, ISORC, 2008, Orlando, USA.

SILVA, G.G.B. **Estudo sobre o impacto da hierarquia de memória em MPSoCs baseados em NoCs**. 2007. Plano de Estudos e Pesquisas. Instituto de Informática, UFRGS, Porto Alegre.

SPURI, M.; BUTAZZO, G. Efficient aperiodic service under earliest *deadline* scheduling. In: IEEE REAL-TIME SYSTEMS SYMPOSIUM, RTSS, 1994, San Juan, Puerto Rico. **Proceedings...** [S.l.]: IEEE Computer Society, 1994. p. 2-11.

STANKOVIC, J.A. et al. Strategic Directions in Real-Time and Embedded Systems. **ACM Computing Surveys**, New York, v.28, n.4, p. 751-763, 1996.

SUN MICROSYSTEMS. **J2ME - Connected, Limited Device Configuration Specification, version 1.0**. 2001. Disponível em: <http://java.sun.com/products/cldc/>. Acesso em: jul. 2006.

SUN MICROSYSTEMS. **jGuru**: Remote Method Invocation (RMI) – Tutorials & Code Camps. 1996. Disponível em: <http://java.sun.com/developer/onlineTraining/rmi/RMI.html>. Acesso em: jul. 2006.

TANENBAUM, A.S. **Computer Networks**. 4th ed. Upper Saddle River: Prentice Hall, 2003. 891 p.

WEHRMEISTER, M.A.; BECKER, L.B.; PEREIRA, C.E. Optimizing Real-Time Embedded Systems Development Using a RTSJ-based API. In: WORKSHOP ON JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS, JTRES, 2., 2004, Agia Napa, Cyprus. **Proceedings...** Berlin: Springer, 2004. p. 292-302. (Lecture Notes in Computer Science, v.3292).

WEHRMEISTER, M.A.; FREITAS, E.P.; PEREIRA, C.E.; RAMMIG, F. **GenERTiCA: A Tool for Code Generation and Aspects Weaving**. Aceito para o IEEE International Symposium On Object-Oriented Real-Time Distributed Computing, ISORC, 2008, Orlando, USA.

WINGARD, D. MicroNetwork-Based Integration of SoCs. In: DESIGN AUTOMATION CONFERENCE, DAC, 38., 2001, Las Vegas, USA. **Proceedings...** New York: ACM Press, 2001. p. 673- 677.

WOLF, W.H. **Computer as components**: principles of embedded computing system design. San Francisco, CA: Morgan Kaufmann, 2001.

WRONSKI, F.; BRIÃO, E.W.; WAGNER, F.R. Evaluating Energy-aware Task Allocation Strategies for MPSoCs. In: IFIP TC-10 WORKING CONFERENCE ON DISTRIBUTED AND PARALLEL EMBEDDED SYSTEMS, DIPES, 5., 2006, Braga.

From Model-Driven Design to Resource Management for Distributed Embedded Systems. New York: Springer, 2006. p. 215-224.

YAU, S.S. et al. Reconfigurable Context-Sensitive Middleware for Pervasive Computing. **IEEE Pervasive Computing**, Los Alamitos, v.1, n.3, p. 33-40, July 2002.

ZEFERINO, C.A.; SUSIN, A.A. SoCIN: A parametric and scalable network-on-chip. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 16., 2003, São Paulo. **Proceedings...** Los Alamitos: IEEE Computer, 2003. p. 169-174.

ZEFERINO, C.A. et al. RASoC: a router soft-core for networks-on-chip. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE, DATE, 7., 2004. **Proceedings...** Washington, DC: IEEE Computer Society, 2004. v.3, p. 198-203.

APÊNDICE A CÓDIGO FONTE DE ALGUMAS APLICAÇÕES

Este apêndice traz o código fonte de algumas aplicações omitidas no texto.

Código da implementação hardware (SystemC) do Filtro FIR com 10 taps.

```
#include "hwtul.h"

void hwtul::execute()
{
   updateInputPorts();
   intrfc2thrd_value = input[0];
   intrfc2thrd_function = input[1];
   intrfc2thrd_rdy2recv = input[2];

   if ( (intrfc2thrd_function == 0) && fntReceived ) { // pronto para receber outra function
      fntReceived = 0;
   }

    if (clk.posedge())
    {
      if ((intrfc2thrd_function == hwtiRESET) && !fntReceived)
      {  // fsm para estado inicial
         fsm_HWTUL = case state_000;
      }
      else if (intrfc2thrd_rdy2recv == 1)
      {
         switch (fsm_HWTUL)
         {
         case state_000:
            if (intrfc2thrd_function == hwtiSTART)
            fsm_HWTUL = state_001;
         break;

         case state_001:
            thrd2intrfc_address = SETOBJbase + hwti0displacement;
            fsm_HWTUL = state_002;
         break;

         case state_002:
            thrd2intrfc_opcode = hwtiGETSTATIC;
            fsm_HWTUL = state_003;
         break;

         case state_003:
            ThisObj = intrfc2thrd_value;
            mem_base = ThisObj + HwRTthreaddisplacement;
            thrd2intrfc_opcode = 0;// clear opcode
            fsm_HWTUL = state_004;
         break;

// Carrega apontador dos coeficientes
         case state_004:
            thrd2intrfc_address = mem_base + 1; // coefsFIR
            fsm_HWTUL = state_005;
```

```
51          break;
52
53          case state_005:
54             thrd2intrfc_opcode = hwtiGETSTATIC;
55             fsm_HWTUL = state_006;
56          break;
57
58          case state_006:
59             coefsFIR = intrfc2thrd_value+1; // o array comeca no endereco sequinte
60             thrd2intrfc_opcode = 0;// clear opcode
61             fsm_HWTUL = state_007;
62          break;
63
64 // Carrega apontador dos valores de entrada
65          case state_007:
66             thrd2intrfc_address = mem_base + 2; // inputFIR
67             fsm_HWTUL = state_008;
68          break;
69
70          case state_008:
71             thrd2intrfc_opcode = hwtiGETSTATIC;
72             fsm_HWTUL = state_009;
73          break;
74
75          case state_009:
76             inputFIR = intrfc2thrd_value+1; // o array comeca no endereco sequinte
77             thrd2intrfc_opcode = 0;// clear opcode
78             fsm_HWTUL = state_010;
79          break;
80
81 // Carrega apontador dos valores de saida (resultados)
82          case state_010:
83             thrd2intrfc_address = mem_base + 3; // outputFIR
84             fsm_HWTUL = state_011;
85          break;
86
87          case state_011:
88             thrd2intrfc_opcode = hwtiGETSTATIC;
89             fsm_HWTUL = state_012;
90          break;
91
92          case state_012:
93             outputFIR = intrfc2thrd_value+1; // o array comeca no endereco sequinte
94             thrd2intrfc_opcode = 0;// clear opcode
95             fsm_HWTUL = state_013;
96          break;
97
98 //     for (ii = 0; ii < 2*NTAPS; ii++) {
99          case state_013:
00             ii = 0;
01             fsm_HWTUL = state_014;
02          break;
03
04          case state_014:
05             if (ii < 2*NTAPS) {
06             fsm_HWTUL = state_015;
07             }
08             else {
09             fsm_HWTUL  = state_033;// estado fora do for - exit
10             }
11          break;
12
13 //     z[state] = *(inputFIR+ii);
14          case state_015:
15             thrd2intrfc_address = inputFIR+ii;
16             fsm_HWTUL = state_016;
17          break;
18
19          case state_016:
20             thrd2intrfc_opcode = hwtiGETSTATIC;
21             fsm_HWTUL = state_017;
22          break;
23
24          case state_017:
```

```
25              z[state] = intrfc2thrd_value;
26              thrd2intrfc_opcode = 0;// clear opcode
27              fsm_HWTUL = state_018;
28            break;
29
30          /* incr state and check for wrap */
31  //      if (++state >= NTAPS) state = 0;
32          /* calc FIR and shift data */
33  //      accum = 0;
34            case state_018:
35               if (++state >= NTAPS) state = 0;
36                accum = 0;
37               fsm_HWTUL = state_019;
38            break;
39
40  //      for (jj = NTAPS - 1; jj >= 0; jj--) {
41            case state_019:
42               jj = NTAPS - 1;
43               fsm_HWTUL = state_020;
44            break;
45
46            case state_020:
47               if (jj >= 0) {
48               fsm_HWTUL = state_021;
49               }
50               else {
51               fsm_HWTUL = state_026;// estado fora do for - outputFIR
52               }
53            break;
54
55  //    accum += *(coefsFIR+jj) * z[state];// *(h+jj) = h[jj]
56            case state_021:
57               thrd2intrfc_address = coefsFIR+jj;
58               fsm_HWTUL = state_022;
59            break;
60
61            case state_022:
62               thrd2intrfc_opcode = hwtiGETSTATIC;
63               fsm_HWTUL = state_023;
64            break;
65
66            case state_023:
67               accum += intrfc2thrd_value * z[state];
68               thrd2intrfc_opcode = 0;// clear opcode
69               fsm_HWTUL = state_024;
70            break;
71
72          /* incr state and check for wrap */
73  //      if (++state >= NTAPS) state = 0;
74            case state_024:
75               if (++state >= NTAPS) state = 0;
76               fsm_HWTUL = state_025;
77            break;
78
79  // fim do for menor
80            case state_025:
81               jj--;
82               fsm_HWTUL = state_020; // estado do teste do for jj
83            break;
84
85  // *(outputFIR+ii) = accum;// *(outputFIR+ii) = outputFIR[ii]
86            case state_026:
87               thrd2intrfc_address = outputFIR+ii;
88               thrd2intrfc_value = accum;
89               fsm_HWTUL = state_027;
90            break;
91
92            case state_027:
93               thrd2intrfc_opcode = hwtiPUTSTATIC;
94               fsm_HWTUL = state_028;
95            break;
96
97            case state_028:
98               thrd2intrfc_opcode = 0;// clear opcode
```

```
99             fsm_HWTUL = state_029;
00          break;
01
02 // waitForNextPeriod
03          case state_029:
04             thrd2intrfc_function = hwtiWAIT_FOR_NEXT_PERIOD;
05             fsm_HWTUL = state_030;
06          break;
07
08          case state_030:
09             thrd2intrfc_opcode = hwtiCALL;
10             fsm_HWTUL = state_031;
11          break;
12
13          case state_031:
14             if (intrfc2thrd_function == hwtiCONTINUE) {
15             thrd2intrfc_opcode = 0;// clear opcode
16             fsm_HWTUL = state_032;
17             }
18          break;
19
20 // fim do for maior
21          case state_032:
22             ii++;
23             fsm_HWTUL = state_014; // estado do teste do for ii
24          break;
25
26 // exit
27          case state_033:
28             thrd2intrfc_function = hwtiEXIT;
29             fsm_HWTUL = state_034;
30          break;
31
32          case state_034:
33             thrd2intrfc_opcode = hwtiCALL;
34             fsm_HWTUL = state_035;
35          break;
36
37          case state_035:
38             thrd2intrfc_opcode = 0;// clear opcode
39             fsm_HWTUL = state_000;
40          break;
41
42          default :
43             fsm_HWTUL = state_000;
44          break;
45
46
47          }
48       } // else.fim (not reset)
49
50
51    }  // if.end
52
53    output[0] = thrd2intrfc_address;
54    output[1] = thrd2intrfc_value;
55    output[2] = thrd2intrfc_function;
56    output[3] = thrd2intrfc_opcode;
57    updateOutputPorts();
58 }
59
60 void hwtul::updateInputPorts()
61 {
62    int i;
63    for (i=0;i<3;i++)
64    {
65       input[i]=inputPort[i].read().to_int();
66    }
67 }
68
69 void hwtul::updateOutputPorts()
70 {
71    int i;
72    for (i=0;i<4;i++)
```

```
73    {
74       outputPort[i]=output[i];
75    }
76 }
```

Código Java do exemplo FIR_1: Lado consumidor dos dados

```
 1 import saito.sashimi.*;
 2 import saito.sashimi.realtime.*;
 3
 4 public class FirTaskset {
 5     public static EDFScheduler sched = new EDFScheduler();
 6     public static Fir10hw hw = new Fir10hw(); // hw
 7     public static Task1 t1 = new Task1(1); // dummy
 8     public static InputDriver input = new InputDriver(7);
 9
10     public static void initSystem() {
11     input.fir = hw;
12
13     Scheduler.setDefaultScheduler(sched);
14
15     hw.addToFeasibility();
16     hw.start();
17     t1.addToFeasibility();
18     t1.start();
19     input.addToFeasibility();
20     input.start();
21
22     sched.setupTimer();
23     idleTask();
24   }
25
26   public static void idleTask()   {
27     while (true)     {
28       FemtoJava.sleep();
29     }
30   }
31 }
```

```
 1 import saito.sashimi.*;
 2 import saito.sashimi.realtime.*;
 3 import saito.sashimi.rmiFJ.*;
 4
 5 public class InputDriver extends RealtimeThread
 6 {
 7     private static final int MY_ADDRESS = 1;
 8     public static final int NTAPS = 10;
 9
10     // ReleaseParameters
11     private static AbsoluteTime m_taskActiveTime = new AbsoluteTime(0,0,0);
12     private static RelativeTime m_taskPrevProcTime = new RelativeTime(0,0,0);
13     protected static RelativeTime period = TimeObjects3._4_ms; // periodo
14     protected static RelativeTime cost = new RelativeTime(0,1,250); // wcet
15     protected static RelativeTime dline = new RelativeTime(0,1,800); // deadline
16     protected static AbsoluteTime m_taskReleaseTime = new AbsoluteTime(0,3,0);
17     protected static PeriodicParameters perpar =
18                         new PeriodicParameters(null, null, period, cost, dline);
19
20     private static int inputArray[] = new int[2*NTAPS];
21     private int inCounter = 0;
22     private int m_TaskID;
23     private int dlineMiss = 0;
24     public Fir10hw fir;
25
26     // para rmi
27     public static RelativeTime _openConRegTimeOut = TimeObjects3._10_ms;
28     public static RelativeTime _openConServTimeOut = new RelativeTime(0,11,0);
```

```
29      public static RelativeTime _msgSendTimeOut = TimeObjects3._10_ms;
30      public static RelativeTime _msgRecvTimeOut = TimeObjects3._10_ms;
31      public static TimeOutParameters _timeout =
32                      new TimeOutParameters(null,             //listenTimeOut
33                                            _openConRegTimeOut,   //openConRegTimeOut
34                                            _openConServTimeOut,  //openConServTimeOut
35                                            _msgSendTimeOut,      //msgSendTimeOut
36                                            _msgRecvTimeOut);    //msgRecvTimeOut
37
38      public static ClientNamingSOCIN naming = new ClientNamingSOCIN(_timeout);
39
40      public static FIRDataStub input;
41      public static final int INPUT_DATA = 0x15;  // código_nome do objeto remoto
42
43      public InputDriver(int TaskID)    {
44          super(null,perpar);
45          m_TaskID = TaskID;
46          m_ResumeTime = m_taskReleaseTime; // setando o t inicial
47          m_ActiveTime = m_taskActiveTime;
48          m_PreviousProcessTime = m_taskPrevProcTime;
49          Naming.setLocalNIAddress(MY_ADDRESS);
50      }
51
52    /** inclui um inteiro no vetor de dados de entrada - input[] */
53    public void addData(int data) {
54        this.inputArray[inCounter] = data;
55        if (inCounter < inputArray.length)
56          inCounter++;
57    }
58
59    public void exceptionTask(){
60      dlineMiss++;
61    }
62
63    protected void initializeStack(){}
64
65    /** mainTask */
66    public void mainTask()  {
67      FemtoJavaIO.write(m_TaskID,10);//for CPU time analysis
68      do {
69          input = (FIRDataStub)naming.lookup(INPUT_DATA);
70      }while (input == null);
71      this.waitForNextPeriod();
72      for(int ii = 0; ii < 2*NTAPS; ii++)  {
73          FemtoJavaIO.write(m_TaskID,10);// for CPU time analysis
74          fir.putValue(input.readData(ii)); // entraga o valor de entrada para o fir
75          this.waitForNextPeriod();
76      }
77      this.finish();
78    }
79  }
```

Código Java do exemplo FIR_1: Lado produtor de dados

```
 1  import saito.sashimi.*;
 2  import saito.sashimi.realtime.*;
 3  import saito.sashimi.rmiFJ.*;
 4
 5  public class ProdTaskset
 6  {
 7      private static final int MY_ADDRESS = 2;
 8
 9      // objetos de timeOut
10      public static RelativeTime _listenTimeOut = TimeObjects2._10_ms; //
11      public static RelativeTime _openConRegTimeOut = TimeObjects2._10_ms;
12      public static RelativeTime _msgSendTimeOut = TimeObjects2._10_ms;
13      public static RelativeTime _msgRecvTimeOut = TimeObjects2._10_ms;
14      public static TimeOutParameters _timeOut =
15                      new TimeOutParameters(_listenTimeOut,   //listenTimeOut
16                              _openConRegTimeOut,   //openConRegTimeOut
```

```
                        null,             //openConServTimeOut
                        _msgSendTimeOut,      //msgSendTimeOut
                        _msgRecvTimeOut);    //msgRecvTimeOut

    // ReleaseParameter para o ConnectionHandler
    public static PeriodicParameters threadParameter =
                  new PeriodicParameters(null, //hrtimeStart
                      null, //hrtimeEnd
                      TimeObjects2._4_ms, //rtimePeriod
                      TimeObjects2._500_us,  //rtimeCost
                      TimeObjects2._1_5_ms);//rtimeDeadline

    public static final int INPUT_DATA = 0x15;  // código_nome do objeto remoto

    public static DataProvider firInput =
                         new DataProvider(_timeOut, threadParameter);

    public static SkelFactory skelFactory = new SkelFactory(firInput);

    public static EDFScheduler sched = new EDFScheduler();
    public static Task1 t1 = new Task1(0x11); // dummy

    public static void initSystem() {

       Scheduler.setDefaultScheduler(sched);
       Naming.setLocalNIAddress(MY_ADDRESS);

       t1.addToFeasibility();
       t1.start();

       firInput.bind(INPUT_DATA);

       sched.setupTimer();
       idleTask();
    }

    public static void idleTask()   {
     while (true)     {
       FemtoJava.sleep();
     }
    }
}
```

```
import saito.sashimi.*;
import saito.sashimi.realtime.*;
import saito.sashimi.rmiFJ.*;

public class DataProvider extends Remote_object {

    private static Arguments arg = new Arguments();

    public DataProvider(TimeOutParameters t, PeriodicParameters p) {
        super(t, p);
    }

    public int readData(int pos) {
        FemtoJavaIO.write(0x40,10);// for CPU time analysis
        return arg.getInput(pos);
    }

    public void writeData(int pos, int value) {
        FemtoJavaIO.write(0x41,10);// for CPU time analysis
        arg.putOutput(pos, value);
    }

}
```

APÊNDICE B CLASSES E MÉTODOS DA APICOM

Este apêndice traz a listagem das classes da API de comunicação acompanhadas de seus respectivos métodos.

Package saito.sashimi.ApiCom

Class Summary	
DataLink	
DataLinkCan	
DataLinkSOCIN	
Message	
Network	
Pack	
PackEvent	
PackHandler	
Transport	
TransportConnection	

saito.sashimi.ApiCom

Class **DataLink**

```
java.lang.Object
  └ saito.sashimi.ApiCom.DataLink
```

---

public abstract class **DataLink**
extends java.lang.Object

Method Summary

protected abstract int	**addressLength**()
abstract int	**getPackLen**()
protected abstract int	**portLength**()
protected abstract void	**receivePack**()
abstract void	**resetEvent**()
protected abstract boolean	**sendPack**(saito.sashimi.ApiCom.Pack op, saito.sashimi.realtime.RelativeTime timeOut)
abstract void	**setEvent**()
abstract void	**setLocalMac**()
void	**setNetwork**(saito.sashimi.ApiCom.Network n)

saito.sashimi.ApiCom

Class **DataLinkSOCIN**

```
java.lang.Object
  └saito.sashimi.ApiCom.DataLink
      └saito.sashimi.ApiCom.DataLinkSOCIN
```

All Implemented Interfaces:

saito.sashimi.IntrInterface

---

public class **DataLinkSOCIN**
extends saito.sashimi.ApiCom.DataLink
implements saito.sashimi.IntrInterface

Method Summary

protected int	**addressLength**()
int	**getPackLen**()
void	**int0Method**()
void	**int1Method**()
protected int	**portLength**()
protected void	**receivePack**
void	**resetEvent**()
protected boolean	**sendPack**(saito.sashimi.ApiCom.Pack op, saito.sashimi.realtime.RelativeTime timeOut)
void	**setEvent**()
void	**setLocalMac**()
void	**spiMethod**()

Methods inherited from class saito.sashimi.ApiCom.DataLink
setNetwork

saito.sashimi.ApiCom

Class **Message**

```
java.lang.Object
  └ saito.sashimi.ApiCom.Message
```

---

public class **Message**
extends java.lang.Object

Method Summary	
boolean	**addByte**(byte b)
byte	**getByteAtMsg**(int pos)
byte[]	**getMsg**()
int	**getMsgLen**()
int	**getNrBytes**()
void	**setMsg**(Message m, int len)
void	**setNrBytes**(int n)

saito.sashimi.ApiCom

Class **Network**

```
java.lang.Object
  └ saito.sashimi.ApiCom.Network
```

---

public class **Network**
extends java.lang.Object

Method Summary

protected int	**addressLength**()
int	**getLocalLogicAddr**()
int	**getPackLen**()
protected int	**portLength**()
void	**pushPack**(saito.sashimi.ApiCom.Pack p)
boolean	**sendPack**(saito.sashimi.ApiCom.Pack p, saito.sashimi.realtime.RelativeTime tim)
void	**setBroadcast**(boolean b)
void	**setDataLink**(saito.sashimi.ApiCom.DataLink dl)
void	**setEventStatus**(boolean stat)
void	**setLocalLogicAddr**(int addr)
void	**setTransport**(saito.sashimi.ApiCom.Transport t)

saito.sashimi.ApiCom

Class **Pack**

```
java.lang.Object
  └ saito.sashimi.ApiCom.Pack
```

---

public class **Pack**
extends java.lang.Object

Method Summary	
boolean	**addByte**(byte b)
byte	**getByteDataAt**(int i)
byte[]	**getData**()
int	**getFrag**()
int	**getLogicAddr**()
int	**getMacSource**()
int	**getNrByte**()
int	**getPort**()
int	**getPosition**()
int	**getPriority**()
int	**getType**()
boolean	**isFragment**()
void	**setData**(byte[] dt)
void	**setFrag**(int f)
void	**setLogicAddr**(int la)
void	**setMacSource**(int m)
void	**setNrByte**(int n)
void	**setPort**(int p)
void	**setPriority**(int p)
void	**setType**(int tp)

saito.sashimi.ApiCom

Class **PackEvent**

```
java.lang.Object
  └saito.sashimi.realtime.AsyncEvent
      └saito.sashimi.ApiCom.PackEvent
```

---

public class **PackEvent**
extends saito.sashimi.realtime.AsyncEvent

Methods inherited from class saito.sashimi.realtime.AsyncEvent
addHandler, fire, handledBy, removeHandler, setHandler

saito.sashimi.ApiCom

Class **PackHandler**

```
java.lang.Object
  └saito.sashimi.realtime.AsyncEventHandler
      └saito.sashimi.ApiCom.PackHandler
```

---

public class **PackHandler**
extends saito.sashimi.realtime.AsyncEventHandler

Method Summary

protected void	**handleAsyncEvent**()
protected void	**init**(saito.sashimi.ApiCom.DataLink dlink)

Methods inherited from class saito.sashimi.realtime.AsyncEventHandler
attachAsyncEvent, deattachAsyncEvent, hasAsyncEventAttached

saito.sashimi.ApiCom

Class **Transport**

```
java.lang.Object
  └ saito.sashimi.ApiCom.Transport
```

public class **Transport**
extends java.lang.Object

Constructor Summary

Transport()
Transport(saito.sashimi.ApiCom.Network net, saito.sashimi.ApiCom.DataLink dLink)
Transport(saito.sashimi.ApiCom.Network net, saito.sashimi.ApiCom.DataLink dLink, int address)

Method Summary

boolean	**closeConnection**(int connectionRef, saito.sashimi.realtime.RelativeTime timeOut)
boolean	**connectionExists**(int connectionRef)
int	**getAddrLength**()
int	**getConnection**(int port, int addr)
int	**getLocalLogicAddr**()
int	**getMaxAddressNumber**()
int	**getMaxPortNumber**()
int	**getPortLength**()
int	**getPortLogicAddr**(int refConnection)
void	**initTransp**()
void	**initTransp**(saito.sashimi.ApiCom.Network net, saito.sashimi.ApiCom.DataLink dLink, int address)
int	**listen**(int port, saito.sashimi.realtime.RelativeTime timeOut)
boolean	**messageBroadcastReady**()
boolean	**messageReady**(int con)
int	**openConnection**(int logicAddr, int destPort, saito.sashimi.realtime.RelativeTime timeOut)
int	**openConnection**(int logicAddr, int destPort, saito.sashimi.realtime.RelativeTime timeOut, int priority)
int	**portLogicAddr2Addr**(int portAddr)
int	**portLogicAddr2Port**(int portAddr)
boolean	**portUsed**(int port)

boolean	**receiveMsg**(saito.sashimi.ApiCom.Message message, int refConnection)
boolean	**receiveMsgBroadcast**(saito.sashimi.ApiCom.Message message)
boolean	**sendBroadcast**(saito.sashimi.ApiCom.Message msg, int maxArriveTime, saito.sashimi.realtime.RelativeTime timeOut)
boolean	**sendMsg**(int connectNum, saito.sashimi.ApiCom.Message msg, int maxArriveTime, saito.sashimi.realtime.RelativeTime timeOut)
boolean	**sendMsg**(int connectNum, saito.sashimi.ApiCom.Message msg, saito.sashimi.realtime.RelativeTime timeOut)
void	**setConnection**(int addr, int port, int pos)
void	**setEventStatus**(boolean stat)
void	**setLocalLogicAddr**(int laddr)
void	**setMsgRdyEvent**(saito.sashimi.realtime.AsyncEvent event, int port)
void	**setNetwork**(saito.sashimi.ApiCom.Network n)
void	**setResquestEvent**(saito.sashimi.realtime.AsyncEvent event, int port)
void	**setupListen**(int port)
void	**solvePack**(saito.sashimi.ApiCom.Pack inP)
void	**solveReply**()
void	**subscribe**()
void	**unSubscribe**()

saito.sashimi.ApiCom

Class **TransportConnection**

```
java.lang.Object
  └ saito.sashimi.ApiCom.TransportConnection
```

---

public class **TransportConnection**
extends java.lang.Object

Method Summary

int	**getConnectionType**()
int	**getLogicAddr**()
static int	**getMaxPriority**()
int	**getPort**()
int	**getPortLogicAddr**()
int	**getPriority**()
void	**setConnectionType**(int conType)
void	**setLogicAddr**(int la)
void	**setPort**(int p)
void	**setPriority**(int p)

APÊNDICE C QUANTIDADE DE CÓDIGO JAVA

Este apêndice traz o tamanho da maior parte do código Java escrito durante este trabalho. São dadas tabelas contendo as classes dos pacotes, seu número de métodos e quantidade de linhas de código.

Pacote: APICOM

	Classe	Número de métodos	Número de linhas
	DataLink	09	84
	DataLinkCan	11	395
	DataLinkSOCIN	11	443
	Message	07	75
	Network	11	110
	Pack	20	189
	PackEvent	0	21
	PackHandler	02	41
	Transport	35	1366
	TransportConnection	10	83
Total		116	2807

Pacote: Realtime

	Classe	Número de métodos	Número de linhas
	CC_EDFScheduler	11	382
	HwRealtimeThread	5	283
Total		16	665

Pacote: MoveThrd

	Classe	Número de métodos	Número de linhas
	MoveThread	7	594
	MvThdEvent	0	23
	MvThreadHandler	2	45
Total		9	662

Pacote: RMI-FJ

	Classe	Número de métodos	Número de linhas
	ClientNaming	2	142
	ClientNamingCAN	0	35
	ClientNamingSOCIN	0	38
	ConnectionHandler	9	222
	Naming	4	79
	ObjectIO	7	88
	ParamCarrier	2	35
	Registry	5	242
	Remote_object	6	77
	ServerNaming	9	262
	ServerRmiFJCAN	3	49
	ServerRmiFJSOCIN	3	62
	SkelAppend	3	42
	Skeleton	9	132
	Stub	3	84
	RealTimeParameters	5	75
Total		70	1664

Total de classes: 31

Total de métodos: 211

Total de linhas: 5798