UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

DAVID CEMIN

ARQUITETURA DE AGENTES MÓVEIS RECONFIGURÁVEIS PARA REDES DE SENSORES SEM FIO

Porto Alegre
2012

DAVID CEMIN

ARQUITETURA DE AGENTES MÓVEIS RECONFIGURÁVEIS PARA REDES DE SENSORES SEM FIO

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Rio Grande do Sul como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.
Área de concentração: Controle e Automação

ORIENTADOR: Prof. Dr. Carlos Eduardo Pereira

CO-ORIENTADOR: Prof. Dr. Marcelo Götz

Porto Alegre
2012

DAVID CEMIN

ARQUITETURA DE AGENTES MÓVEIS RECONFIGURÁVEIS PARA REDES DE SENSORES SEM FIO

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____
Prof. Dr. Carlos Eduardo Pereira, Doutor pela Universidade de Stuttgart – Stuttgart, Alemanha

Banca Examinadora:

Prof. Dr. Ricardo Pezzuol Jacobi, UFRGS
Doutor pela Université Catholique de Louvain, Louvain-la-Neuve – Bélgica

Prof. Dr. Altamiro Amadeu Susin, UFRGS
Doutor pelo Institut National Polytechnique de Grenoble (INPG) – Grenoble, França

Prof. Dr. Marcelo Soares Lubaszewski, UFRGS
Doutor pelo Institut National Polytechnique de Grenoble (INPG) – Grenoble, França

Coordenador do PPGEE: _____
Prof. Dr. João Manoel Gomes da Silva Júnior

Porto Alegre, Setembro de 2012.

DEDICATÓRIA

Dedico este trabalho aos meus pais, Claudir Francisco Cemin e Izolma Cemin, e à minha irmã Aline Cemin.

AGRADECIMENTOS

Agradeço à minha família por todo o sacrifício que sempre fizeram para que eu pudesse dar continuidade aos meus estudos.

À Universidade Federal do Rio Grande do Sul por me prover ensino público e de qualidade durante a minha graduação e também durante o mestrado.

Ao Programa de Pós-Graduação em Engenharia Elétrica pela oportunidade de desenvolver este trabalho.

Ao professor Carlos Eduardo Pereira pela disponibilidade de me orientar durante este período e também por todas as oportunidades criadas.

Ao professor Marcelo Götz por todas as horas de disposição que teve durante este trabalho e por todas as correções, sugestões, críticas que sem elas este trabalho talvez não tivesse chegado até aqui.

Ao colega Rodrigo Allgayer pelo apoio que ajudou muito este trabalho dar os passos iniciais.

Aos colegas Marco Aurélio Lisboa Silveira e Brás Felipe Patta Rosa pelas horas de conversa e por todo apoio técnico dado a este trabalho.

Aos colegas de mestrado pelas horas de estudo e descontração.

Ao Sifu Sérgio Queiroz por me ensinar a ser paciente e humilde galgando um a um os degraus do conhecimento.

À minha namorada Luciana pelo carinho e compreensão.

RESUMO

Redes de sensores sem fio (RSSFs) heterogêneas podem combinar nós estáticos e nós móveis. Os nós móveis podem ainda conter um *hardware* mais sofisticado quando comparado aos nós estáticos. Veículos aéreos não tripulados (VANTs) podem conferir mobilidade ao nó sensor aumentando a flexibilidade da RSSF onde ele está inserido. Tanto os VANTs quanto outros nós sensores comuns podem conter uma arquitetura de *hardware* reconfigurável, como por exemplo um FPGA, e com isso adquirir um poder computacional diferenciado. RSSFs propiciam um grande e interessante espectro de aplicações possíveis, tais como vigilância aérea, suporte à segurança pública entre outros. As RSSFs podem ser configuradas através do uso de agentes móveis, que são capazes de migrar carregando as tarefas que serão executadas nos nós.

Neste cenário, este trabalho descreve uma arquitetura de agentes reconfiguráveis para redes de sensores sem fio. Os agentes são capazes de serem executados como um agente puramente em *software*, ou também como um agente em *hardware*, dependendo do ambiente de execução disponível e do *design* escolhido. A arquitetura proposta para o agente reconfigurável apresenta a transparência necessária ao agente para que o resto do sistema não perceba a natureza dos agentes que estão sendo executados na plataforma. Além disso, a arquitetura permite a migração dinâmica de agentes que reconfiguram o sistema também de uma maneira transparente. São mostrados exemplos de casos de uso que demostram a viabilidade de uso da arquitetura proposta e este trabalho ainda mostra a análise realizada sobre estas plataformas.

Palavras-chave: Sistemas embarcados, sistemas multi-agente, arquiteturas reconfiguráveis, redes de sensores sem fio.

ABSTRACT

Heterogeneous wireless sensor networks (WSN) can combine static nodes and mobile nodes. Theses mobile nodes may contain a sophisticated hardware when compared to static nodes. Unmanned aircraft vehicles (UAVs) can confer mobility to the sensor node increasing the flexibility of the WSN to where it is inserted. UAVs as well as other common sensor nodes can have a reconfigurable hardware architecture, as, for example, an FPGA and with this achieve a differentiated computational power. WSNs enable a vast and interesting spectrum of possible applications, like aerial surveillance, public security support, among others. The WSNs can be configured by the use of mobile agents, which are capable of migrating among the nodes, carrying the tasks to be executed and that will be instantiated on a given node.

In this scenario, this work describes an architecture of reconfigurable agents to wireless sensor networks. The agents can be implemented purely in software or as a hardware agent, depending on the available execution environment and on the chosen design. The proposed architecture presents the necessary transparency to the agent so that the rest of the system is not aware of the nature of the agents that are implemented on the platform. Furthermore, the architecture enables dynamic migration of agents that reconfigure the system in a transparent way as well. In this work, use cases examples that demonstrate the feasibility of using the proposed architecture are shown, as well as the analysis performed on these platforms.

Keywords: Embedded systems, multi-agent systems, reconfigurable architectures, wireless sensor networks.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ACL	Agent Communication Language
AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
BRAM	Block Ram
CMOS	Complementary Metal Oxide Semiconductor
CPLD	Complex Programmable Logic Device
DDR	Double Data Rate
EPLD	Erasable Programmable Logic Device
EPROM	Erasable Programmable Read-Only Memory
EEPROM	Electrically Erasable Programmable Read-Only Memory
FIPA	Foundation for Intelligent Physical Agents
GPIO	General Purpose Input Output
HTTP	Hypertext Transfer Protocol
IIOP	Internet Inter-ORB Protocol
JTAG	Joint Test Action Group
JVM	Java Virtual Machine
LUT	Look-up Table
MAC	Medium Access Control
MCU	Micro Controller Unity
MIPS	Million Instructions Per Second
MMU	Memory Management Unit
OSI	Open System Interconnection
PLD	Programmable Logic Device
PROM	Programmable Read Only Memory
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing

RLDRAM	Reduced Latency Dynamic Random Access Memory
RSSF	Rede de Sensores Sem Fio
RTOS	Real-Time Operating System
RLDRAM	Reduced-Latency Dynamic Random Mccess Memory
RSSF	Redes de sensores sem fio
SDRAM	Synchronous Dynamic Random Access Memory
SPI	Serial Peripheral Interface bus
SPLD	Simple Programmable Logic Device
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VANT	Veículo aéreo não-tripulado
WAP	Wireless Application Protocol
WSN	Wireless Sensor Network

SUMÁRIO

1 INTRODUÇÃO

1.1 Motivação do Trabalho

Redes de sensores sem fio têm sido utilizadas em inúmeras aplicações tanto com objetivo de prover informações a sistemas computacionais que auxiliam o homem na tomada de decisão (KUORILEHTO; HANNIKAINEN; HAMALAINEN, 2005), quanto para prover aos próprios sistemas computacionais informações para que estes atuem de maneira autônoma de forma a realizar atividades de interesse de seus usuários, como por exemplo, em serviços de localização utilizados em sistemas ubíquos e em computação pervasiva (HIGHTOWER; BORRIELLO, 2003).

Uma maneira de aumentar as possibilidades de uso das redes de sensores sem fio (RSSFs) é fazer com que elas sejam capazes de fornecer informações mais ricas utilizando-se sensores com capacidades e características heterogêneas. Estas características vão desde o tipo de dado que o sensor é capaz de fornecer, passando pelas diferentes plataformas computacionais que suportam a sua atividade, até chegar a sua capacidade de mobilidade.

Uma rede de sensores consiste em múltiplos nós sensores trocando dados normalmente em um ambiente sem fio. Cada sensor pode coletar, processar e transferir informações do ambiente (AKYILDIZ et al., 2002) sem precisar mandar os dados coletados para outro nó que concentre a fusão dos dados. Cada sensor pode fazer seu processamento local e enviar os dados já processados, diminuindo o tráfego e também a energia gasta pela rede.

Existem muitas áreas que podem se beneficiar pelo uso de redes de sensores, como por exemplo automação residencial (JIN et al., 2008) com sensores espalhados pelo ambiente monitorando temperatura da casa, ou ainda monitorando as condições climáticas. Outro exemplo de uso reside em aplicações militares (LEE et al., 2009) onde se faz uso de redes de sensores para monitoramento de ambiente e proteção de tropas. Tem-se também utilização de redes de sensores em campos de engenharia biomédica (AHMED; RAJA, 2009) no monitoramento de pacientes em enfermarias.

Devido à descentralização do processamento que é intrínseco à topologia de redes de sensores, inúmeras aplicações podem se beneficiar da utilização deste tipo de arquitetura, tais como:

- Monitoramento de áreas de desastre.
- Monitoramento de afluentes em rios.
- Sistemas de vigilância.

- Monitoramento de qualidade de produtos.

Devido ao tamanho dos nós sensores, eles podem ser facilmente acoplados à veículos aéreos não-tripulados (VANTs) e com isso adicionar mobilidade física aos nós sensores. Este tipo de abordagem pode ser utilizado também para o monitoramento de regiões de interesse como por exemplo regiões de desastres naturais como queimadas ou enchentes. O uso de VANTs em RSSFs aumenta a flexibilidade da rede por permitir que os sensores se desloquem até a localidade que se deseja monitorar.

A eficiência energética é um desafio a ser superado em arquiteturas de redes de sensores. Muitas vezes não se têm acesso aos nós e então torna-se inviável a substituição da bateria dos mesmos. Alguns estudos (LIU et al., 2007) usam protocolos de roteamentos mais eficientes para tentar diminuir o consumo de energia na transmissão de dados entre os nós, enquanto outros (PARK; DING; BYON, 2008) procuram diminuir o tamanho dos pacotes que são transmitidos entre os nós.

Para permitir uma melhor disseminação de dados em redes de sensores sem fio, existem propostas de utilização de agentes móveis (CHEN et al., 2006) para realizar estas tarefas. Agentes são neste contexto uma ou mais entidades responsáveis por executar as atividades necessárias no nó sensor. Um agente é capaz de tomar decisões baseadas nas características do nó sensor e na RSSF onde ele está inserido.

Nós sensores têm hardware heterogêneo e com capacidade de processamento e memória limitados. Em uma topologia comum de RSSF, é normal se ter nós fazendo um trabalho colaborativo onde diferentes nós fazem partes diferentes deste trabalho. A utilização da migração de agentes que carregam diferentes tarefas para superar os desafios relacionados à limitação de hardware pode ser vista como uma abordagem para superar as dificuldades impostas pelas RSSFs.

Um agente móvel é capaz de se mover entre diferentes nós carregando as tarefas que serão executadas nestes nós. Uma RSSF pode conter diversos agentes, sejam eles móveis ou não, e este sistema como um todo é chamado de sistema multi-agente (SMA). As RSSFs aliada aos SMAs podem conter requisitos que variam ao longo do tempo devido à diversas características que podem ou não estar associadas ao sistema, como por exemplo desgaste de dispositivos ou ainda alterações climáticas. O uso de agentes em sistemas que mudam suas características dinamicamente traz consigo o benefício da reconfiguração do sistema em tempo de execução, visto que os agentes podem carregar novas tarefas ao sistema fazendo com que ele seja reprogramado, o que aumenta a flexibilidade da RSSF onde eles estão inseridos.

A reconfiguração do sistema não está limitada somente às tarefas que estão sendo executadas a nível de *software* no processador principal do sistema. Este sistema pode conter algum tipo de *hardware* reconfigurável, como por exemplo um *Field Programmable Gate Array* (FPGA). O uso de FPGAs aumenta a flexibilidade dos sistemas pois permite a reprogramação da arquitetura de *hardware* permitindo que o sistema seja reconfigurado quase que por completo. A reconfiguração de *hardware* em RSSF permite que a plataforma não fique presa somente a uma aplicação, permitindo o uso de plataformas mais genéricas que atendem um maior número de aplicações (ALLGAYER, 2009).

O uso de arquiteturas reconfiguráveis podem contribuir para o desenvolvimento de SMAs, visto que os agentes podem se beneficiar das características oferecidas em *hardware* para reprogramar o sistema. Agentes podem ser implementados desta forma diretamente em *hardware* (MENG, 2006) utilizando-se um dispositivo reprogramável, como por exemplo um FPGA. A reprogramação dinâmica destes dispo-

sitivos faz com que o uso de agentes não esteja limitado somente ao processador principal do sistema.

1.2 Objetivos

Este trabalho tem como objetivo criar uma arquitetura que suporte a migração de agentes entre plataformas de *software* e de *hardware*. Aliada à migração, os agentes devem ser capazes de se comunicar de forma transparente independentemente da forma como estão implementados. Desta forma, a arquitetura é reconfigurável, pois os agentes são capazes de migrar entre os nós alterando-se a configuração tanto de *software* quanto de *hardware*. A arquitetura deve ser ainda compatível com outras arquiteturas de SMAs existentes, permitindo o interfaceamento entre elas.

Como será visto ao longo deste trabalho, o SMA será desenvolvido utilizando-se como base um *framework* já existente que permite o rápido desenvolvimento de SMAs mantendo a compatibilidade com outros sistemas. Para isso é utilizado o Jade (TILAB, 2012) como arquitetura de suporte aos agentes. Os agentes em *hardware* serão implementados em FPGAs e serão descritos através de uma linguagem de descrição de *hardware* de alto nível, como por exemplo VHDL. Será criada uma camada de interfaceamento dos agentes de *software* e de *hardware* que é chamada de Wrapper, pois encapsula as requisições de ambos tipos de agentes.

O sistema será validado através de experimentos implementados diretamente sobre uma plataforma de testes que contém dispositivos análogos ao que se encontra em nós sensores comerciais. Será implementada toda a arquitetura de *software* dos agentes bem como a arquitetura de *hardware* desenvolvida no FPGA. A arquitetura será feita de forma modular, permitindo a sua expansão e uso em outros projetos que tenham como base o uso de SMAs. Como resultado espera-se ter uma plataforma de agentes que suporte comunicação entre os diferentes tipos de agentes e ainda permitindo a migração destes agentes, o que caracteriza a arquitetura heterogênea de agentes móveis.

1.3 Organização

Este trabalho está organizado da seguinte maneira. No Capítulo 2 são apresentados os conceitos relativos às RSSFs, discutindo-se questões de arquitetura da rede e do nó sensor. No Capítulo 3 são apresentados os conceitos de arquiteturas reconfiguráveis onde são discutidas arquiteturas de *software* e *hardware* que suportam reconfiguração. No Capítulo 4 são apresentados os conceitos de agentes e SMA, bem como é apresentada a plataforma do Jade, que é utilizada neste trabalho. No Capítulo 5 são discutidos os trabalhos relacionados a este e que serviram como base para a criação da arquitetura aqui proposta. No Capítulo 6 é apresentada a proposta de arquitetura deste projeto enquanto que no Capítulo 7 é discutida a sua implementação. O Capítulo 8 descreve os estudos de caso utilizados na validação da proposta deste trabalho. O Capítulo 9 analisa os resultados obtidos neste trabalho e descreve brevemente os trabalhos futuros. Finalmente os anexos e apêndices mostram mais com mais detalhes aspectos que foram deixados de fora do texto.

2 REDES DE SENSORES SEM FIO

O uso de sensores ditos inteligentes tem sido estimulado pelo avanço tecnológico nas áreas de comunicação sem fio, sensoriamento e também na arquitetura de microprocessadores. Atualmente é comum se ter diferentes tipos de sensores integrados em um único dispositivo, o que possibilita o seu uso em equipamentos embarcados com grande facilidade. O baixo custo destes equipamentos e a difusão da comunicação sem fio propiciam o uso de redes de sensores sem fio em regiões que não se conseguiria alcançar com um processamento centralizado devido ao custo e tamanho dos equipamentos (AKYILDIZ et al., 2002). A Figura 1 mostra a coleta de dados feita por um gerenciador remoto de uma rede de sensores sem fio.

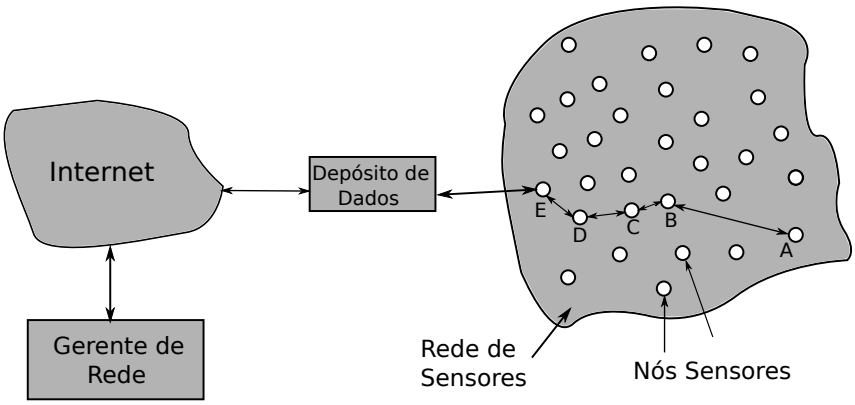

Figura 1: Rede de sensores típica (AKYILDIZ et al., 2002).

Os sensores são densamente distribuídos na área em que se pretende observar um determinado fenômeno. Estes dispositivos geralmente são pequenos e baratos, para que possam ser distribuídos em larga escala, e consequentemente seus recursos em termos de energia, memória, processamento e largura de banda são limitados. Os nós são equipados com dispositivos de sensoriamento, como por exemplo sensores de pressão e temperatura, acelerômetros, magnetômetros, câmeras, microfones, etc (NEWMAN; KEMP, 2007). Estes sensores são usados para monitoramento de condições em diferentes localidades como por exemplo temperatura, umidade, movimentação de veículos, condições atmosféricas, pressão, níveis de ruído, presença ou falta de algum objeto, velocidade ou ainda níveis de *stress* mecânico em alguns objetos.

2.1 Arquitetura de Hardware de um Nó Sensor

Uma arquitetura típica é composta por processador, memória, transmissor/receptor de radiofrequência, uma fonte de energia e sensores. Esta estrutura é mostrada na Figura 2, onde também são mostradas as ligações entre os dispositivos.

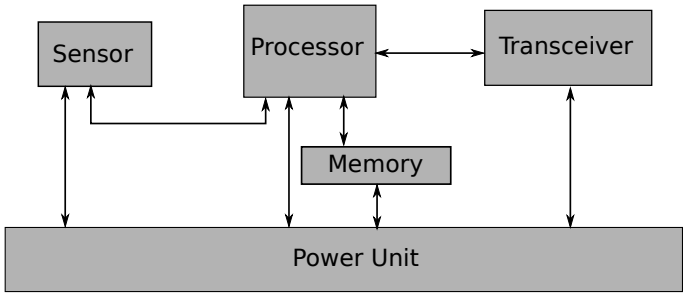

Figura 2: Arquitetura típica de um nó sensor (AKYILDIZ et al., 2002).

2.1.1 Fonte de Energia

Atualmente a fonte de energia é o principal fator de decisão para o tamanho e vida útil de um nó sensor (AKYILDIZ et al., 2002). Similar ao que acontece com telefones móveis, a fonte de energia é geralmente o maior componente encontrado em um nó sensor. Existem soluções que tentam justamente resolver este problema fazendo um nó extremamente pequeno cuja fonte de energia provém de um *link* a *laser*. Este é o caso do *Smart Dust* (O'BRIEN et al., 2009). Este tipo de abordagem pode ser uma solução para diversas topologias de redes de sensores onde se tem linha de visada entre os nós.

Um dos gargalos da miniaturização e do consumo de energia atualmente são os transmissores de rádio, e a redução de consumo dos mesmo requer otimizações através de todas as camadas de comunicação. Diferentes modulações também contribuem para um menor consumo do transmissor e há diversas pesquisas na área de transmissores de rádio que buscam o desenvolvimento de rádios que tenham dimensões reduzidas e apresentem baixo consumo de energia (ENZ; SCOLARI; YODPRASIT, 2005).

Outros fatores também podem afetar no consumo do nó sensor é quantidade de acessos à memoria externa ao processador. Diminuir o número de instruções gastas para acessar a memória e também diminuir o tempo gasto acessando a mesma podem contribuir para a diminuição do consumo de energia. Isso pode ser feito utilizando-se processadores de baixa potência e memórias de baixa latência.

2.1.2 Comunicação Sem Fio

A comunicação sem fio é um dos elementos mais importantes em um nó sensor devido ao fato de ser a parte que geralmente mais consome energia no módulo. Existem projetos tais como o *Pico Radio* que se propõem a desenvolver uma arquitetura de baixo consumo para rádios utilizados em nós sensores. É importante ressaltar que o protocolo de comunicação e o roteamento de pacotes na rede têm grande influência no consumo do módulo, sendo assim estas áreas se tornam também objetos de interesse na pesquisa sobre a diminuição do consumo de energia dos nós sensores.

Os primeiros nós sensores usavam na sua maioria o transmissor *RFM TR1000*

(RFM, 2011), como por exemplo o *Mica Mote* (HILL; CULLER, 2002), devido principalmente ao baixo consumo e tamanho. Novas plataformas como por exemplo a *Mica2* (XBOW, 2011) são baseadas no *Chipcon CC1000* (CHIPCON, 2011) pois fornece modulação FSK mais confiável, frequência de modulação selecionável e ainda uma arquitetura de baixo consumo. Existem diversos outros plataformas que se baseiam em transmissores sobre a pilha *802.15.4*, que é o caso do *BTNode* (ZURICH, 2011) que utiliza *bluetooth*

Desde a introdução do padrão IEEE 802.15.4 para RSSFs, grande parte das novas plataformas está utilizando o *transceiver* sem fio CC2420 da Chipcon (AKYILDIZ et al., 2002), o qual é um dos primeiros *chipsets* compatíveis com o padrão IEEE 802.15.4 (YANG, 2006). Embora o consumo de corrente do CC2420 (19.7 mA) seja maior do que o CC1000 (7.4 mA), o CC2420 consegue taxas mais altas de até 250 Kbps, o que é 6.5 vezes mais alta que a do CC1000 (38.4 Kbps). O CC2420 ainda incorpora encriptação em hardware AES-128, e também o padrão IEEE 802.15.4 MAC o que permite que o próprio transmissor atue como um coprocessador de pacotes, evitando sobrecarregar o processador principal com o processamento de pacotes. Comparado ao CC1000, onde o microprocessador tem que processar todos os pacotes da camada MAC, o CC2420 reduz significativamente o trabalho computacional do ponto de vista do processador, e desta forma melhorando a performance como um todo do nó sensor. Várias plataformas utilizam este *transceiver*, entre elas podemos citar o Telos, Tmote sky, MicaZ, Pluto, iMote2, Sun Spot e o BSN *node*.

2.1.3 Memória e Armazenamento

Nós sensores tipicamente apresentam pouca ou nenhuma memória volátil, então o armazenamento de informações é feito tipicamente em uma memória não volátil, como por exemplo em uma EEPROM. Este tipo de abordagem aumenta entretanto a latência de acesso a um dado que deve ser lido ou escrito em uma memória externa ao processador. Idealmente pode-se ter uma memória RAM para armazenamento de informações do processador, reduzindo-se assim o tempo de processamento e também a energia gasta com requisições de acesso à memória.

Nós sensores como por exemplo o *Mica Mote* (XBOW, 2011) ou o *Sunspot* (SUNSPOT, 2011) utilizam memória RAM, entretanto é uma quantidade relativamente baixa - 4 KB e 512 KB respectivamente. Nós sensores também utilizam memória *Flash* para armazenamento de informações bem como para armazenar programas ou imagens para possibilitar a reprogramação da rede (FOK; ROMAN; LU, 2005b).

É importante que a memória utilizada apresente baixa latência, o que fará com que o tempo de acesso à memória gasto pelo processador seja menor. Isso pode contribuir para redução da energia gasta pelo nó uma vez que o processador passa menos tempo processando os dados que provém da memória do nó. Um tipo de memória que apresenta este tipo de característica são as memórias RLDRAM (MICRON, 2011).

2.1.4 Unidade de Processamento

As RSSFs tem como uma característica um poder de processamento muito baixo devido aos requisitos de tamanho e consumo dos nós-sensores. Por estes motivos, nós-sensores típicos utilizam unidades microcontroladas (MCUs) de baixo processamento e também baixo consumo. Entre as MCUs que estão disponíveis atualmente no mercado, se destacam a *Atmel ATmega 128L* e a Texas Instrument (TI) *MSP430*

pelo fato de serem MCUs com baixa potência, terem interfaces para diversos sensores e ainda pela larga quantidade de ferramentas de desenvolvimento disponíveis. Existe sempre a possibilidade de se utilizar outros processadores conforme a demanda do produto, como por exemplo um ARM Cortex M8, com unidade de gerência de memória.

Podem ser utilizadas ainda unidades de Processamento digital de sinais para atender aplicações específicas que necessitam deste tipo de recurso. Os *Digital Signal Processors* (DSPs) servem para processar sinais discretos com filtros digitais, que minimizem o efeito do ruido nos sinais. O *hardware* implementado dentro do DSP é relativamente mais simples quando comparado ao hardware dedicado para o processamento analógico do sinal. A utilização de um DSP é válida em ambientes com muito ruído, que exige muita filtragem do sinal. O custo computacional para se fazer o mesmo tipo de operações dentro do processador principal é geralmente maior. Um ponto fraco na utilização de DSPs é no que diz a implementação de protocolos. O DSP é muito bom para fazer operações numéricas mas não é útil para implementar algum protocolo de rede por exemplo. Este tipo de tarefa comumente já é implementado dentro de sistemas operacionais comuns, como o Linux por exemplo.

Os nós sensores das RSSFs contam com um *hardware* muito limitado e requisitos muito críticos de energia, que fazem com que o processamento e a comunicação se tornem tarefas muito críticas. Os DSPs dedicados tendem a suprir a demanda de alto processamento para aplicações específicas que podem já conter algum tipo de aceleração em hardware para tarefas mais comuns relacionadas ao processamento de sinais e tendem a aumentar a vida útil do nó sensor, entretanto é uma tecnologia mais cara.

Para suprir requisitos ainda mais específicos, podem ser utilizados circuitos integrados de aplicação específica, ou *Application-specific integrated circuit* (ASICs). Os ASICs são circuitos integrados dedicados para alguma aplicação específica. ASICs oferecem uma estrutura projetada especificamente para a aplicação, o que lhes confere uma performance mais alta. A maior desvantagem do seu uso é o alto custo de fabricação e a falta de reconfiguração. ASICs são utilizados como complemento de processadores ou DSPs, e não substituindo os mesmos. A ideia de se usar um ASIC é a utilização de um coprocessador para uma tarefa muito específica e de alto custo para o processador principal.

2.1.5 Unidade de Sensoriamento

A unidade de sensoriamento dos nós-sensores é composta geralmente por sensores e conversores analógico-digital. Alguns nós ainda apresentam atuadores nestes módulos tornando-os não só responsáveis por fazer as medidas, mas também por atuar no sistema onde eles estão inseridos. A Figura 3 mostra a interface típica entre unidade de processamento e os sensores contidos no nó-sensor. Em uma RSSF podem existir múltiplos nós com características diversificadas. Estes nós podem ser nós simples com apenas um sensor, sendo naturalmente menos flexíveis por atenderem um número reduzido de aplicações. Os nós podem ser também complexos, com mais de uma unidade de sensoriamento. Este tipo de nó pode ser encontrado em VANTs por exemplo, onde é necessário ter mais de um tipo de sensor para que se possa controlar a aeronave.

A unidade de sensoriamento é muito dependente da aplicação, pois depende muito do fenômeno que se está tentando monitorar. O tipo de sensor pode au-

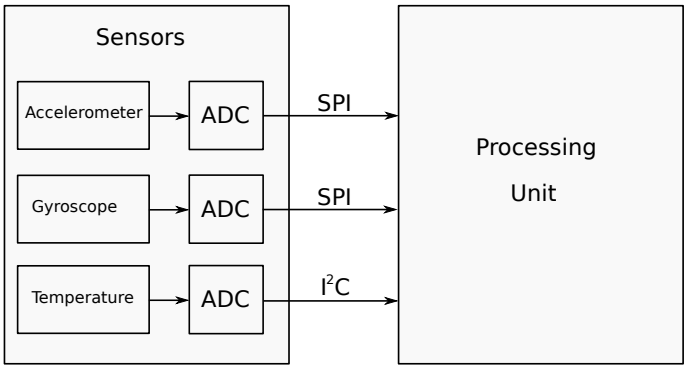

Figura 3: Comunicação entre os sensores e o processador (AKYILDIZ et al., 2002).

mentar a complexidade do hardware do nó sensor, devido principalmente a energia requerida por este sensor. Existem sensores que podem servir como uma maneira de realimentar o sistema do nó sensor com energia provinda do fenômeno que eles estão monitorando.

- **Acelerômetros**: Utilizado para medir movimentos em 2D ou 3D de objetos. Estes sensores são utilizados para medir-se fenômenos como atividades vulcânicas, rigidez de estruturas ou até mesmo irregularidades em trilhos de trem.

- **Sensores piezoelétricos**: Sensores piezoelétricos transformam pressão em um sinal elétrico. Este tipo de sensor vem sendo utilizados diversas aplicações como por exemplo em monitoramento de gás em minas de carvão (HAN et al., 2011) ou redes de sensores sem fio subaquáticas (SANCHEZ et al., 2011). Devido às suas características de conversão de um sinal mecânico em um sinal elétrico, estes sensores também são utilizados para a realimentação do nó sensor, convertendo a energia do meio em energia reaproveitável pelo nó-sensor (LEE et al., 2008).

- **Sensores capacitivos**: Sensores capacitivos podem ser utilizados para medir pressão bem como níveis de líquidos em reservatórios. Alguns trabalhos ainda apresentam o uso de sensores capacitivos para medida de umidade, tendo como utilização o monitoramento em tempo real de atividade respiratória (ANDRE et al., 2010).

- **Sensores de temperatura**: Sensores de temperatura são os mais comuns de serem encontrados em nós-sensores e praticamente toda plataforma conta com algum tipo de monitoramento de temperatura (VEERASINGAM et al., 2009).

- **Sensores magnéticos**: Sensores magnéticos e magneto-resistivos também tem suas aplicações em redes de sensores sem fio, como por exemplo o monitoramento de tráfego de veículos em estradas (CHINRUNGRUENG; KAEWKAMNERD, 2009).

2.2 Conclusão

Este capítulo mostrou um resumo do que são redes de sensores sem fio e seus componentes. Nós-sensores foram analisados em detalhes mostrando seus principais

componentes e comparando diferentes arquiteturas. RSSFs têm utilidade em diversas áreas sendo que a característica de baixo custo dos nós faz com que as RSSFs sejam principalmente utilizados em aplicações onde o processamento centralizado não alcança, como em monitoramento de desastres por exemplo.

A RSSF consiste em sensores geograficamente distribuídos para monitorar condições físicas ou ambientais e atuar cooperativamente passando-se os dados coletados para o usuário do sistema. De uma maneira geral, os nós sensores são compostos por:

- Unidade de processamento;
- Unidade de sensoriamento;
- Transmissor/Receptor;
- Memória;
- Fonte de energia.

Um nó sensor pode variar de tamanho quando comparamos diferentes fabricantes, sendo desde o tamanho de um grão de areia (O'BRIEN et al., 2009) até nós bem maiores (XBOW, 2011). O tamanho dos nós sensores está diretamente ligado à aplicação onde eles estão inseridos, sendo que a limitação de tamanho está usualmente ligada à limitação de energia gasta pelo nó sensor. Em geral, nós mais compactos tendem a consumir menos energia por possuírem menos componentes (MüLLER et al., 2012).

A diversidade das aplicações das RSSFs faz com que os nós sensores tenham diferentes capacidades e requisitos. A RSSF pode ser heterogênea contendo nós estáticos e nós móveis, e ainda sensores com diferentes complexidades. O acoplamento de nós sensores em VANTs traz um aumento na versatilidade das aplicações, uma vez que os nós ganham mobilidade física. Estes nós devem ter maior capacidade de processamento pois a complexidade inerente ao VANT requer este tipo de característica, devido ao maior número de sensores envolvidos e das tarefas que estes nós executam.

3 ARQUITETURAS RECONFIGURÁVEIS

Existem diversos métodos na computação tradicional que são utilizados na execução de algoritmos. A utilização de circuitos integrados dedicados à aplicações específicas, ou *Application Specific Integrated Circuits* (ASIC) entrega soluções eficientes. Entretanto este tipo de solução não é nada versátil, pois uma vez que os ASICs são fabricados, eles não podem mais ser alterados, o que aumenta consideravelmente o custo de uma solução deste tipo.

Uma solução mais flexível é através da utilização de microprocessadores, pois grande parte da complexidade computacional está em *software*, que pode ser alterado a qualquer instante. A solução com microprocessadores não é tão eficiente pois a leitura e execução das instruções é sequencial, e ainda existe o tempo gasto com movimentação de dados internamente ao invés do real processamento, o que faz com que esta solução tenha um *overhead* mais alto quando comparada aos ASICs.

A computação reconfigurável aparece neste contexto preenchendo esta lacuna que existe entre microprocessadores e ASICs, atingindo alta performance executando um determinado processamento em *hardware* e ainda permitindo a sua reconfiguração.

3.1 ASICs

ASICs são utilizados em aplicações onde ter uma alta performance na aplicação supera os custos envolvidos na fabricação do circuito. Existem diversos processos de fabricação de ASICs, o que leva a se ter diversas categorias de dispositivos.

O design baseado em arranjo de portas (*gate arrays*) foi introduzido no final da década de 60 por companhias como IBM e Fujitsu, mas só começou a ser difundida na metade da década de 70 com a utilização de tecnologias CMOS, o que reduziu significativamente o preço dos circuitos integrados. Os *gate arrays* são baseados na ideia de célula básica, consistindo em uma coleção de transistores, resistores e outros dispositivos desconectados, sendo que o processo físico é que vai definir a maneira como estes dispositivos serão conectados.

Projetos que utilizam somente *gate arrays* são raramente implementados por projetistas de circuitos integrados, pois este tipo arranjo pode ser substituído completamente pelo uso de FPGAs, que podem ser programados pelo usuário e, desta forma, oferecendo um custo muito mais baixo com performance relativamente parecida.

O *design* completamente customizado (*full custom*), em contraste às células básicas, tem por objetivo implementar todas as camadas do ASIC, inclusive os componentes ativos. Os benefícios de um projeto completamente customizado geralmente inclui área reduzida, melhorias de performance e ainda a possibilidade de integrar

componentes analógicos e outros pré projetados, como por exemplo núcleos de microprocessadores. As desvantagens deste tipo de abordagem podem incluir um aumento no tempo de projeto e também no preço. O aumento na complexidade nas ferramentas utilizadas para fazer o projeto do circuito integrado bem como o uso de profissionais mais bem qualificados e experientes justifica o aumento no preço final do projeto.

Uma outra abordagem é a utilização de células padrão (*standard cells*) no lugar de arranjo de portas. Este tipo de abordagem apareceu no início da década de 80 como uma maneira de resolver os problemas encontrados pelo uso de arranjo de portas. Células padrão não utilizam o conceito de células básicas, e não existem componentes pré-fabricados no *chip*. Nas células padrão, os blocos de lógica é que são criados pela composição de células ou portas básicas vindas de uma biblioteca. O uso desta abordagem permite a criação de funções lógicas com número menor de transistores em comparação às células padrão. Na década de 90, ferramentas de síntese lógica foram criadas o que permitiu gerar-se um *netlist* descrevendo a ligação das portas através de uma linguagem de descrição de hardware de alto nível, tal como VHDL ou Verilog.

3.2 Dispositivos Integrados

Diapositivos integrados podem ser divididos genericamente em categorias baseadas em complexidade e flexibilidade. Em um extremo, tem-se os *Programmable Logic Devices* (PLDs), que fornecem uma arquitetura reconfigurável e programável, enquanto que em outro extremo tem-se ASICs, que oferecem uma arquitetura dedicada para solucionar problemas de alta complexidade oferecendo uma plataforma com alta performance. Para preencher a lacuna que existe entre estas arquiteturas que diferem-se entre flexibilidade e complexidade, aparecem os *Field Programmable Gate Arrays* (FPGA), que são dispositivos programados pelo usuário que conseguem implementar funções de alta complexidade em um sistema com uma performance aceitável.

3.2.1 PLDs

Os primeiros dispositivos programáveis, que foram chamados genericamente de PLDs, começaram a aparecer no mercado no final da década de 70, com a capacidade de executar funções lógicas relativamente complexas. No início da década de 70 eles já existiam como sendo uma extensão de tecnologias baseadas em PROMs (*Programmable Read Only Memory*).

Estes dois tipos de PLDs, foram divididos em categorias onde os mais complexos são chamados de CPLDs (*Complex Programmable Logic Devices*) e os mais simples são chamados de SPLDs (*Simple Programmable Logic Devices*). Na metade da década de 80, os CPLDs começaram a ser fabricados com tecnologias baseadas em EPROM e CMOS, o que diminuiu o consumo destes dispositivos ainda aumentando a complexidade dos circuitos que poderiam ser implementados com eles. Atualmente, existe uma variação dos CPLDs que pode ser apagada e reprogramada, que é chamada de EPLD (*Erasable Programmable Logic Device*). PLDs são utilizados para a inicializar sistemas com configurações padrão pois a sua programação não é perdida quando ele não está alimentado.

3.2.2 FPGAs

Existe um espaço muito grande entre a flexibilidade oferecida pelos PLDs e a complexidade conferida aos ASICs. Na metade da década de 80, os PLDs já não suportavam mais a complexidade necessária para a fabricação de tecnologias reconfiguráveis, e os ASICs eram uma solução muito cara e demorada para algumas situações. Os FPGAs foram inicialmente desenvolvidos pela Xilinx, e apareceram neste contexto como sendo uma tecnologia intermediária entre flexibilidade e customização.

Os primeiros dispositivos eram baseados em blocos lógicos programáveis, sendo que cada bloco era composto de uma *look-up table* (LUT) de três entradas, um registrador que poderia atuar como um *flip-flop* ou um *latch* e um multiplexador (MUX). A Figura 4 ilustra estes blocos programáveis.

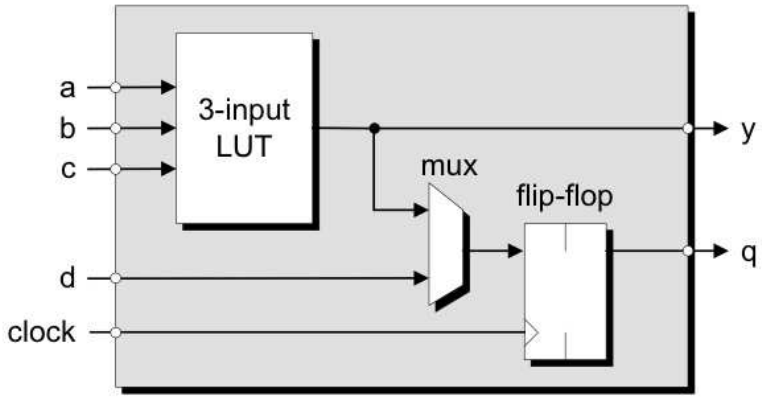

Figura 4: Bloco lógico programável encontrado em FPGAs (MAXFIELD, 2004).

Atualmente FPGAs incorporam diversas funcionalidades, como por exemplo blocos dedicados de DSP, ou ainda a possibilidade de implementação de microcontroladores em lógica programável, chamados de *soft-cores*. Os FPGAs utilizam células baseadas em memórias SRAM, o que faz com que a sua programação se perca quando a alimentação do dispositivo é interrompida, além de fornecer um baixo tempo de programação. Além destas funcionalidades, atualmente existem também os SoC FPGAs, que são FPGAs com um núcleo físico de um processador no seu chip, como por exemplo o Virtex4 da Xilinx que conta com um núcleo de PowerPC.

3.3 Reconfiguração dinâmica

A reconfiguração dinâmica de dispositivos permite que estes sejam reconfigurados durante o seu uso. Normalmente, este abordagem está associada ao uso de FPGAs, onde eles são capazes de serem reprogramados durante o seu uso através de um processador externo, ou mesmo utilizando-se microcontroladores implementados internamente ao FPGA.

Programar um FPGA baseado em tecnologia SRAM é basicamente escrever 0s ou 1s nas regiões que correspondem aos dispositivos lógicos. A Figura 5 mostra esta abordagem com um bloco que está previamente desconfigurado e então passa a estar configurado.

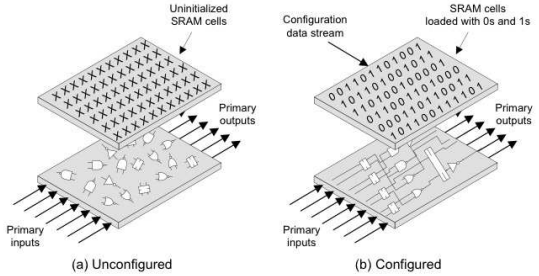

Figura 5: Reconfiguração dinâmica de FPGAs com células SDRAM (MAXFIELD, 2004).

Uma aplicação da reconfiguração dinâmica é o auto teste dos sistemas. Inicialmente, o FPGA é carregado com um *bitstream* que tem como função testar o sistema onde ele está inserido (KANSAL et al., 2011). O FPGA neste caso é testado, e após a execução desta rotina de testes, um outro *bitstream* é carregado com as funções que deverão ser executadas normalmente no FPGA. Reconfigurar dinamicamente os dispositivos aumenta a versatilidade, uma vez que ganha-se a possibilidade de trocar o *hardware* do sistema sem a necessidade de ter acesso físico ao mesmo.

FPGAs baseados em células SRAM trouxeram ainda a possibilidade de reconfigurar partes do circuito enquanto outras partes ainda continuam operando. Essa técnica é conhecida como reconfiguração parcial, pois permite que alguns blocos lógicos sejam reprogramados sem afetar outros que estão em funcionamento. Reconfiguração parcial é um tema de interesse deste trabalho embora este projeto tenha sido desenvolvido sem o uso de reconfiguração parcial.

Sistemas modulares implementados em FPGA podem se beneficiar da possibilidade de se poder carregar e descarregar módulos em tempo de execução. Os *bitstreams* dos FPGAs atualmente estão com tamanhos razoáveis o que faz com que a reconfiguração completa do dispositivo seja demorada. A reprogramação parcial permite que seja possível reprogramar partes do FPGA sem que outras deixem de funcionar no processo, e ainda com a vantagem de levar menos tempo para ser feita (SEDCOLE et al., 2005).

3.4 Conclusão

Arquiteturas reconfiguráveis permitem que o sistema seja programado e reprogramado sistematicamente. Este tipo de flexibilidade é necessária pois os requisitos do sistema podem mudar ao longo do tempo. Existem diversas arquiteturas programáveis atualmente que permitem a reconfiguração do sistema. Este trabalho tem um interesse nas arquiteturas com FPGAs pois elas permitem o paralelismo de operações conferindo ainda a possibilidade de rápida reconfiguração. Reconfiguração parcial também é objeto de interesse deste trabalho por possibilitar a reconfiguração do *hardware* sem que haja interrupção em blocos que estão sendo executados dentro do FPGA.

4 SISTEMAS MULTI-AGENTES

Sistemas multi-agentes (SMA) são sistemas compostos por múltiplos agentes que interagem entre si dentro de um ambiente. O ambiente, neste caso, tem um papel importante na arquitetura dos SMA pois eles definem a maneira pela qual os agentes estarão interagindo. Estes sistemas são utilizados para resolver problemas que são de difícil resolução para um agente único ou por um sistema monolítico. A inteligência agregada aos agentes faz com que eles sejam capazes não somente de reagirem a estímulos externos, mas também de aprenderem com estes estímulos e adaptar-se a eventuais alterações na rede onde eles estão inseridos.

4.1 Agentes

Atualmente não existe um consenso sobre a definição do termo agente pois existem ainda muitos debates sobre o assunto. Existe um consenso geral que autonomia é uma das premissas que um agente deve seguir, mas as discussões vão muito além disso. Diferentes aplicações podem exigir ou não a autonomia do agente. Por exemplo, algumas aplicações podem exigir que agentes aprendam baseados no meio que estão e a partir disso eles passem a tomar decisões. Para outras aplicações, entretanto, isso não é importante, pois somente o fato dos agentes interagirem com o ambiente, fazendo medições e atuando já é suficiente. Uma das definições de agente feita em (WOOLRIDGE; WOOLDRIDGE, 2001) é que um agente é um sistema computacional que está situado em algum ambiente, e é capaz de agir autonomamente neste ambiente para que os objetivos de projeto sejam cumpridos. Outra definição feita por (RUSSELL; NORVIG, 2003) aborda o agente como sendo uma entidade autônoma que realiza observações através de sensores e que atua sobre um determinado ambiente guiado por seus objetivos.

De uma maneira geral, um agente é uma entidade inserida em um ambiente onde ele é capaz de perceber como este ambiente se comporta através da observação de fenômenos neste ambiente, obtida por meio de sensores, e a partir de certas premissas ele é capaz de atuar neste ambiente. A Figura 6 mostra um agente genérico interagindo com o ambiente a sua volta.

Um agente tem as seguintes características:

- **Autonomia**: Agentes operam sem intervenção direta de humanos ou outros e ainda têm algum tipo de controle sobre suas ações.

- **Habilidade social**: Agentes interagem com outros agentes através de algum tipo de linguagem de comunicação entre agentes.

- **Reatividade**: Agentes percebem o ambiente e são capazes de atuar sobre o mesmo.

- **Pró-atividade**: Agentes não agem simplesmente em resposta a algum estímulo do ambiente, eles também são capazes de exibir iniciativa iniciando algum tipo de tarefa no meio onde se encontram.

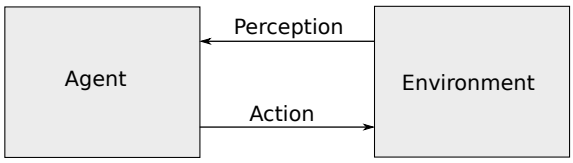

Figura 6: Um agente simples interagindo com o ambiente (WOOLRIDGE; WOOLDRIDGE, 2001).

Agentes podem ser comparados a objetos no modelo de orientação a objetos, entretanto a arquitetura interna de um agente se difere da arquitetura de um objeto, bem como o paradigma de comunicação. Os objetos são uma instância de uma classe e podem ter qualquer tipo de estrutura e de variáveis declaradas em seu interior, sendo que os métodos, ou seja, as funções associadas à classe, são a maneira pela qual os objetos têm de enxergar outros objetos. Um agente guarda em sua estrutura interna regras e estruturas associadas ao ambiente no qual ele se encontra bem como informações de outros agentes.

Um exemplo de agente é qualquer sistema de controle, seja ele complexo ou extremamente simples. Na literatura é muito utilizado o simples caso de um termostato. Estes dispositivos são utilizados como sensores de temperatura de um ambiente e produzem como saída sinais que indicam a temperatura do ambiente, podendo ser utilizados para determinar-se se o ambiente está muito quente ou muito frio. A partir da leitura destes sinais, o atuador vai começar a esquentar ou resfriar a sala, dependendo da temperatura de referência utilizada.

4.1.1 Modelo BDI

Os conceitos utilizados em inteligência artificial (IA) e em SMA têm origens em outras áreas do conhecimento que não a computação, como por exemplo na filosofia e na psicologia na análise do comportamento humano. Bratman desenvolve em (BRATMAN, 1987) uma teoria de planejamento e intenções. Intenções são tratadas como elementos de planos parciais de ação. Estes planos têm papéis básicos em raciocínio prático, que suportam a organização das atividades dos seres humanos ao longo do tempo e socialmente. Bratman explora o impacto desta abordagem em uma vasta classe de problemas, incluindo as relações entre intenção e ação intelectual, e também a distinção entre efeitos esperados e desejados.

Para Bratman, intenção e desejo são ambas atitudes pró-ativas, ou seja, atitudes mentais relacionadas à ação, mas a intenção é distinguida como uma atitude de controle de conduta. Ele identifica comprometimento como sendo o fator que diferencia o desejo da intenção, levando à persistência temporal dos planos.

A arquitetura de agentes chamada de *Belief-Desire-Intention* (BDI) é baseada nestas premissas. O modelo BDI endereça parcialmente estas premissas levantadas

por Bratman. A persistência temporal, por exemplo, no sentido explícito de referência ao tempo, não é explorada. A natureza hierárquica dos planos é mais facilmente implementada. Um plano consiste em um número de passos, os quais podem ou não invocar outros planos. A definição hierárquica dos planos implica um tipo de persistência temporal, visto que o plano abrangente continua efetivo enquanto que planos subsidiários estão sendo executados.

Na sua essência, o modelo BDI provê um mecanismo para separar as atividades de selecionar um plano da lista de planos que estão ativos. Consequentemente, agentes BDI são capazes de balancear o tempo gasto na tomada de decisão e na execução destes planos. Uma terceira atividade, que é fazer o planejamento, não está no escopo do modelo BDI e é deixada ao programador e ao projetista do sistema.

A arquitetura do modelo BDI divide-se então nas crenças, desejos e intenções dos agentes. As crenças (*beliefs*) representam o estado do agente, ou seja, suas crenças sobre o mundo, incluindo ele mesmo e outros agentes. *Beliefs* podem inclusive incluir regras de inferência, permitindo criar-se novas crenças. É importante ressaltar que a crença de um agente não representa necessariamente a realidade, e elas podem ser alteradas ao longo do tempo. Desejos (*desires*) representam o estado motivacional do agente. Eles representam objetivos ou situações que o agente gostaria de realizar. Exemplos podem ser: encontrar o melhor preço em uma lista, ir a uma festa ou tornar-se rico. Intenções (*intentions*) representam o estado deliberativo com o qual o agente tem algum compromisso. Planos são sequências de ações que um agente pode executar para alcançar uma ou mais de suas intenções. Planos podem conter outros planos, por exemplo, um plano de dirigir um carro envolve um plano de achar as chaves do carro. Isso reflete que no modelo de Bratman, planos são inicialmente parcialmente definidos, sendo que os detalhes vão sendo preenchidos ao longo da sua execução.

4.1.2 Sistemas Multi-Agentes e Seus Ambientes

Sistemas complexos tais como SMA são compostos por agentes de naturezas iguais ou diferentes. Estes agentes são capazes de interagir entre si diretamente através de troca de mensagens ou indiretamente, atuando no meio e fazendo com que outros agentes percebam suas intenções. O ambiente no qual o SMA se encontra tem um papel tão importante quanto aos agentes, pois é sobre ele que os agentes estão atuando e fazendo suas medidas. Em (WOOLRIDGE; WOOLDRIDGE, 2001) os ambientes são caracterizados como sendo acessíveis ou inacessíveis, determinísticos ou não determinísticos, estáticos ou dinâmicos e ainda discretos ou contínuos.

- **Acessível x Inacessível**: Um ambiente acessível é aquele onde pode obter-se informações completas, precisas e atualizadas sobre o estado do ambiente. A maioria dos ambientes reais (incluindo por exemplo o mundo físico e a Internet) são inacessíveis.

- **Determinístico x Não-Determinístico**: Um ambiente determinístico é aquele no qual qualquer ação tem um efeito garantido. Não existem incertezas sobre o estado que será levado o sistema após ser efetuada alguma ação sobre ele.

- **Estático x Dinâmico**: Um ambiente estático é aquele onde se pode assumir que ele permanecerá inalterado com exceção das ações tomadas pelos agentes

que nele estão inseridos. Em contrapartida, um ambiente dinâmico é aquele que existem outros processos operando sobre ele, e no qual as alterações vão muito além do controle do agente, como no exemplo do termostato citado anteriormente. A maioria dos ambientes reais são dinâmicos.

- **Discreto x Contínuo**: Um ambiente é dito discreto se existe um número fixo e finito de ações e percepções a ele.

Um exemplo simples disso seria uma sala com sensores de temperatura espalhados por ela para medir a temperatura e controlar a temperatura da sala a partir de um ar-condicionado central. Rigorosamente, este sistema seria inacessível, pois nem todos os pontos da sala podem ser medidos. Entretanto é possível a partir de medidas de alguns pontos inferir todos os valores de temperatura da sala. Este sistema é também não-determinístico e dinâmico, pois as ações de controle de temperatura tomadas pelos agentes não necessariamente terão um efeito garantido pois podem existir eventos externos que atrapalhem no controle de temperatura, como por exemplo um humano entrando e saindo desta mesma sala. Por fim, este sistema é discreto pois existe um número finitos de sensores e agentes atuando sobre ele.

4.2 Classificação de Sistemas Multi-Agentes

Em (RUSSELL; NORVIG, 2003) os SMA são classificados como sendo reativos ou cognitivos enquanto que em (WOOLRIDGE; WOOLDRIDGE, 2001) também se trata de uma outra classe de agente chamada de agentes híbridos.

4.2.1 Sistemas Multi-Agentes Reativos

Agentes reativos também são chamados de agentes baseados em comportamento. Este tipo de agente, em contraste com os agentes que são baseados em crença que têm a sua própria visão do mundo, é definido somente por um conjunto de comportamentos. Agentes reativos não necessitam de memória ou de uma visão do mundo exterior pois eles só reagem aos estímulos que recebem. Neste tipo de abordagem é muito comum o comportamento dos agentes ser baseados em modelos de organização biológica, como por exemplo formigas, abelhas ou cupins (FREITAS, 2012). A Figura 7 mostra um diagrama de blocos de um agente puramente reativo. Pode-se perceber que o agente aplica um comportamento baseado em uma entrada e isso é o resultado que é posto na sua saída.

Figura 7: Modelo de um agente reativo (RUSSELL; NORVIG, 2003).

4.2.2 Sistemas Multi-Agentes Cognitivos

Agentes cognitivos têm ações baseadas em outros tipos de organização, tais como a social humana. Este tipo de agente é capaz de aprender novos comportamentos

baseados na sua leitura do ambiente. Este tipo de agente conhece outros agentes e é capaz de se comunicar com os mesmos através de troca de mensagens diretas. Os agentes cognitivos ainda carregam o conceito de inteligência coletiva, sendo capazes de interagir com outros agentes e tomar uma decisão em grupo.

O modelo de agente cognitivo chamado de Clarion, primeiramente citado (SUN; PETERSON, 1996), é uma arquitetura voltada para o aprendizado do agente. Essa arquitetura é separada em dois níveis, um conceitual, ou chamado de conhecimento declarativo, e outro sub conceitual, também chamado de conhecimento procedural. O nível conceitual abstrai o ambiente e é capaz de guardar aprendizado em uma espécie de memória. Um diagrama de blocos que representa esta arquitetura pode ser visto na Figura 8.

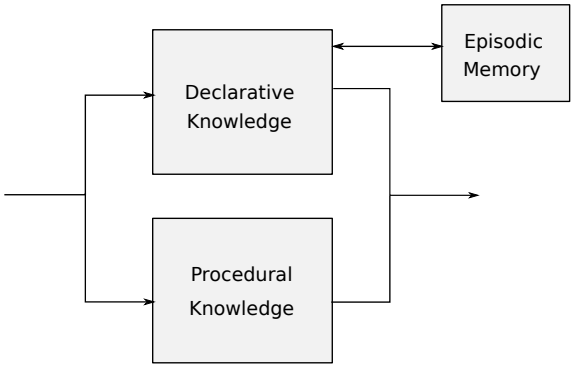

Figura 8: Modelo de um agente cognitivo (SUN; PETERSON, 1996).

4.2.3 Sistemas Multi-Agentes Híbridos

Agentes híbridos são entidades que têm o comportamento tanto reativo quanto pró-ativo. Estes sistemas são implementados de forma a terem diferentes subsistemas tratando em paralelo a parte reativa e a parte cognitiva. Agentes híbridos são capazes então de não só reagir somente a estímulos e com isso modelar um comportamento, mas também de criar um raciocínio coletivo com os outros agentes baseado no sistema onde eles sem encontram. A Figura 9 mostra uma arquitetura horizontal chamada de máquina de *Turing*.

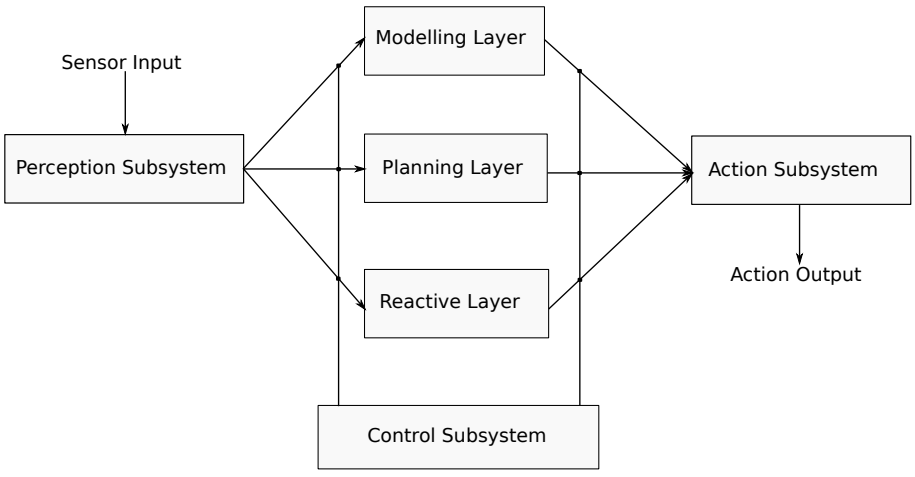

Figura 9: Modelo de um agente híbrido (WOOLRIDGE; WOOLDRIDGE, 2001).

4.3 Agentes Móveis

Mobilidade é um conceito importante para agentes que estão sendo executados em aplicações de RSSFs. Este trabalho tem o foco na mobilidade de agentes entre nós, então este tópico é de fundamental interesse. De uma maneira geral, um agente móvel é uma entidade capaz de se mover de um nó sensor para outro na rede. A sua execução pode ser iniciada em um nó e continuar sendo executada no outro nó a partir do ponto onde parou antes da migração. Esta migração pode estar carregando o código do agente, ou seja, seu comportamento previamente definido, e também código e dados, sendo que neste caso o escopo do agente é transferido junto durante o processo de migração.

Alguns autores chamam os dados de *estado* do agente (CHESS et al., 2000), e uma migração carregado somente código é chamada de *migração sem estado* e uma migração onde os dados são carregados juntos no processo é chamada de uma migração com *estado completo*. Existem outros trabalhos na área de mobilidade de agentes que dão outro nome a este processo (FOK; ROMAN; LU, 2005b), onde uma migração do agente sem os dados é chamada de migração *fraca* enquanto uma migração do agente com os dados é chamada de migração *forte*.

Conforme definido em (LANGE; OSHIMA, 1999), agentes móveis podem trazer diversas melhorias para a arquitetura de uma RSSF e existem pelo menos sete boas razões para se usar este tipo de tecnologia.

1. *Redução da carga na rede.* Agentes móveis podem ser enviados à localidades remotas onde há necessidade de se fazer algum tipo de tarefa, o que pode fazer com que o tráfego na rede seja diminuído significativamente. Quando um grande volume de dados está sendo armazenado em algum lugar remoto, é mais fácil e barato enviar a aplicação para onde estão os dados do que o contrário.

2. *Redução na latência da rede.* Sistemas com requisitos críticos de tempo real precisam ter uma rede com baixa latência para que o controle distribuído consiga ser executado dentro dos requisitos especificados. Agentes móveis podem resolver o problema de dependência de uma rede com baixa latência pois eles podem ser enviados para onde está a seção crítica, evitando que o controle fique dependente da rede.

3. *Encapsulamento de protocolos.* Novos protocolos estão em constante desenvolvimento e o núcleo da rede não é alterado na mesma proporção. Como resultado, protocolos geralmente sofrem de um problema de manutenção de códigos legados para manter-se uma interface coerente entre as diversas implementações. Visto que agentes móveis podem se mover entre os nós, eles podem então criar um novo meio de comunicação que pode ser facilmente alterado.

4. *Autonomia e assincronismo.* Tarefas que precisam manter uma conexão aberta entre um dispositivo móvel e uma rede fixa podem ser muito caras e, por vezes, nem sequer são realizáveis. Agentes podem resolver este problema encapsulando tarefas que serão enviadas aos nós remotos e então elas passam a ser executadas autonomamente sem dependência com a rede física evitando a necessidade de se manter uma conexão entre as duas redes.

5. *Adaptação dinâmica.* Visto que agentes são entidades que têm uma certa inteligência, eles podem ler os dados do ambiente e reagir autonomamente devido a qualquer mudança nos requisitos do sistema.

6. *Heterogeneidade.* Agentes são dependentes somente do ambiente onde eles estão sendo executados, e por causa disso, eles podem prover boas condições para a integração de sistemas.

7. *Tolerância à falhas.* Agentes podem monitorar o hardware do sistema que eles estão inseridos e a partir de falhas iminentes se auto redistribuírem de acordo com a necessidade. A maioria das situações de falha acontecem muito rapidamente e, às vezes, não existe tempo hábil para manualmente se trocar a configuração da rede.

4.4 Jade

Jade é um acrônimo para *Java Agent DEvelopment*, e é um *framework* de software utilizado em SMA implementado em java e está em desenvolvimento desde 2001 (TILAB, 2012). A plataforma do Jade permite a coordenação de múltiplos agentes compatíveis com as especificações normatizadas pela FIPA (FIPA, 2002) para comunicação entre agentes.

O Jade cria o conceito de múltiplos *containers* para agentes, que podem estar espalhados através da rede e juntos eles formam a plataforma como um todo. Cada plataforma deve ter um único *container* principal que carrega dois agentes especiais chamados de agentes AMS e DF. O Agente AMS (*Agent Management System*) é a autoridade na plataforma que pode criar ou destruir outros agentes, *containers* e ainda desativar a plataforma. O agente DF (*Directory Facilitator*) implementa o serviço de páginas amarelas que publica os serviços de todos os agentes na plataforma para que os agentes possam saber os serviços oferecidos por outros agentes.

4.4.1 Comunicação entre agentes

A comunicação entre os agentes Jade é feita através da troca de mensagens, onde é utilizada FIPA-ACL como base de cada mensagem. A plataforma de agentes pode ser distribuída em diversos *hosts* onde cada um necessita somente de uma única máquina virtual java (JVM) sendo executada. Cada JVM é basicamente um *container* de agentes que provê um ambiente de execução completo para a execução dos agentes e permite que diversos agentes serem executados concorrentemente no mesmo *host*.

Cada agente é diretamente conectado a um *container*, o qual é controlado externamente por algum gerenciador como por exemplo um computador comum. Os agentes comunicam-se diretamente aos *containers* mas também comunicam-se entre si, sendo que eles conhecem os serviços dos outros agentes através do *container* principal, que guarda tabelas com serviços de todos os agentes que estão ligados a eles. Tem-se então dois paradigmas diferentes de comunicação. Por um lado temos uma arquitetura cliente-servidor, onde os clientes são os agentes que se comunicam com o servidor, ou seja, o *container*. Por outro lado temos uma topologia ponto a ponto, pois os agentes também se comunicam entre si. A Figura 10 mostra a arquitetura de alto nível do Jade, com diferentes *containers* e diferentes agentes se comunicando entre si.

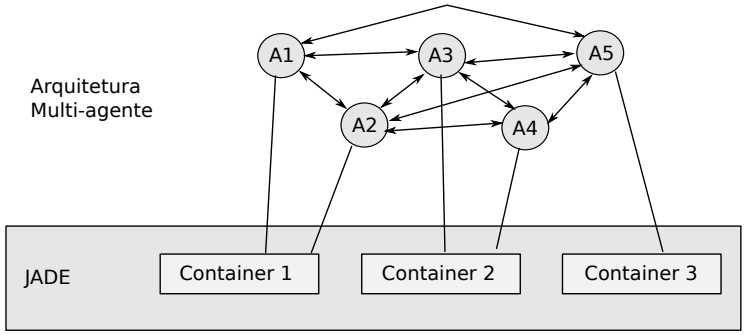

Figura 10: Arquitetura de alto nível do Jade (TILAB, 2012).

A FIPA tem um conjunto de especificações que definem a maneira como os agentes se comunicam entre si (FIPA, 2002). O objetivo destas especificações é definir formatos de mensagem, protocolos e encapsulamento utilizados na comunicação, para que possa manter a interoperabilidade entre diversas implementações.

Um dos principais componentes no modelo de comunicação da FIPA é o canal de comunicação entre agentes (ACC). O ACC é o componente na plataforma de agentes que fornece os serviços de transporte das mensagens. Na arquitetura da FIPA, um agente tem três opções ao enviar uma mensagem para outro agente situado em uma plataforma remota (LAUKKANEN; HELIN; LAAMANEN, 2002).

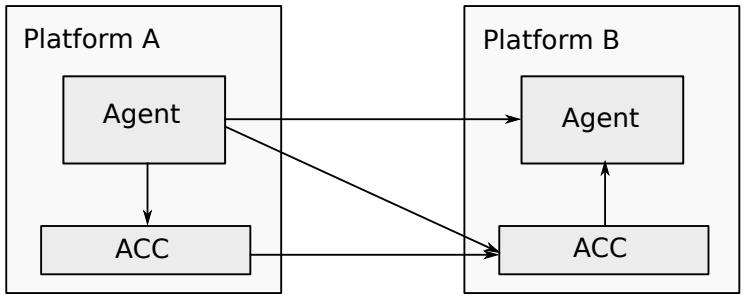

Figura 11: Diferentes tipos de comunicação entre agentes (TILAB, 2012).

Primeiro, um agente A envia uma mensagem para o ACC local e neste caso não está sendo utilizado o protocolo de transporte de mensagens (MTP). Após, o ACC local envia a mensagem ao ACC remoto que por sua vez entrega a mensagem ao agente B. Segundo, um agente A poderá enviar uma mensagem diretamente ao ACC remoto, que por sua vez entregará a mensagem ao agente B. Nesta opção, o agente A deverá implementar o MTP utilizado na comunicação na sua implementação, para que o o ACC remoto consiga se comunicar diretamente com este agente. Por último, o agente A e o agente B podem comunicar-se diretamente um com o outro sem usar os serviços fornecidos pela plataforma. A Figura 11 mostra estas três diferentes maneiras de levar uma mensagem a partir de um agente A até um agente B.

Pode-se dividir a formação das mensagens em diversos níveis, ou em camadas de comunicação, conforme pode ser visto na Figura 12 (NGUYEN; SCHAU; ROSSAK, 2011), onde é mostrado somente as camadas referentes à formação da mensagem que é trocada entre os agentes.

A primeira camada mostrada é a camada que define a linguagem de comunicação entre agentes - *Agent Communication Language - ACL*. Esta camada especifica a

Figura 12: Camadas de comunicação utilizadas entre os agentes (TILAB, 2012).

sintaxe e a semântica das mensagens. FIPA-ACL foi desenvolvida especificamente para descrever e facilitar o processo de comunicação entre agentes. Uma mensagem ACL pode ser considerada uma coleção de tipos de mensagens, onde cada uma corresponde a um significado pré-definido. As mensagens ACL não lidam com a troca física através da rede, mas somente especificando o conteúdo de troca pois existe outra camada que se preocupa com definir o transporte das mensagens. As especificações da FIPA (FIPA, 2002) definem 22 tipos de mensagens que são chamadas de *performative*.

A segunda camada define o encapsulamento das mensagens, e com a introdução desta camada, as mensagens podem ser enviadas entre as plataformas independentemente do seu conteúdo. O ACC transfere as mensagens de um agente ao outro sem ler a mensagem ACL, ele somente cria uma camada a mais e envia a mensagem. Segundo a especificação da FIPA, são definidos três tipos de envelopes nesta camada. O primeiro é definido pela FIPA-00075 (FIPA, 2002) que especifica o transporte de mensagens entre agentes usando o *Internet Inter-Orb Protocol* - IIOP. O segundo é definido pela FIPA-00085, que especifica a sintaxe do envelope de uma mensagem com o padrão XML. Finalmente o terceiro é definido pela FIPA-00088, que trata de envelope com codificação de *bits*, ou seja, a mensagem é codificada em padrões pré-definidos de valores em hexadecimal. A listagem A.1 mostra uma estrutura de dados que faz o encapsulamento das mensagens que são recebidas pelos ACCs (FIPA, 2002).

A terceira camada define a estrutura das mensagens utilizadas pelo protocolo de transporte de mensagens (MTP). Ela é responsável por entregar as mensagens na ordem correta ao agente destino e também informar ao agente de origem quando ocorre algum erro de comunicação. A FIPA define três tipos de MTPs:

- MTP baseado em IIOP - FIPA00075.
- MTP baseado em WAP - FIPA00076.
- MTP baseado em HTTP - FIPA00084.

A camada de transporte e sinalização tem uma contribuição importante na performance de comunicação de qualquer sistema distribuído, incluindo sistemas multiagente. Ela é responsável pela transferência de dados em baixo nível sobre a rede. É esperado que esta camada forneça um serviço de transporte eficiente e confiável.

Tipicamente são utilizados TCP e UDP nesta camada, sendo que o Jade utiliza TCP.

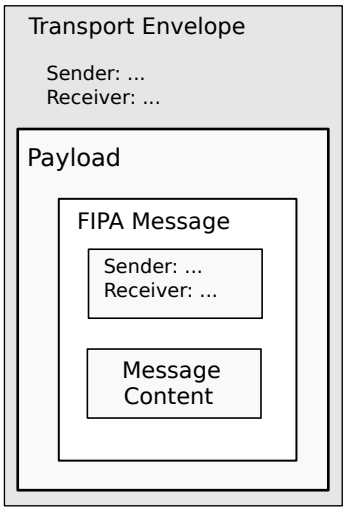

Figura 13: Encapsulamento das mensagens ACL.

Quando um agente quer enviar uma mensagem a outro agente, ele constrói uma mensagem ACL com o conteúdo apropriado. A parte ACL da mensagem contém informações sobre o remetente (*sender*) e o destinatário (*receiver*) bem como outras informações relacionadas à mensagem. Então a mensagem é transformada em um *payload* e é incluída no pacote para ser enviada. Ainda é adicionado a este pacote um envelope, que inclui a informação sobre o *sender* e o *receiver* bem como informações sobre a maneira de enviar a mensagem. Esta mensagem encapsulada ainda pode conter informações adicionais, tais como informações relacionadas à segurança. A Figura 13 mostra uma mensagem ACL após este encapsulamento.

Uma mensagem ACL contém diversos parâmetros de configuração para que ela esteja dentro do padrão definido pela FIPA. A Tabela 1 mostra alguns dos parâmetros que são utilizados com mais frequência. A estrutura de dados que implementa toda a recomendação descrita pela FIPA pode ser vista na listagem A.2.

Tabela 1: Conjunto de parâmetros de uma mensagem ACL.

Parâmetro	**Descrição**
Performative	O tipo do ato de comunicação da mensagem ACL.
Sender	Identificação do agente de origem da mensagem.
Receiver	Identificação do(s) agente(s) de destino.
Reply-to	Nome do agente para onde as mensagens subsequentes deverão ser enviadas.
Content	Conteúdo da mensagem.
Language	Linguagem na qual o conteúdo é descrito.
Ontology	A ontologia usada para dar significado aos símbolos do conteúdo da mensagem.
Reply-with	Expressão a ser utilizada pelo agente que responde a mensagem para identificar esta mensagem.
In-reply-to	Expressão que referencia uma ação anterior para a qual esta mensagem é a resposta.

4.4.2 Estrutura interna

O *framework* do Jade suporta a programação de diversos comportamentos para descrever seus agentes. Entre os comportamentos mais utilizados, pode-se destacar:

- *Cyclic*: Comportamentos cíclicos são utilizados por agentes que fazem tarefas repetitivas ao longo do tempo. Por exemplo, leitura de sensores ou monitoramento do estado de outros agentes.
- *Oneshot*: Comportamentos deste tipo são utilizados por agentes que vão executar uma tarefa uma determinada vez somente. Pode-se ter por exemplo um agente que migra quando uma certa condição é atingida.
- *Finite State Machine (FSM)*: Comportamentos do tipo máquina de estados são utilizados por agentes que necessitam de diferentes tipos de resultados baseados em alguma variável do agente.

A arquitetura do Jade é versátil ao ponto de podermos criar diferentes comportamentos para atender a uma necessidade específica. Os agentes são basicamente classes em Java onde seus comportamentos são mapeados para a arquitetura do Jade escrevendo-se os métodos que o Jade espera que sejam implementados.

O Jade implementa um gerenciador de agentes que é responsável por monitorar a vida útil dos mesmos. É este gerenciador que é responsável por criar ou destruir os agentes quando necessário. Existe também um mecanismo para escalonamento dos comportamentos dos agentes, que é utilizado internamente para atender tarefas específicas de agentes que dependem de requisitos temporais, por exemplo.

O Jade também permite que seus agentes se comuniquem com agentes externos a sua plataforma. Isso é possível pois as estruturas de comunicação do Jade são implementadas conforme a normatização da FIPA-ACL, o que garante que qualquer agente que estiver dentro da norma poderá trocar informações com agentes implementados usando o Jade.

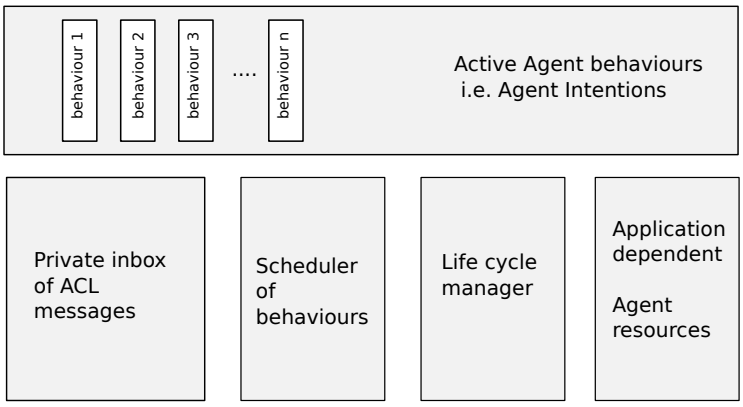

Figura 14: Arquitetura interna de um agente Jade genérico.

A Figura 14 mostra os blocos que compõem um agente Jade (TILAB, 2012). É possível se observar que um agente pode ter mais de um comportamento implementado, que é o que geralmente acontece. Todo agente contém um espaço de memória reservado para implementar a comunicação entre os agentes, que é descrita pelo bloco de mensagens ACL. Ainda, cada agente tem um espaço reservado para

guardar recursos que são dependentes de aplicação, ou ou mesmo os dados referentes ao contexto da execução do agente. Uma vez que o agente migra, todas estas informações podem ser carregadas junto com ele.

A mobilidade de agentes é dada pela capacidade de um agente migrar ou por fazer uma cópia dele mesmo em algum outro nó da RSSF. Existem diversos métodos dentro do pacote *jade.core* que são dedicados à mobilidade de agentes. O método principal da migração é o *doMove()*, pois é este que efetivamente migra o agente para uma localidade remota. Existem mais dois métodos que são chamados antes e logo após o agente ser movido, que são o *beforeMove()* e o *afterMove()* respectivamente.

Os métodos que movem o agente entre os nós sempre carregam seu contexto e fazem com que o agente continue a sua operação do ponto onde foi parado antes da migração. Uma outra abordagem é dada pelos métodos de clonagem do agente. Estes métodos fazem uma cópia do agente no nó de destino disparando uma nova instância deste agente e abortando a instância original. Não existe cópia de dados neste caso, os agentes são inicializados a partir de seu ponto inicial de execução.

4.5 Possíveis Aplicações de Agentes

Existe uma imensa variedade de aplicações possíveis de SMA citadas na bibliografia sendo que a maioria das propostas faz uso de *frameworks* abertos que permitem a fácil interligação destes SMA. Entre as aplicações possíveis podemos destacar o uso de veículos aéreos não tripulados em missões críticas (FREITAS, 2012; HAMA, 2012), controle de tráfego (LI et al., 2006), automação de sistemas de manufatura (PEIXOTO, 2012), controle de desastres (SADIK et al., 2006), fusão de dados (NESTINGER et al., 2008), monitoramento de sistemas de potência (PONCI; DESHMUKH, 2008), monitoramento de pacientes (MILLER; SURESH, 2009), sistemas de controle (BUSE; WU, 2004), análise de dados (BUSE; FENG; WU, 2003) e também monitoramento de ambientes (FOK; ROMAN; LU, 2005b).

O uso de SMA com agentes móveis aumenta a flexibilidade da RSSF, uma vez que os agentes podem ser executados em diferentes nós da rede dependendo da aplicação. Estes agentes podem ser utilizados para configurar a RSSF dinamicamente, sem a necessidade de tê-la previamente configurada. Este tipo de característica tem aplicações em ambientes que mudam constantemente de requisitos e necessitam serem programados dinamicamente.

4.6 Conclusão

Os agentes utilizados na computação tem suas origens na psicologia e análise do comportamento humano. Apesar da definição de agente ser meio difusa, existe um consenso que os agentes são entidades que possuem uma certa inteligência sendo capazes de perceber e atuar no meio onde eles se encontram. Os agentes são caracterizados como entidades autônomas, ou seja, agentes são capazes de atuar no meio sem intervenção humana. Agentes devem também ser capazes de se comunicar com outros agentes através de uma linguagem comum, o que caracteriza a habilidade social dos agentes. Agentes ainda devem ser reativos e pró-ativos, não só respondendo aos estímulos que lhes são conferidos mas também sendo capazes de tomar algum tipo de iniciativa.

Os agentes podem ainda ser móveis, sendo capazes de migrar entre diferentes

nós em uma rede. Agentes móveis podem migrar entre os nodos da rede, carregando consigo os dados associados ao escopo, bem como seus comportamentos. Agentes móveis podem ser usados para levar tarefas a nós que estão fisicamente distantes e também podem servir como uma maneira de reconfigurar a rede com novas tarefas, fazendo com que elas migrem entre os nós baseados nos requisitos da rede. Agentes podem agregar diversas tarefas fazendo que durante a sua migração, diversas configurações sejam enviadas junto ao agente que está se deslocando.

O Jade é um *framework* utilizado na criação de aplicações com agentes móveis. A utilização do Jade como plataforma de agentes traz como benefícios a implementação de uma linguagem de comunicação entre os agentes compatível com as normas estabelecidas pela FIPA. O Jade é uma plataforma de interesse neste trabalho pois ela abstrai a comunicação entre os agentes em uma linguagem normatizada. O uso de um *framework* de desenvolvimento de agentes faz com que se tenha um ambiente padrão de desenvolvimento e também garante que os agentes estarão falando uma linguagem em comum. Uma vantagem de se usar o Jade para desenvolvimento do SMA é que ele fornece um ambiente onde existe uma integração natural entre os domínios dos agentes, sendo que o sendo que os programadores dos agentes não necessitam ter um conhecimento prévio. Isso faz com que o aprendizado do Jade seja muito rápido, pois só exige conhecimentos da linguagem java.

5 TRABALHOS RELACIONADOS

Acho melhor colocar algo como: Este capitulo analisa arquiteturas de sistemas multi-agentes propostas na literatura e que posseum objetivos similares ao do proposto neste trabalho. Em especial, propostas de implementação de agentes diretamente em *hardware* são discutidas. Esta análise é importante pois ela serve como base para a criação da arquitetura proposta na utilização de partes dos trabalhos aqui introduzidos, tanto no uso do mesmo *framework* quanto na reutilização das ideias propostas como maneira de se propor o modelo de arquitetura deste trabalho.

5.1 Agilla

Agilla é um *middleware* para programação de nós sensores em RSSF (FOK; ROMAN; LU, 2005a). A ideia geral da proposta é distribuir os nós sensores fisicamente na área de interesse e configurar posteriormente a rede conforme a necessidade. A rede é programada com uma arquitetura de agentes móveis utilizando-se uma plataforma dedicada. Esta arquitetura tem como base a plataforma de nós sensores chamada de Mica2 (XBOW, 2011) e a nível de sistema operacional é utilizado o TinyOS (LEVIS et al., 2005). Este de tipo de arquitetura aumenta consideravelmente a flexibilidade da RSSF pois permite a reprogramação dinâmica dos nós sensores após a sua distribuição física no ambiente.

5.1.1 Modelo de Agente

O modelo proposto pode ser visto na Figura 15. Cada agente mantém um espaço de tuplas e uma lista de vizinhos sendo que eles tem suporte à múltiplos agentes. O espaço de tuplas é local e é dividido pelos agentes que estão residindo no nó sensor em um dado momento do tempo. Instruções especiais permitem que os agentes acessem remotamente o espaço de tuplas de outros nós. A lista de vizinhos contem os endereços de todos os nós que estão separados por um *hop*. Os agentes podem migrar carregando seu código e estado, mas o espaço de tuplas não é carregado.

Uma aplicação Agilla consiste em um vários agentes autônomos, possivelmente de diferentes tipos, distribuídos pela RSSF. Em (FOK; ROMAN; LU, 2005b) é mostrado um exemplo para detecção de queimadas em florestas, com diversos tipos de agentes se comunicando com a finalidade de detectar o incêndio o mais rapidamente possível. A comunicação entre os agentes torna-se um ponto importante no modelo pois é o gargalo da troca de informações entre os agentes e tipicamente é a parte que mais consome energia no nó sensor, conforme já foi discutido no Capítulo 2 deste trabalho. Esta comunicação no modelo do Agilla é feita através do espaço de

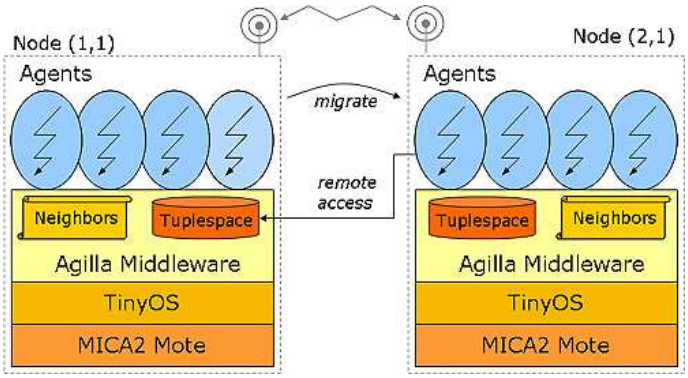

Figura 15: Modelo de agente proposto no Agilla (FOK; ROMAN; LU, 2005a).

tuplas. O espaço de tuplas do Agilla é oferecido através de um modelo de memória compartilhada onde um dado é uma tupla.

Um espaço de tuplas é uma implementação do paradigma de memória associativa para computação paralela e distribuída. Ele provê um repositório de tuplas que pode ser acessado concorrentemente e pode ser visto como uma forma de memória compartilhada distribuída. Um exemplo simples seriam dois grupos de processadores, um produzindo e outro consumindo dados. Os produtores colocam os dados no espaço de tuplas enquanto os consumidores buscam os dados que combinam com um certo padrão. Existe uma metáfora para isso, chamada de metáfora do *blackboard* (CORKILL., 1991), onde um grupo de especialistas colabora em um quadro-negro para criar uma solução para um problema complexo cooperativamente. O processo todo se dá com os especialistas colaborando e colocando parte da resolução do seu problema no quadro-negro até que o problema seja resolvido. Espaço de tuplas são a base teórica da linguagem Linda (GELERNTER, 1985) desenvolvida por David Gelernter e Nicholas Carriero na universidade de Yale.

O espaço de tuplas provê um alto nível de desacoplamento que garante que cada agente continue autônomo e fornece uma maneira conveniente de um agente descobrir seu contexto. Por exemplo, visto que cada nó pode ter um diferente número de sensores, Agilla cria tuplas especiais dentro do espaço de tuplas indicando que tipo de sensores estão disponíveis. O Agilla ainda adiciona reações ao espaço de tuplas, criando uma espécie de agente reativo, que é responsável por avisar outros agentes quando uma determinada tupla é inserida no sistema. Isso evita o constante monitoramento entre os agentes e, consequentemente, diminui o tráfego na rede.

5.1.2 Arquitetura do Middleware

A arquitetura do Agilla pode ser dividida em três grandes camadas, sendo que a mais alta contem os agentes, a camada do meio contem os componentes do *middleware* enquanto que a camada de baixo é onde se situa o sistema operacional TinyOS. O *middleware* da arquitetura consiste de um gerenciador de agentes (*agent manager*), gerenciador de contexto (*context manager*), gerenciador de instruções (*instruction manager*), gerenciador de espaço de tuplas (*tuple space manager*) e também a máquina de execução do Agilla (*agilla engine*), o que pode ser visto na Figura 16.

O *agent manager* é responsável por manter o contexto dos agentes, alocar memória para um agente que está chegando na plataforma e desalocar esta memória

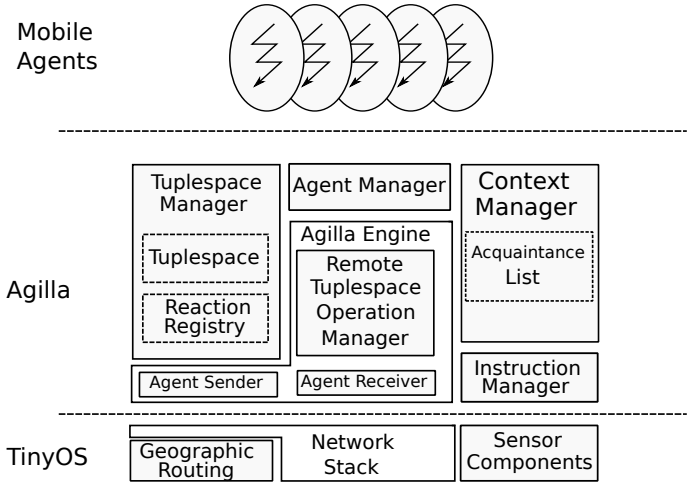

Figura 16: Arquitetura do Agilla (FOK; ROMAN; LU, 2005a).

quando este agente é terminado ou quando ele migra. Ele também é responsável por determinar quando um agente está pronto para ser executado. Por *default* o gerenciador de agentes suporta no máximo 4 agentes, mas isso é facilmente alterado sendo que os fatores que limitam são a velocidade do processador e a memória física disponível no sistema.

O *context manager* determina a localização do nó sensor bem como a de seus vizinhos. Ele utiliza *beacons* para descobrir os vizinhos e armazena o local dos vizinhos em uma lista que é acessível aos agentes através de instruções especiais.

O *instruction manager* tem um papel muito importante na arquitetura do Agilla visto que o TinyOS não provê alocação dinâmica de memória, este bloco é responsável por fazer a gerência de memória do nó sensor. Quando um agente migra e chega no nó sensor, ele especifica quando de memória de instrução é necessária para que ele seja executado, então o gerenciador de instruções aloca um valor mínimo de um bloco de 22 blocos de 1 *byte* para armazenar o código do agente. Eles chegaram a este valor mínimo empiricamente baseado no *overhead* necessário para que a maioria dos códigos da arquitetura fossem executados.

O *tuple space manager* implementa todas as operações não bloqueantes no espaço de tuplas e também a característica reativa das mesmas. Este bloco também é responsável por alocar dinamicamente a memória para o espaço de tuplas, e por *default* é alocado um valor de 600 *bytes* sendo que uma tupla pode ter até 25 *bytes*, para que uma tupla consiga ficar dentro do valor máximo de uma mensagem, que é 27 *bytes*.

O *agilla engine* serve como um *microkernel* que controla todas as execuções concorrentes dos agentes no nó. Ele implementa uma política de escalonamento do tipo *round-robin* onde cada agente pode executar um numero finito de instruções (4 instruções) antes de chavear de contexto. Quando um agente migra, o Agilla o divide em diversas mensagens. Uma migração requer no mínimo duas mensagens, sendo que uma é de estado e outra de código. Muitos agentes requerem mais mensagens, visto que eles podem ter dados no *stack* e no *heap*. Se uma única mensagem for perdida, a migração vai falhar, e para minimizar este problema, os agentes migram entre nós que distam no máximo de um *hop*.

O *middleware* suporta dois tipos básicos de migração de agentes que são cha-

madas de *soft migration* e *hard migration*. O primeiro tipo faz com que o agente migre sem carregar seu contexto de execução. A execução do agente é parada no nó de origem e seu código é migrado, começando novamente a execução do seu início no nó de destino. Isso é semelhante a uma operação de *clone* que é utilizada no *framework* do Jade. No segundo tipo de migração, o agente carrega junto com seu código de execução o seu contexto, podendo continuar a ser executado no nó destino do mesmo ponto onde foi parado antes da migração.

5.1.3 Arquitetura do Agente

A arquitetura do agente Agilla pode ser vista na Figura 17 e ele é basicamente composto por um *stack*, um *heap* e diversos registradores. O *heap* é uma unidade de armazenamento que permite que o agente armazene até 12 variáveis. O agente também conta com três registradores de 16 bits. Um contendo a identificação do agente (ID), outro contendo o contador de programa (PC) e finalmente um registrador que armazena um código de condição do agente. O ID é único para cada agente e é mantido quando um agente migra de um nó para outro. O PC é o endereço da próxima instrução a ser executada. O registro de condição guarda o estado de execução do agente, que é usado internamente pela arquitetura do Agilla.

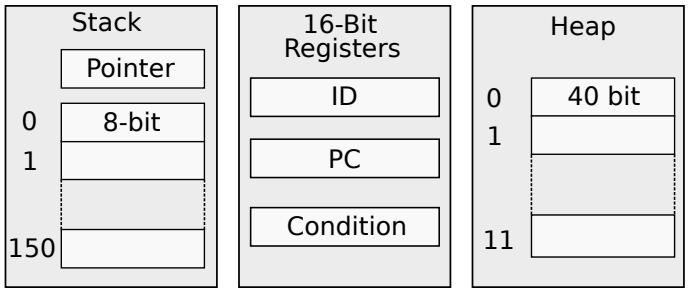

Figura 17: Formato do agente Agilla (FOK; ROMAN; LU, 2005a).

5.1.4 Resumo

A arquitetura do Agilla define um modelo próprio de agente que é fortemente atrelado à arquitetura do sistema operacional (SO) e do *hardware* onde este agente está inserido. Dentro das suas limitações ligadas ao *hardware* e ao SO, o Agilla fornece uma solução de alta performance para o tipo de problema está sendo resolvido pela sua arquitetura. É possível, com a arquitetura do Agilla, ter agentes móveis e customizados ainda contendo um tamanho muito pequeno, diminuindo assim o tamanho das mensagens trocadas entre os agentes, e, consequentemente, aumentando a vida útil da bateria do nó sensor. O Agilla permite a migração de agentes com ou sem carregar o contexto do agente durante a migração, que são chamadas de *hard migration* e *soft migration* respectivamente.

O Agilla introduz conceitos que são úteis no desenvolvimento deste trabalho. A migração de agentes em RSSFs onde são carregados o escopo da tarefa é um conceito importante e de onde foram tiradas algumas ideias para o desenvolvimento deste trabalho. O conceito de *middleware* também tem um papel importante, pois ele cria um ambiente abstrato para o desenvolvimento dos agentes, fazendo que a programação do sistema seja simples para o usuário. Este trabalho também utiliza conceitos do modelo de agente do Agilla onde cada nó é capaz de suportar vários

agentes se comunicando e ainda cada agente conhece os serviços oferecidos pelos outros agentes.

5.2 Agentes em Hardware

Diversos trabalhos se propõe a utilizar uma arquitetura reconfigurável em *hardware* para a implementação de agentes. Em (SCHNEIDER; NAGGATZ; SPALLEK, 2007) é desenvolvida uma arquitetura de agentes em hardware baseada em modelos BDI. É definida uma camada em software para fazer a gerência do ambiente, a qual é responsável por controlar a comunicação entre os agentes em hardware, coordenar o trabalho cooperativo entre eles, gerenciar as propriedades dos agentes bem como monitorar o hardware procurando possíveis falhas. É uma interface com o usuário escrita em uma linguagem de alto nível (Java) que serve como um gerenciador de agentes. A arquitetura proposta consiste em um microcontrolador (μC) implementado em lógica programável (PL) dentro do FPGA ligado diretamente aos módulos que implementam o agente. A Figura 18, adaptada de (SCHNEIDER; NAGGATZ; SPALLEK, 2007), mostra esta arquitetura.

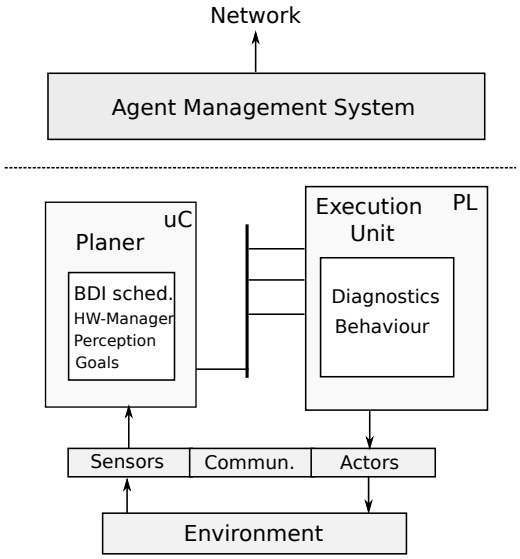

Figura 18: Arquitetura de uma proposta de agentes em hardware (SCHNEIDER; NAGGATZ; SPALLEK, 2007).

A estruturação dos agentes em *hardware* é feita de forma horizontal utilizando-se de diferentes unidades funcionais separadas por camadas. Essas camadas são divididas em nível físico, no processamento das informações dos sensores, nível de conhecimento, onde são processadas as metas dos agentes e em nível social com a cooperação entre os agentes. Os agentes se comunicam com a interface de gerência que controla a organização destes agentes na arquitetura. Habilidades sensoriais e cognitivas bem como as reativas são características dos agentes e implica em se ter interfaces especiais bem como blocos de processamento de sinal dentro do agente. Para fazer o ajuste automático, foi implementado um núcleo de um microcontrolador A8M que integra os módulos desenvolvidos em lógica programável.

Outra arquitetura é a proposta em (MENG, 2006) que define uma plataforma de agentes em *hardware* utilizada em sistemas com requisitos de tempo real. Neste

trabalho, é proposto um modelo de agente do tipo BDI com uma interface de comunicação chamada *On Demand Message Passing* (ODMP), que é a proposta central deste trabalho pois os resultados mostram uma redução na complexidade dos agentes devido ao uso desta interface. O modelo BDI proposto pode ser visto na Figura 19 sendo que existe uma correlação entre *beliefs* e *desires* pois eles influenciam um ao outro e ambos influenciam *intentions*. O modelo inclui 3 portas externas utilizadas para comunicação entre os agentes, controle e também entrada e saída de dados. A porta de controle é utilizada para o sincronismo dos agentes com o sistema, a porta de entrada e saída é utilizada para trocar informações com o ambiente onde os agentes estão inseridos e finalmente a porta para comunicação entre agentes é utilizada para a troca de informações diretamente com outros agentes do sistema.

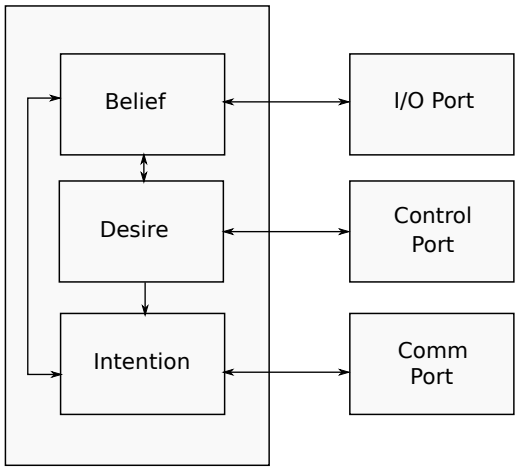

Figura 19: Agentes BDI em hardware.

Agentes são geralmente projetados para um propósito específico e eles podem fazer uma ou até muitas tarefas em paralelo. Caso os agentes devam executar mais tarefas em paralelo, é possível ou aumentar a complexidade dos agentes, o que pode fazer aumentar o esforço de desenvolvimento, ou ainda fazer com que eles trabalhem cooperativamente. Geralmente, uma tarefa complexa de tempo real pode ser construída como um conjunto de agentes, onde cada agente tem a sua própria intenção e seu próprio conjunto de metas. Visto que os agentes podem trabalhar de maneira assíncrona, mensagens somente serão transmitidas por demanda, o que leva ao desenvolvimento do protocolo ODMP proposto neste trabalho (MENG, 2006). Este protocolo é basicamente uma fila do tipo *First-In/First-Out* (FIFO) com prioridades, onde as mensagens marcadas como urgentes são colocadas no início da fila.

Para a implementação do módulo de reconfiguração, foi criado um módulo chamado de reconfiguração virtual de agente (VAR). Este método consiste de um número virtual de estados que cada agente pode estar durante seu tempo de vida. Estes estados são definidos pelos *bitstreams* que são carregados na memória do FPGA na sua inicialização. Assim, neste esquema de configuração do FPGA os registradores são fixos em tempo de execução, e o dispositivo é reconfigurado somente selecionando um estado ativo no multiplexador de entrada. Este tipo de abordagem reduz significativamente a latência devido à reprogramação do módulo.

Um estudo de caso foi feito sobre o sistema de visão de um robô móvel navegando em um ambiente com obstáculos e se deslocando entre dois pontos em uma sala. Foi

feita uma proposta de *codesign* dos agentes em *hardware* e em *software* sendo que o projeto de cada módulo foi feito separadamente pois o foco do trabalho está na arquitetura do sistema multi-agente, e não nas tarefas que os agentes executam.

5.3 Trabalhos diversos

Existem inúmeros trabalhos que fazem o uso do conceito de agentes móveis tanto em *software*, utilizando algum *framework* padrão, tanto como em *hardware*, fazendo uso da reconfigurabilidade dinâmica em FPGAs. Em (NAJI, 2005) é feita a fusão de sensores utilizando-se agentes em FPGA sendo que cada *bitstream* corresponde a um agente que pode conter diversos tipos de comportamento. Os agentes podem ser alterados em tempo de execução dada a reprogramação dinâmica oferecida pelos FPGAs. Em (BENSO et al., 2005) são criados agentes móveis em *hardware* na criação de sistemas que são capazes de se auto monitorarem e se auto diagnosticarem. Os agentes em *hardware* são utilizados para trocar-se o comportamento do sistema para regiões onde existem falhas.

O uso de Jade como arquitetura dos sistemas multi-agente se aplica em diversas áreas de aplicação. Em (GOMEZ-GUALDRON; VELEZ-REYES; COLLAZO, 2007) é proposto um protótipo de sistema multi-agente para reconfiguração de sistemas elétricos de potência. Agentes são utilizados para reconfigurar o sistema elétrico de potência quando alguma falha ocorre. Já em (FILGUEIRAS; LUNG; OLIVEIRA RECH, 2012) é proposto uma extensão ao JADE para suportar escalonamento de tempo real sobre aplicações rodando usando a plataforma do Jade. Ter a possibilidade de se executar tarefas em tempo real aumenta a gama de aplicações possíveis utilizando-se a plataforma do Jade. Em (SAAIM et al., 2005) é proposta uma arquitetura de agentes utilizada para se fazer reconhecimento distribuído de voz utilizando-se a plataforma do Jade.

Neste último trabalho, o uso de uma arquitetura de agentes móveis é utilizada devido a motivação da diminuição do tráfego de dados proporcionado pelo uso de agentes. Diferentes campos de aplicação se beneficiam da plataforma do Jade, o que demonstra que ela é uma plataforma versátil. A interoperabilidade do Jade devido ao uso de mensagens compatíveis com as normas da FIPA facilita e também motiva o uso deste *framework*.

5.4 Conclusão

Neste capítulo foram analisadas arquiteturas que utilizam o conceito de agentes móveis tanto em *software* quanto em *hardware* com o objetivo de relacioná-las com a proposta deste trabalho. O Agilla é uma arquitetura que oferece diversos aspectos que podem ser aproveitados conceitualmente neste trabalho. O conceito de agentes em *hardware* já foi introduzido em alguns trabalhos sendo que idealmente é utilizada a reconfiguração dinâmica de FPGAs para a troca de agentes em *hardware*. Conforme mencionado na Seção 4.4, o Jade é uma plataforma de desenvolvimento de agentes que permite a criação de agentes móveis com o comportamento desejado. O Jade é de fácil uso pois não requer conhecimentos muito aprofundados dos conceitos de agentes, criando uma interface em java que encapsula os agentes. O Jade ainda implementa uma linguagem de comunicação entre agentes normatizada pela FIPA. A Tabela 2 mostra as principais características apresentadas por cada

trabalho analisado e que serão aproveitadas neste trabalho.

Tabela 2: Principais características dos sistemas analisados.

Arquitetura	**Características**
Agilla	Modelo agente móvel para RSSFs. Alta performance.
Agentes em hardware	Reconfiguração dinâmica. Reconfiguração parcial. Agentes móveis.
Jade	FIPA-ACL. Migração de agentes. Abstração do hardware.

O uso de arquiteturas reconfiguráveis dinamicamente permite que se altere o comportamento de um sistema durante o seu uso. As arquiteturas reconfiguráveis em *software* já permitem este tipo de configuração, o que garante a flexibilidade dos sistemas. Ter disponível este tipo de configuração a nível de *hardware* aumenta a quantidade de aplicações que podem se beneficiar da reconfiguração do sistema.

Entre os trabalhos analisados que utilizavam reconfiguração dinâmica do *hardware* para a troca de agentes, normalmente utiliza-se um microcontrolador ou microprocessador para a reprogramação do FPGA através das portas internas de configuração. Nos trabalhos analisados, não existe a migração de agentes de *software* para *hardware* e vice-versa bem como a comunicação só é feita entre agentes do mesmo tipo. Na proposta deste trabalho, que é descrita no Capítulo 6, são levadas em consideração as arquiteturas aqui analisadas para a criação de um modelo de agente que seja capaz de se comunicar transparentemente tanto com agentes em *software* quanto como com agentes em *hardware*, bem como estes agentes são capazes de migrar entre as diferentes plataformas carregando o escopo da sua tarefa.

6 PROPOSTA DO TRABALHO

Redes de sensores sem fio (RSSFs) podem conter diversos dispositivos coletando, processando informações e comunicando-se entre si. As RSSFs podem ser ainda heterogêneas, combinando nós com mobilidade física e nós estáticos. Este tipo de nó sensor pode tornar-se interessante para diversas aplicações, como por exemplo vigilância, agricultura de precisão, monitoramento de linhas de transmissão entre outras (FREITAS, 2012). A combinação de nós sensores estáticos com nós móveis aumenta a flexibilidade da rede pois confere mobilidade física aos nós podendo levá-los para missões específicas. Neste contexto, veículos aéreos não-tripulados (VANTs) são exemplos de utilização para nós fisicamente móveis.

O desenvolvimento de aplicações usando RSSF pode ser baseado no uso de sistemas multi-agente (SMA) pois este paradigma é naturalmente distribuído, e o uso de SMA em RSSFs vem sendo cada vez mais difundido (FOK; ROMAN; LU, 2005b). O uso de agentes em RSSFs possibilita, por exemplo, VANTs receberem missões moldadas como um agente. Este tipo de de abordagem permite o envio de diferentes agentes para VANTs que estão localizados remotamente. Além disso, o uso de agentes permite a transferência de agentes entre um VANT e outro, possibilitando a continuação da execução da tarefa no segundo VANT. Isso pode ser necessário em casos onde o VANT está entrando ou saindo da rede por algum motivo, como por exemplo a execução de alguma missão. Estes nós sensores podem estar equipados com um *hardware* reconfigurável como um ambiente alternativo ao processador, o que aumenta o poder computacional e a flexibilidade do nó.

Neste contexto, este trabalho propõe uma arquitetura de um agente reconfigurável, que é capaz de ser executado como um agente puramente em *software*, bem como um agente puramente em *hardware*. A arquitetura do Jade (TILAB, 2012) é utilizada como um *framework* de desenvolvimento de agentes que implementa a camada de comunicação entre eles. Embora tenha sido utilizado o Jade, outras arquiteturas de desenvolvimento de agentes podem ser utilizadas no contexto deste trabalho pois boa parte dos conceitos apresentados neste trabalho se aplicam também a outros *frameworks*. Este trabalho também se propõe a propiciar que a comunicação entre os agentes puramente em *software* ou *hardware* seja feita de forma transparente. Desta forma, os agentes não necessitam saber da natureza dos outros agentes enquanto trocam as mensagens, o que faz com que os agentes reconfiguráveis possam ser inseridos na rede de uma forma transparente.

Um nó sensor típico é composto por processador, memória, transmissor/receptor de radiofrequência, uma unidade de energia, tipicamente uma bateria, e sensores. Para suportar-se o uso de agentes em *hardware*, está sendo proposta a inserção de um dispositivo reconfigurável, como por exemplo um FPGA, em alguns nós da

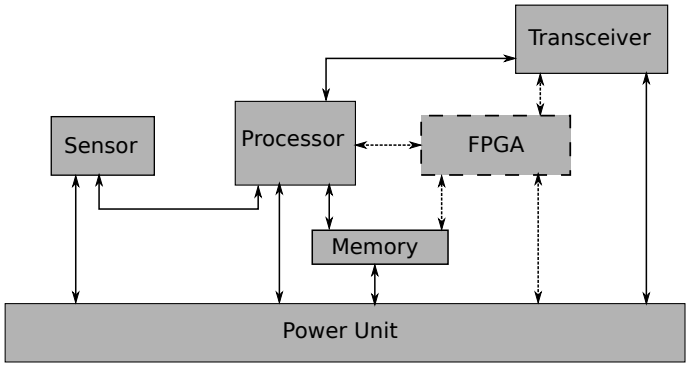

Figura 20: Arquitetura de um nó sensor modificada.

RSSF. Este dispositivo deve ser reconfigurável para que se tenha a flexibilidade de reconfiguração do agente em tempo de execução sem a necessidade de trocar fisicamente o hardware do nó. Os FPGAs aparecem como uma tecnologia que fornece este tipo de interface de *hardware* programável à arquitetura. A Figura 20 mostra esta arquitetura com os dispositivos mencionados.

A arquitetura proposta pode ser dividida em três grandes partes, que podem ser vistas na Figura 21. A camada de aplicação, que é a camada mais alta deste trabalho, é executada sobre uma máquina virtual java (JVM) onde rodam os agentes em *software* e que neste trabalho utilizaram o *framework* do Jade para serem desenvolvidos. O Jade provê uma arquitetura de agentes móveis que usam uma linguagem de comunicação normatizada pela FIPA, o que agrega valor a este trabalho pois faz com que ele seja facilmente acoplado a outras RSSFs. O Jade também facilita a migração dos agentes, fornecendo no próprio *framework* métodos de fácil uso que possibilitam a migração e o clone dos agentes em outros nós.

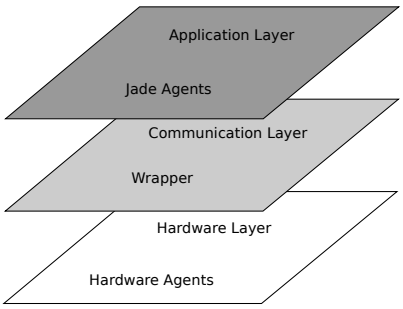

Figura 21: Arquitetura em camadas utilizada no nó sensor.

A camada do meio é descrita pelo Wrapper de agentes, que é uma camada de implementações em *software* que serve como abstração do *hardware* para os agentes do Jade. Esta camada é responsável por fazer o controle de acesso aos dispositivos, garantindo a exclusão mútua de acessos concorrentes e também fornecendo uma interface para configuração e transferência de informações ao FPGA. O Wrapper é uma entidade que encapsula as requisições provindas dos agentes em *software* e as mapeia para dentro dos agentes implementados em *hardware*. Esta camada ainda é responsável por manter a comunicação dos agentes em *hardware* transparentes para os agentes em *software*, fazendo com que os agentes se comuniquem transparentemente.

A camada mais interna representa os agentes implementados em *hardware*. Estes agentes são capazes de migrar entre diferentes nós, carregando seu *bitstream* e contexto, através da plataforma fornecida pela camada mais superior desta arquitetura. Os agentes em *hardware* podem ser configurados dinamicamente através de requisições das outras camadas. O processador principal é responsável por configurar o FPGA com os agentes e a camada do Wrapper copia o contexto dos agentes entre as camadas de *software* e de *hardware*.

São propostos alguns mecanismos para tomada de decisão na configuração do FPGA. De uma maneira geral, os agentes podem ser configurados pelo usuário, através de um comando enviado ao agente onde se quer que seja executado um agente em *hardware* para finalmente este dispositivo receber a implementação em *hardware* do agente. Os agentes também podem estar dotados de uma certa inteligência, sendo eles mesmos capazes de perceber o ambiente onde se encontram. Assim, estes agentes requisitam a sua versão em *hardware* para finalmente conseguirem configurar o FPGA com o *bitstream* desejado.

6.1 Proposta de Arquitetura do Agente Reconfigurável

Este trabalho utiliza a plataforma do Jade, descrita no capítulo 4 para desenvolver o sistema multi-agente pois esta plataforma implementa a linguagem de comunicação entre agentes (ACL) já compatível com com as normas da FIPA. Além disso, esta plataforma vem sendo implementada e testada há mais de dez anos, o que garante uma estabilidade maior a este trabalho. Como pode ser visto no capítulo 5, diversos trabalhos já utilizam o Jade como plataforma multi-agente, o que mostra que a plataforma é funcional.

Algumas partes da estrutura do Jade foram desenhadas para serem executadas em um processador, visto que elas dependem de informação sequencial, como é o caso do *behavioural scheduling* ou ainda do *life cycle manager*. Este trabalho não se propõe a reimplementar a integralmente a arquitetura do Jade dentro de um FPGA pois isso levaria à dependência deste trabalho ao *framework* do Jade, uma vez que parte da implementação do agente estaria em *hardware*. A utilização do Jade como sendo uma plataforma puramente de *software* faz com que não haja esta dependência, pois os agentes em *hardware* não conhecem a plataforma de *software* que está sendo executada um nível acima.

O Jade já implementa a comunicação entre agentes e também faz com que os agentes conheçam os serviços oferecidos pelos outros agentes através do *main container*. Agentes reconfiguráveis necessitam comunicar-se com os agentes em Jade de uma forma transparente para manter o sistema sendo executado independentemente da natureza do agente. Como uma decisão de projeto, optou-se por utilizar-se dessa estrutura de comunicação em software e implementar somente o comportamento do agente em hardware. Com isso ganha-se a transparência com outras plataformas que são compatíveis com as normas da FIPA.

O desenvolvimento dos agentes em *hardware* e *software* é feito neste trabalho de forma independente. Não foi criada nenhuma ferramenta que faça o desenvolvimento conjunto das arquiteturas de *hardware* e de *software* pois este tipo de abordagem tiraria o foco deste trabalho, que está no estudo de maneiras de manter a transparência na comunicação entre os agentes e ainda promovendo a migração destes agentes entre as diferentes plataformas. O estudo e implementação dos agentes está

sendo feito inicialmente de uma forma manual, como pode ser visto no Capítulo 8. Os agentes em *hardware* são criados em um espaço de memória padrão para que os agentes em *software* tenham uma maneira única de configurar os agentes em *hardware*, ou seja, para que os mecanismos que fazem o reconhecimento dos agentes utilizem sempre a mesma região de endereços. Os comportamentos específicos de cada agente são implementados em outra região de memória reservado para este fim e o significado desta região vai depender especificamente de cada aplicação.

Os comportamentos fornecidos nativamente pelo *framework* do Jade, que são descritos na Seção 4.4, podem ser utilizados para atender requisitos de uma aplicação específica, mas também novos comportamentos podem ser criados e isso precisa ser analisado caso a caso. Os agentes em *software* têm métodos específicos para descobrir se a migração para o *hardware* é possível. Estas abordagens são descritas na Seção 6.3.

O Agente em *hardware* é desenvolvido de forma que ele conheça parte das estruturas implementadas em software. Para isso, é necessário que se mantenha uma estrutura padrão em software para encapsular os dados dos agentes e mapear estes dados para dentro da estrutura de *hardware*. É necessário que se tenha uma interface responsável por transferir as informações entre as duas estruturas de forma que nada se altere na comunicação entre os agentes. Esta interface é chamada neste trabalho de *wrapper* de agentes, pois ele concentra as requisições dos agentes em Java para acessar o *hardware* e também encapsula os dados que vêm do *hardware* para dentro de uma estrutura que é conhecida pelos agentes Jade.

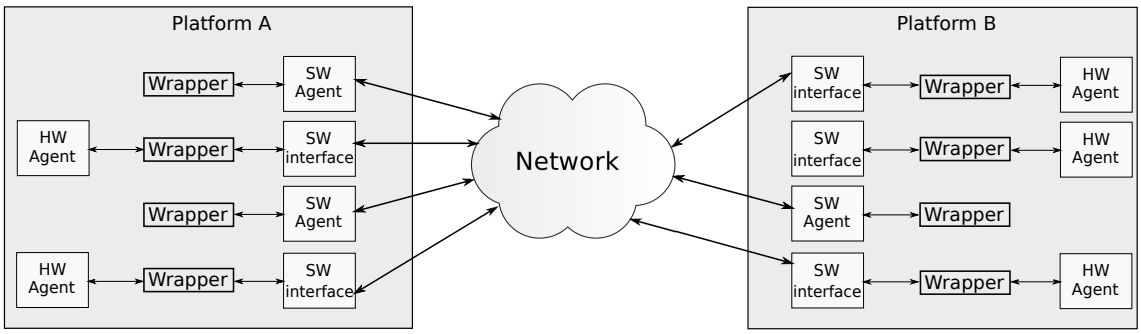

Figura 22: Comunicação entre agentes em diferentes plataformas.

A Figura 22 mostra plataformas de agentes se comunicando com diferentes implementações de agentes em *software* e em *hardware*. Os agentes em *hardware* se comunicam através do canal de comunicação criado pelo Jade com mensagens no padrão da FIPA. As diferentes plataformas não necessariamente necessitam ter todas as versões, em *hardware* e em *software* do agente. A migração do agente entre diferentes plataformas é feita através da arquitetura fornecida pelo Jade enquanto que a migração para o *hardware* é feita com auxílio do Wrapper de agentes criado neste trabalho. Os agentes em *software* utilizam o Wrapper para acessar o *hardware* disponível na plataforma. Já os agentes em *hardware* são acessados através de uma interface em *software* que por sua vez acessa o Wrapper que faz a comunicação efetiva com os agentes.

6.2 O Wrapper de Agentes

O Wrapper de agentes é uma camada de abstração criada neste trabalho com o intuito de encapsular as requisições dos agentes ao *hardware* do nó sensor. Todo controle de acesso a dispositivos é feito por esta camada em baixo nível que concentra as requisições de todos os agentes e faz o controle de fluxo de acesso aos módulos do nó. Como o Wrapper concentra as requisições de acesso ao *hardware*, esta camada também é responsável por implementar as funções que fazem a configuração dinâmica do FPGA a partir do processador.

Do ponto de vista dos agentes em *software*, o Wrapper provê serviços de acesso aos dispositivos, acesso de leitura e escrita aos módulos implementados no FPGA e serviços de reprogramação do FPGA. Os agentes enxergam este Wrapper como um provedor de serviços locais a cada nó, e é através desta camada que os agentes são capazes de descobrir sobre a existência do FPGA no nó sensor que eles estão inseridos. A Figura 23 mostra o Wrapper do ponto de vista dos agentes em *software*.

Os agentes enxergam uma camada de acesso ao FPGA, com funções de programação, leitura e escrita e também outras funções quaisquer para acessar algum outro dispositivo que esteja conectado diretamente ao processador, como algum sensor por exemplo, ou ainda na escrita de alguns sinais para atuar no sistema onde os agentes se encontram. Na Figura 23, o bloco *Sw Agent* representa todo o agente em *software*, que utiliza o Wrapper como uma camada de acesso ao *hardware*, seja para a leitura de sensores ou para a escrita de alguma saída que está conectada diretamente ao processador. O Wrapper tambem acessa diretamente o FPGA, que pode conter outras implementações que não necessariamente agentes. Desta forma, os agentes em *software* são capazes de acessar os dispositivos implementados em lógica programável.

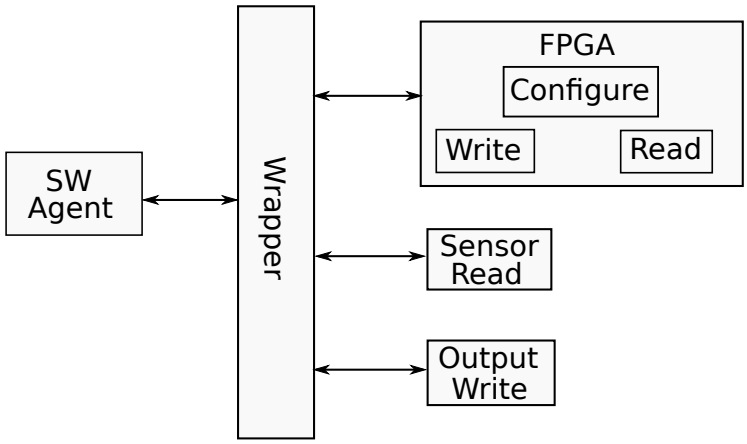

Figura 23: Serviços oferecidos pelo Wrapper aos agentes em software.

Do ponto de vista dos agentes em *hardware*, o Wrapper fornece os serviços relacionados à comunicação com outros agentes. Quando um agente em *hardware* deseja se comunicar com outro agente, este deverá sinalizar a camada de *software* de alguma maneira. Isso pode ser feito em tempo de implementação do agente, associando-se um sinal do agente a algum tipo de sinalização para o processador. O processador pode ficar monitorando as alterações neste sinal que pode servir como gatilho da operação de encapsulamento das informações para envio a outro agente. Esta tarefa ficará sendo executada na camada do Wrapper.

A interpretação do Wrapper do ponto de vista do agente em *hardware* depende muito da aplicação. Caso o agente seja puramente reativo, o Wrapper pode ser visto como uma simples interface de onde chegam requisições e para onde o agente deve responder. Diversos serviços podem ser implementados dentro do Wrapper para que ele forneça o suporte necessário aos agentes em *hardware*. A Figura 24 mostra o Wrapper quando analisado do ponto de vista dos agentes em *hardware*.

O agente em *hardware* enxerga no Wrapper uma interface de comunicação, que é a camada fornecida pelo Jade de comunicação com os outros agentes. O Wrapper também faz a interface de requisições de acesso ao *hardware* que é feita pelas camadas superiores.

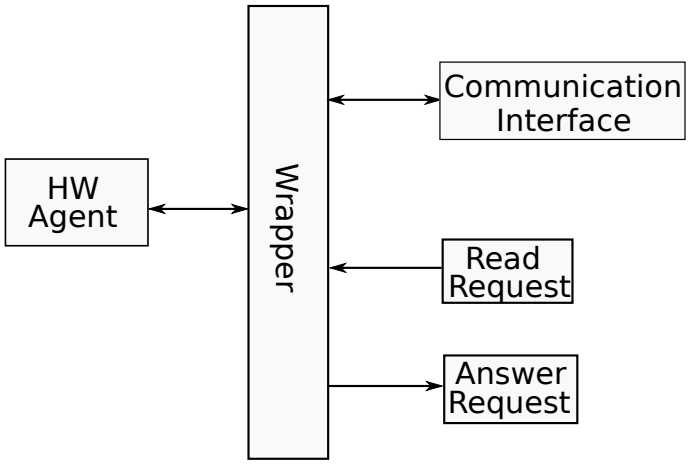

Figura 24: Serviços oferecidos pelo Wrapper aos agentes em hardware.

O Wrapper funciona como uma camada de abstração das informações que são provenientes de outros agentes. Em geral, ele vai aplicar um *parser* nas mensagens que chegam no formato ACL e mapear para o agente em *hardware* somente o que for relevante para o comportamento do agente implementado. Isso faz com que se tenha menos *overhead* na implementação do agente em hardware, diminuindo seu tamanho, e, consequentemente, simplificando a sua implementação.

6.3 Abordagens de Configuração dos Agentes em Hardware

A reconfiguração dinâmica dos agentes em *hardware* requer a reprogramação do FPGA pelo processador principal, e se pode ter diversas maneiras de fazer com que os agentes sejam executados em *hardware*. Primeiramente, quando um agente migra pra uma plataforma, ele precisa ter um artifício de saber se existe um FPGA capaz de executar o comportamento do agente. Isso serve para garantir que o agente pode continuar a sua execução em *software* sem a possibilidade de ser migrado para o *hardware*. Por outro lado, saber da existência de um *hardware* capaz de executar o agente também serve como motivação para a migração do agente para o FPGA.

Uma primeira abordagem, e a talvez a mais simples delas, é fazer com que os agentes forneçam serviços ao usuário do sistema que informem da possibilidade de migração do agente entre *software* e *hardware*. Estes serviços precisam ser implementados pontualmente dentro de cada agente, com métodos acessíveis à gerência do Jade, assim o usuário será capaz de escolher através da interface gráfica, que pode ser vista na Figura 25, o momento da migração do agente. Este serviço já

é nativamente oferecido pela plataforma do Jade para a migração de agentes entre diferentes plataformas de software.

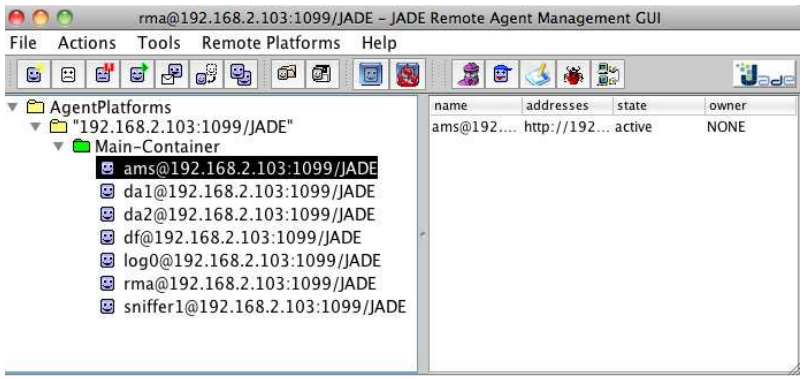

Figura 25: Interface gráfica do Jade para controle dos agentes.

Uma abordagem um pouco mais elaborada é fornecer uma certa inteligência aos agentes para que eles possam tomar atitudes de configurar o FPGA quando necessário. Para isso, criam-se registradores específicos na versão do agente em *hardware* que guardam uma *string* indicando o nome do agente que está sendo executado. Uma vez que a versão em *software* consegue ler esta variável e ela é igual ao valor que ela espera, então o agente sabe que ele pode migrar para o *hardware*. O fluxograma descrito na Figura 26 mostra o algoritmo criado para a tomada de decisão de migração do agente de *software* para *hardware*.

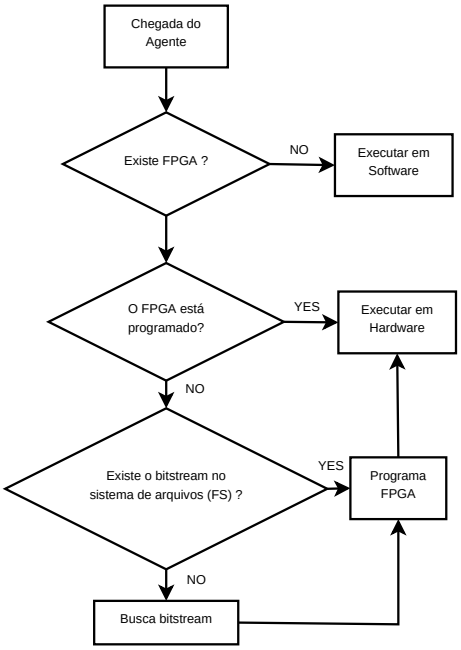

Figura 26: Fluxograma para tomada de decisão de migração dos agentes.

Uma vez que o agente migra para um nó, ele primeiramente verifica a existência de um FPGA no dispositivo, caso não exista, ele começa a sua execução em software. Quando existe um FPGA no nó-sensor, o agente começa o processo de migração para o *hardware*. Caso o FPGA já esteja programado, o agente é migrado para o *hardware*, porém, nos casos onde o FPGA ainda não se encontra programado com o *bitstream*

do agente, é necessário que se programe o FPGA. Para isso, inicialmente o agente procura pelo *bitstream* em seu sistema de arquivos (FS) ou até mesmo em outros dispositivos. Uma vez que o agente encontra o *bitstream*, ele programa o FPGA e inicia a sua migração para o *hardware*.

Em ambas abordagens o agente em *hardware* é programado dinamicamente pelo processador principal do dispositivo sem a necessidade do uso de conexões do tipo JTAG por exemplo. Com este tipo de arquitetura, é possível fazer com que os agentes em *hardware* migrem pelos nós como se fossem agentes em *software*. Para um melhor aproveitamento do FPGA, é sugerido o uso da programação parcial do dispositivo. Com isso, os agentes que estavam sendo executados no *hardware* não são interrompidos para a substituição do *bitstream*, e ainda o tempo de configuração é menor, devido ao tamanho do *bitstream* ser menor.

O uso da programação parcial do FPGA aumenta a versatilidade e a flexibilidade da arquitetura pois insere a possibilidade de se ter diversos módulos de *hardware* reconfiguráveis na arquitetura, como pode ser visto na Figura 27. A arquitetura divide-se em uma região estática e uma região reconfigurável. A região estática mantém um único *bitstream* que só é alterado na reprogramação completa do FPGA. A região reconfigurável pode ser alterada em tempo de execução, sendo que os módulos que são implementados nesta região são independentes, ou seja, a reprogramação de um módulo não afeta o funcionamento de outro módulo.

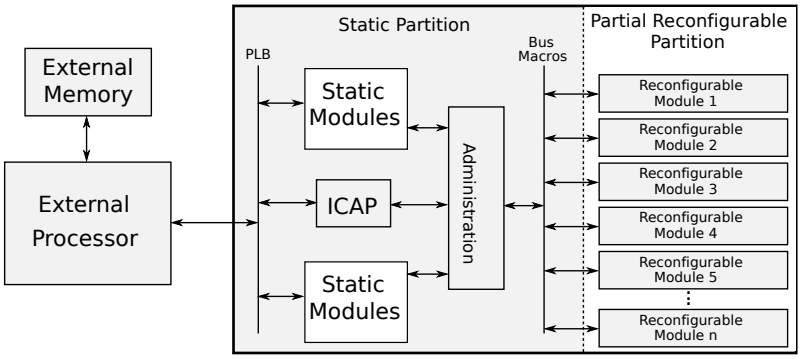

Figura 27: Arquitetura de programação parcial do FPGA.

A proposta inclui o uso de *Peripheral Local Bus* (PLB) para interconectar o processador aos diversos blocos implementados no FPGA. Neste barramento também é conectado o *Internal Configuration Access Port* (ICAP) que provê mecanismos que permitem a alteração dos módulos da região reconfigurável durante em tempo de execução. A Xilinx oferece módulos que facilitam a comunicação entre as regiões estática e reconfigurável, que são chamados de *Bus Macros*. Entre estes módulos ainda é esperado um administrador dos módulos de *hardware*, que é responsável por multiplexar os diversos sinais entre as duas regiões.

Assim, é possível transferir os *bitstreams* que encontram-se em uma memória externa para as regiões reconfiguráveis. O processador externo busca as informações na memória e acessa o FPGA para reprogramá-lo, tanto na região estática quanto na região reconfigurável. Este tipo de proposta de arquitetura ainda é expansível, podendo-se configurar processadores embarcados no FPGA, também chamados de *soft-processors*, para o controle do acesso à memória externa, eliminando assim o uso do processador externo somente com o intuito de reprogramar os módulos internos do FPGA.

A utilização de reprogramação parcial do FPGA independe se é o agente quem está decidindo migrar para o *hardware* ou se este comando provém do usuário do sistema. Esta abordagem está inserida na proposta deste trabalho com os seguintes objetivos:

- Diminuir o tempo de reconfiguração do FPGA.
- Aumentar a flexibilidade da arquitetura.
- Prover mecanismos de substituição de alguns agentes sem afetar demais agentes no sistema.

6.4 Resumo

Este trabalho se propõe a projetar uma arquitetura reconfigurável que suporte a migração de agentes móveis entre plataformas de *hardware* e *software* em RSSFs. Os nós da RSSF podem ser fisicamente móveis, quando acoplados a um VANT por exemplo, o que confere uma maior flexibilidade à rede. A mobilidade física dos nós sensores aliada à mobilidade dos agentes que estão sendo executados na RSSF aumenta a gama de aplicações possíveis que podem se beneficiar deste tipo de arquitetura. O nó sensor ainda pode contar com um dispositivo que permita a reconfiguração em *hardware*, como por exemplo um FPGA. Isso permite a reprogramação dinâmica do FPGA presente no nó com um *bitstream* que representa um agente que estava sendo executado na rede. Com este tipo de possibilidade, aliada ao fato do transporte do contexto de execução do agente para dentro da arquitetura de *hardware* resulta em uma arquitetura que suporta a migração de agentes antes executados em *software* passando a serem executados em *hardware*, ou vice-versa.

7 DETALHAMENTO DA IMPLEMENTAÇÃO

Este capítulo descreve a implementação da arquitetura proposta. Aqui são descritos com maiores detalhes as camadas que descrevem as aplicações descritas neste trabalho. Fazendo-se uma análise *top-down*, a arquitetura pode ser dividida em três grandes partes. A primeira são os agentes móveis e a máquina virtual java que é usada como camada de abstração do sistema. Os agentes móveis estão utilizando o *framework* do Jade como base e então os agentes desta camada são implementados em java.

Após, tem-se o sistema operacional e seus *drivers* que fazem a comunicação com outros dispositivos, tais como sensores e os módulos implementados no FPGA. Toda comunicação dos agentes que estão sendo executados em *hardware* e em *software* é feita por uma camada de abstração no sistema operacional, chamada de Wrapper, a qual é responsável por, entre outras coisas, mapear os dados entre os agentes em *software* e em *hardware*. O Wrapper cria uma camada de abstração do *hardware* para os agentes da camada superior e inferior pois ele também cria mecanismos de acesso a um meio comum, garantindo a exclusão mútua de acessos concorrentes.

A camada mais inferior é descrita pelos agentes implementados em *hardware*. A arquitetura proposta prevê o uso de um espaço de memória onde os agentes serão implementados. À medida que eles vão sendo criados, este espaço vai sendo preenchido com a descrição de cada agente. A arquitetura não prevê a implementação automática dos agentes em *hardware*, mas sim em prover mecanismos para que estes agentes sejam descritos e ainda garantir que eles se comuniquem com outros agentes de forma transparente bem como a migração destes agentes para outras plataformas.

Todo desenvolvimento deste trabalho foi feito em uma placa dedicada com as seguintes caracterísiscas principais:

- FPGA Virtex 6 LX130.
- Processador PowerPC core e300.

Foi utilizada mais de uma placa de desenvolvimento para a validação do trabalho. Durante os testes foi também utilizado um PC como plataforma para execução dos agentes. As placas de desenvolvimento continham os agentes sendo executados tando em *software* quanto os agentes sendo executados em *hardware* e foram utilizadas simultaneamente ao uso do PC na migração dos agentes entre as diferentes plataformas. Foi compilada uma versão de Linux embarcado, Kernel 2.6.32 juntamente com o Busybox, que contém um conjunto de ferramentas Unix em um só aplicativo. A máquina virtual Java utilizada foi a Oracle JSE 1.6.29 enquanto que

a versão do Jade utilizada foi a 4.1. Maiores detalhes da implementação podem ser encontrados no apêndice E.

7.1 Arquitetura de Agentes

A arquitetura de agentes proposta neste trabalho contém tanto agentes em *software*, quanto agentes em *hardware*. A proposta do trabalho inclui a comunicação transparente entre estes dois tipos de agentes bem como a migração destes agentes entre as plataformas de *software* e *hardware*. Um mecanismo de interfaceamento destes dois tipos de agentes foi proposto como uma camada de abstração que ainda serve como gerenciador dos recursos em *hardware*. A seguir, serão detalhadas as implementações das arquiteturas dos agentes de *software* e de *hardware*.

7.1.1 Agentes em Software

Os agentes implementados pelo Jade seguem um padrão simples. Em geral, os agentes estendem a classe *Agent* do Jade, e para implementar um simples agente basta seguir algumas diretrizes básicas do *framework*. A Listagem 7.1 mostra um exemplo simples de utilização onde um agente é definido bem como seu comportamento. Neste exemplo, é criado um simples agente que é executado ciclicamente baseado no tempo de inicialização passado para a classe, que neste caso é de 1 segundo. Este agente move-se duas vezes, uma vez quando chega em 3 segundos e outra quando chega aos 10 segundos, e aos 15 segundos de execução ele é terminado.

```
1 public class MobileAgent extends Agent {
2  protected void setup() {
3   super.setup();
4   addBehaviour(new MyTickerBehaviour(this, 1000));
5
6   System.out.println("Hello World. I am an agent!");
7   System.out.println("My LocalName: " + getAID().
       getLocalName());
8   System.out.println("My Name: " + getAID().getName());
9   System.out.println("My Address: " + getAID().
       getAddressesArray()[0]);
10 }
11 private class MyTickerBehaviour extends TickerBehaviour {
12  Agent agent;
13  int counter;
14  public MyTickerBehaviour(Agent agent, long interval) {
15   super(agent, interval);
16   this.agent = agent;
17  }
18  protected void onTick() {
19   if (counter == 3) {
20    AID remoteAMS = new AID("ams@localhost", AID.ISGUID);
21    remoteAMS.addAddresses("http://localhost:7778/acc");
22    PlatformID destination = new PlatformID(remoteAMS);
23    agent.doMove(destination);
24   }
25   if (counter == 10) {
26    AID remoteAMS = new AID("ams@localhost", AID.ISGUID);
27    remoteAMS.addAddresses("http://localhost:7778/acc");
28    PlatformID destination = new PlatformID(remoteAMS);
29    agent.doMove(destination);
```

```
30    }
31    if (counter < 15)
32     System.out.println(counter++);
33    else
34     agent.doDelete();
35   }
36 }
37}
```

Listagem 7.1: Exemplo de mobilidade de agentes com o Jade

7.1.2 Agentes em Hardware

A implementação dos agentes em *hardware* em detalhes não é de grande interesse para este trabalho. Isso pode ser visto nos estudos de caso implementados no Capítulo 8, onde foi proposto um agente FPU e utilizada uma arquitetura já implementada para a descrição do agente. Entretanto, é necessário que se mantenha o padrão de implementação entre diferentes modalidades de agentes em *hardware* para que a arquitetura continue funcionando adequadamente. Para isso, Foi criado um padrão de mapa de memória que deverá ser respeitado para qualquer agente genérico criado, sendo que existe uma região de 512 endereços de 8 bits alocada para memória de troca de dados entre os agentes, que é mostrada na Figura 28 mapeada pela faixa de endereços de 0x00 até 0x1FF.

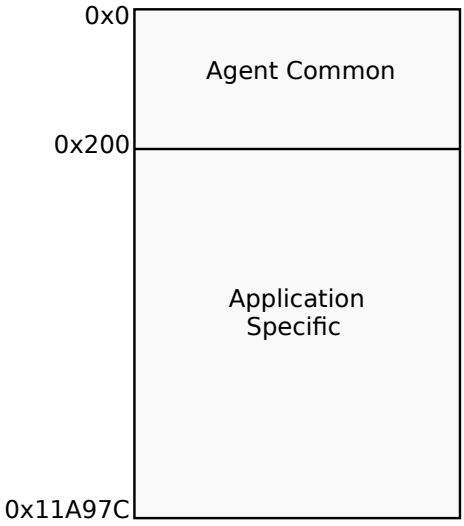

Figura 28: Mapa de memória padrão para a migração de agentes.

Esta região é reservada para os sinais em comum entre os agentes que são utilizados pela arquitetura para a identificação do agente. Por exemplo, nesta faixa de endereços deve conter a *string* que indica o nome do agente. É a partir desta *string* que o agente verifica que ele programou o *bitstream* correto. O *design* padrão espera que o agente verifique a validade da implementação em *hardware* através de uma leitura de um endereço e posterior comparação com um padrão que é conhecido pelo agente. Por exemplo, um agente Kalman poderia ter uma *string* igual a *kalman* gravada a partir do endereço $0x100$. Com isso, o agente em *software*, após a programação do FPGA, leria este endereço para comparar as duas *strings*.

A faixa de endereços mostrada na Figura 28 é decorrente do tamanho da memória interna utilizado neste trabalho. O FPGA dispõe de 9280 Mbits de memória interna,

mapeada em BRAMs, o que resulta na faixa de endereços apresentada na Figura mencionada. Com a utilização de outro FPGA esse valor pode ser alterado, mas deverá sempre garantindo uma pequena separação das informações referentes ao controle das informações correspondente aos dados dos agentes.

7.2 Implementação do Wrapper

A interação entre agentes de *hardware* e de *software* é obtida utilizando-se abstrações em diversas camadas. Primeiramente, foi criada um suporte a nível de Kernel do sistema operacional com funções de escrita e leitura bem como funções que permitem a reprogramação dinâmica do FPGA diretamente através do processador principal. O acesso ao FPGA pelo processador é feito através de um barramento chamado Localbus, onde os sinais necessários com dados, endereços, *latches* etc, são controlados pelo controlador do barramento interno ao processador. O *driver* no Kernel é responsável por receber estes valores do espaço de usuário e copiá-los para a região onde se encontra o controlador do barramento. A reconfiguração dinâmica do FPGA é feita através de pinos de GPIO do processador, onde o *bitstream* é enviado serialmente para o FPGA. A Listagem 7.2 mostra parte das funções utilizadas na programação do FPGA no espaço do Kernel. Para maiores detalhes sobre o *hardware* desta arquitetura, bem como os métodos para mapear os endereços em memória e a programação do FPGA são encontradas no Apêndice E.

```
 1
 2void fpga_pulse_prog(void)
 3{
 4 gpio_data_set(0, FPGA_PROG_PIN, 0);
 5 set_current_state(TASK_INTERRUPTIBLE);
 6 schedule_timeout(100);
 7 gpio_data_set(0, FPGA_PROG_PIN, 1);
 8}
 9
10int fpga_init_sts(void)
11{
12 char val;
13
14 gpio_data_get(0, FPGA_INIT_PIN, &val);
15
16 return val;
17}
18
19inline void config_bit(unsigned char bit)
20{
21 gpio_data_set(0, FPGA_D0_PIN, bit);
22 gpio_data_set(0, FPGA_CCLK_PIN, 0);
23 gpio_data_set(0, FPGA_CCLK_PIN, 1);
24}
25
26void fpga_program(unsigned int size, unsigned char *data)
27{
28 unsigned int i, r;
29 unsigned char bitm;
30
31 for (i = 0; i < size; i++) {
32  for (bitm = 1, r = 0; r < 8; bitm <<= 1, r++) {
```

```
33    config_bit(minor, bitm & data[i]);
34   }
35  }
36 }
37
38 int fpga_done_status(void)
39 {
40  char val;
41
42  gpio_data_get(0, FPGA_DONE_PIN, &val);
43
44  return val;
45 }
```

Listagem 7.2: Funções utilizadas na programação do FPGA.

Segundo, uma aplicação a nível de usuário que acessa a camada do Kernel é responsável por criar mecanismos para garantir a exclusão mútua entre diferentes agentes tentando acessar o mesmo dispositivo concorrentemente. Isso é extremamente necessário para evitar problemas de um processo interferindo em outro durante períodos de escrita e leitura de um mesmo espaço de memória. Toda a cópia de informações entre os agentes em *software* e *hardware* é feita através de funções de escrita e leitura que garantem que não existem acessos concorrentes ao mesmo dispositivo, o que pode ser visto na Listagem 7.3.

```
 1 typedef struct {
 2  unsigned short address;
 3  unsigned char op;
 4  unsigned char *data;
 5 } st_fpga_data;
 6
 7 int
 8 fpga_read(int fd, unsigned short add, unsigned char *data)
 9 {
10  st_fpga_data req;
11
12  bzero(&req, sizeof(req));
13
14  pthread_mutex_lock(&fpga_mutex);
15
16  req.address = add;
17  req.op = FPGA_READ;
18  memcpy(req.data, data, sizeof(data));
19
20  if (ioctl(fd, req) < 0) {
21   pthread_mutex_unlock(&fpga_mutex);
22   fprintf(stderr, "Error opening fpga for reading!\n");
23   return -1;
24  }
25  pthread_mutex_unlock(&fpga_mutex);
26
27  return 0;
28 }
29
30 int
31 fpga_write(int fd, unsigned short add, unsigned char data)
32 {
33  st_fpga_data req;
```

```
34
35 bzero(&req, sizeof(req));
36
37 pthread_mutex_lock(&fpga_mutex);
38
39 req.address = add;
40 req.op = FPGA_WRITE;
41 *req.data = data;
42
43 if (ioctl(fd, req) < 0) {
44  pthread_mutex_unlock(&fpga_mutex);
45  fprintf(stderr, "Error opening fpga for writing!\n");
46  return -1;
47 }
48 pthread_mutex_unlock(&fpga_mutex);
49
50 return 0;
51}
```

Listagem 7.3: Funções utilizadas para troca de informações com os módulos implementados no FPGA.

Finalmente, o mapeamento entre os agentes em si é feito através de uma simples máquina de estados implementada em software que é responsável por fazer a cópia dos dados do espaço dos agentes em software para o espaço dos agentes em hardware e também por inicializar a execução do agente em hardware uma vez que a migração foi concluída.

Nos exemplos implementados neste trabalho, os agentes em *hardware* já estão previamente implementados, isto é, existe em algum lugar da rede de sensores um *bitstream* do FPGA que representa o mesmo agente em *software* que está em processo de migração. Quando um agente móvel é transferido para um nó específico, a primeira tarefa atribuída a este agente é a tentativa de abertura de um *file descriptor* (FD) referente ao FPGA no nó onde este agente está situado.

A partir do retorno da função de leitura do FD, o agente é capaz de perceber a presença do FPGA e finalmente começa a procura por uma implementação da sua versão em hardware. Esta procura é feita enviando-se mensagens a outros agentes e também ao *main container*, e uma vez que o agente encontra seu *bitstream*, ele finalmente é transferido ao mesmo nó físico do agente requisitante. Após a transferência do arquivo, o agente entra em um estado onde o FPGA vai ser programado através das chamadas de programação implementadas no espaço de usuário. Quando um agente dispara a tarefa de migração para hardware, a máquina de estados mostrada na Figura 29 é executada enquanto que a Listagem 7.4 mostra a implementação desta máquina de estados dentro do código do Wrapper. Este algoritmo foi discutido na proposta deste trabalho no Capítulo 6.

```
1typedef struct {
2 unsigned int cur_state;
3 unsigned int next_state;
4 void *fpga_data;
5 unsigned int data_size;
6 unsigned int reset;
7 unsigned char migr_type;
8} st_fsm_wrapper;
9
10static int
```

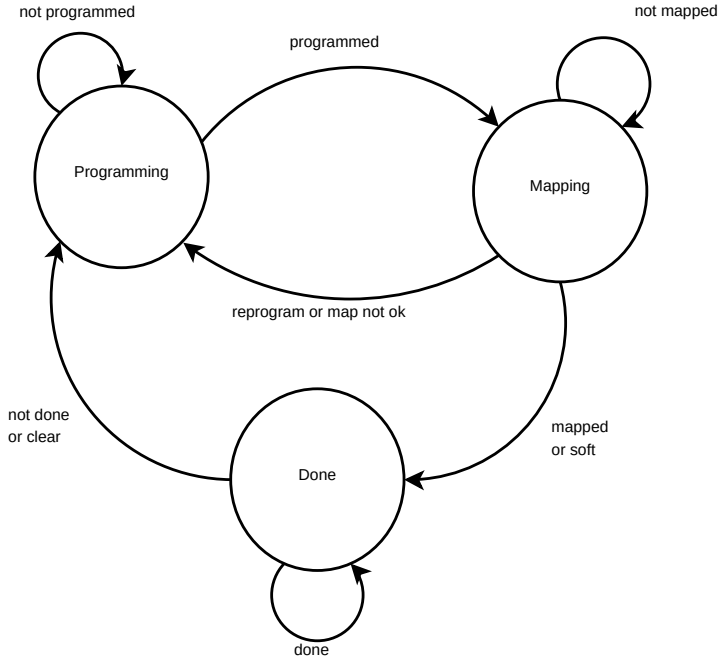

Figura 29: Máquina de estados usada na migração dos agentes.

```
11 fsm_wrapper_next_state(st_fsm_wrapper *fsm)
12 {
13  unsigned int cur = fsm->cur_state;
14
15  switch(cur) {
16   case PROGRAMMING:
17    if (fpga_program(fsm->fpga_data) < 0) {
18     printf("Error programming fpga!\n");
19     return -1;
20    }
21    fsm->next_state = MAPPING;
22    break;
23   case MAPPING:{
24    unsigned int size = fsm->data_size;
25    unsigned int i;
26
27    if (fsm->migr_type == MIGR_HARD) {
28     for (i = 0; i < size; i++) {
29      if (fpga_write_byte(&fsm->data[i], AGENT_DATA_COMMON +
           i) < 0) {
30       printf("error mapping!\n");
31       return -1;
32      }
33     }
34    }
35    fsm->next_state = DONE;
36    break;
37   }
38   case DONE:
39    if(fsm->reset)
40     fsm->next_state = PROGRAMMING;
41    break;
42
```

```
43   default:
44    printf("undefined. Error!\n");
45    return -1;
46    break;
47  }
48
49  return 0;
50 }
```

Listagem 7.4: Máquina de estados simplificada do mapeamento de dados entre agentes.

A máquina fica no estado de programação do FPGA até que a programação termine ou algum erro ocorra. Em caso de erro, um controle externo decide pela continuação das tentativas. Uma vez que o FPGA é programado com sucesso, começa-se o processo de cópia dos dados entre os agentes. A duração desta fase depende do tipo de migração, se é *soft* ou *hard* [1], pois ou se copia o estado diretamente para o agente em hardware, ou manda-se ele ser executado a partir do início e então não é necessária a cópia dos dados dos agentes.

O Wrapper se comunica com ambos agentes tanto em *hardware* quanto em *software*. Quando um agente está requisitando informação para um dispositivo mapeado em memória por exemplo, este agente envia uma requisição ao Wrapper, que é indicado pelo número 1 na Figura 30. O Wrapper por sua vez acessa o hardware e responde a requisição do agente em *software* mostrados respectivamente pelas setas 2 e 3 na Figura 30.

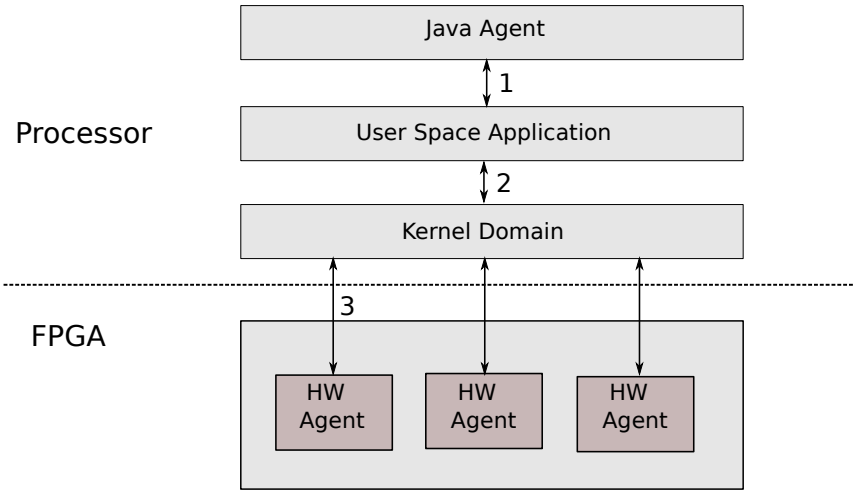

Figura 30: Fluxo de dados dentro do *wrapper*.

Vale a pena observar que a comunicação entre os agentes em Java e o Wrapper é feita através de um *socket* do tipo TCP. O Wrapper mantém na sua essência um conjunto de comandos que refletem as requisições feitas pelos agentes móveis. A Listagem B.1 mostra as principais funções implementadas no Wrapper. A função *treatClientPack()* implementa o tratamento dos comandos para novos agentes. Por exemplo, um agente que lê a temperatura do nó através de um sensor provavelmente

[1] Os termos *soft* e *hard migration* foram citados em (FOK; ROMAN; LU, 2005b) onde *soft migration* significa uma migração sem carregar o contexto enquanto que *hard migration* é um tipo de migração onde o contexto do agente migra junto com o código do agente.

implementará um método que faça uma leitura do sensor de temperatura. Este comando é mapeado pelo caso TEMP_GET na listagem mencionada.

7.3 Conclusão

A implementação dos agentes em *software* foi simplificada neste trabalho pelo uso da plataforma do Jade. O Jade implementa diversos serviços que facilitam o desenvolvimento e diminuem o tempo gasto para implementação dos agentes. O Jade implementa a comunicação entre os agentes já nos padrões da FIPA, o que garante o interfaceamento dos agentes implementados neste trabalho com outros agentes externos. O Jade ainda facilita a migração do agente entre diferentes nós carregando não só o código, mas também o contexto da tarefa que está sendo migrada.

A implementação e detalhamentos dos agentes em *hardware* não são o foco direto deste trabalho, todavia uma análise sobre os casos de uso e considerações sobre padrões de implementação foram feitas. Os agentes em *hardware* devem ser capazes de se comunicar com agentes em *software* bem como serem capazes de migrar para uma plataforma de *software* carregando seu contexto para continuar a execução na plataforma de *software*. A comunicação entre os diferentes tipos de agente se dá sobre a plataforma chamada neste trabalho de Wrapper de agentes.

O Wrapper concentra os serviços de acesso aos dispositivos de *hardware* da plataforma do nó sensor. É através desta camada que os agentes puramente em *software* conseguem programar os FPGAs e mapear o conteúdo do agente para dentro do espaço de memória que o caracteriza, mapeado para dentro do FPGA. O Wrapper ainda garante a exclusão mútua de acesso aos dispositivos através de chamadas do sistema operacional que são garantidamente atômicas, isto é, que são executadas em um único ciclo, evitando a perda do *lock* por chaveamento de contexto do escalonador. Uma implementação mais detalhada do Wrapper pode ser vista em (CEMIN, 2012).

8 ESTUDOS DE CASO

Este capítulo aborda estudos de caso com a finalidade de validar a arquitetura proposta neste trabalho. Os estudos de caso servem não somente como uma maneira de mostrar o funcionamento da arquitetura, mas também como sendo um artifício para medir-se os custos associados a cada parte do projeto. Como já mencionado no Capítulo 6, este trabalho não se propõe a estudar diferentes implementações em *hardware* ou mesmo em *software* de agentes. Com exemplos simples já é possível verificar o funcionamento da arquitetura e desta forma estudar a viabilidade de se implementar agentes com comportamento mais complexo.

Um simples modelo de agente implementa um comportamento reativo sobre uma determinada ação, respondendo a estímulos que chegam a ele. A partir desta definição, cria-se um agente puramente reativo que reage conforme as informações que ele recebe. Esta primeira abordagem é de uma arquitetura de agentes onde um agente fornece uma certa quantidade de informações para outro agente em uma plataforma remota e este segundo agente reage modificando estas informações e fornecendo uma resposta. Este agente que está na plataforma remota implementa um módulo de operações em ponto flutuante, e é chamado de Agente FPU (*Floating Point Unit*).

Neste estudo será analisado o custo de migração do agente entre as diferentes arquiteturas de *software* e de *hardware*. Serão feitas medidas para analisar o custo de migração dos agentes entre as plataformas de *software* medindo-se o tamanho dos agentes e com isso estimando-se o *overhead* imposto pelo Jade. Após, serão feitas medidas para analisar o custo de migração deste agente para a sua versão em *hardware*. Com estas medidas, se terá o custo de migração do agente na plataforma proposta. Será analisado ainda o custo de uso da plataforma após a migração, ou seja, será medido o custo de execução do agente em *software* e em *hardware* bem como os tempos de cópia de informações entre os diferentes tipos de agente.

Para dar sequência aos experimentos, será descrito um segundo estudo de caso utilizando-se um filtro de Kalman (KALMAN, 1960). A ideia deste estudo é estudar o custo de migração de um agente mais complexo que o agente FPU para que se tenha uma comparação entre os dois tipos de agentes. Este estudo é importante do ponto de vista de estimação de custo de diferentes tipos de agentes para uma posterior classificação dos tipos de agentes que são mais adequados para serem executados seja em *software* seja em *hardware*.

8.1 Agente FPU

Para a implementação deste exemplo é necessária a criação de dois agentes que possam se comunicar entre si, sendo que o primeiro será um agente que requisita alguma informação, e o segundo agente será um que simplesmente responde às requisições do primeiro agente. O primeiro, chamado de *Requester*, fornece dois números e uma operação a um segundo agente, que é chamado de FPU (*Floating Point Unit*), que recebe os dois números e a operação e devolve o resultado. A Figura 31 ilustra esta situação.

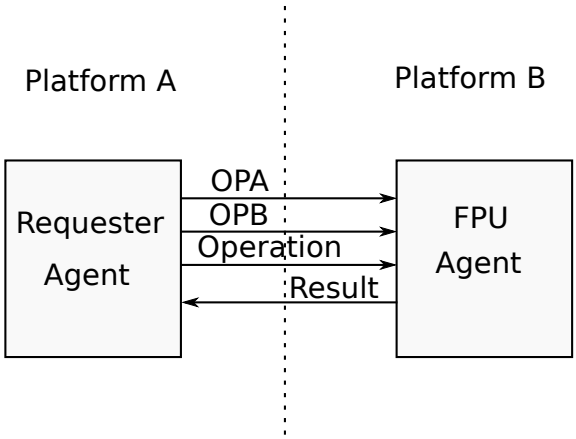

Figura 31: Agente FPU.

Escrever as informações de uma forma serial requer a criação de uma classe com métodos específicos para acessar as informações. Estes métodos são serializados junto com as informações que são trocadas entre os agentes, o que contribui para o aumento do tamanho das mensagens que são trocadas entre os agentes. Além disso, o Jade cria um envelope para a mensagem o que faz com que aumente o *overhead* nas mensagens imposto pelo Jade. A Listagem 8.1 mostra a classe com os dados a serem enviados do *Requester agent* para o *FPU agent*.

```
1public class RequesterDataStruct implements java.io.
    Serializable {
2 double opa;
3 double opb;
4 int operation;
5
6 public RequesterDataStruct() {
7  this.opa = 0;
8  this.opb = 0;
9  this.operation = 0;
10 }
11 public void dataSet(double op1, double op2, int oper) {
12  this.opa = op1;
13  this.opb = op2;
14  this.operation = oper;
15 }
16 public double getOpa() { return this.opa; }
17 public double getOpb() { return this.opb; }
18 public int getOperation() { return this.operation; }
19}
```

Listagem 8.1: Estrutura utilizada na requisição dos dados do agente remoto.

A implementação do agente é relativamente simples, pois a tarefa dele é ler dois números e aplicar uma operação sobre eles e devolver o resultado. O núcleo deste módulo pode ser visto na Listagem 8.2 onde são mostradas somente as operações feitas baseadas nas informações recebidas pelo agente. O código completo deste agente é mostrado na Listagem C.1 que está no Apêndice C.

```
1 void treatOperation()
2 {
3  switch(this.operation) {
4   case def.Agents.FPU_ADD:
5    this.result = this.opa + this.opb;
6    break;
7
8   case def.Agents.FPU_SUB:
9    this.result = this.opa - this.opb;
10   break;
11
12  case def.Agents.FPU_MUL:
13   this.result = this.opa * this.opb;
14   break;
15
16  case def.Agents.FPU_DIV:
17   this.result = this.opa / this.opb;
18   break;
19
20  default:
21   System.out.println("Error receiving command!");
22   this.result = 0;
23   break;
24 }
25}
```

Listagem 8.2: Núcleo do agente FPU implementado em *software*.

Já a versão em *hardware* deste agente é relativamente complexa, pois implementa o padrão de representação de números em ponto flutuante definida pela IEEE-754. O agente espera dois números e uma operação e com isso ele fornece uma resposta, identicamente ao que foi feito pelo mesmo agente implementado em *software*. A Figura 32 ilustra este módulo se comunicando com o mundo externo através de uma camada que é definida por partes do agente implementado em *software*.

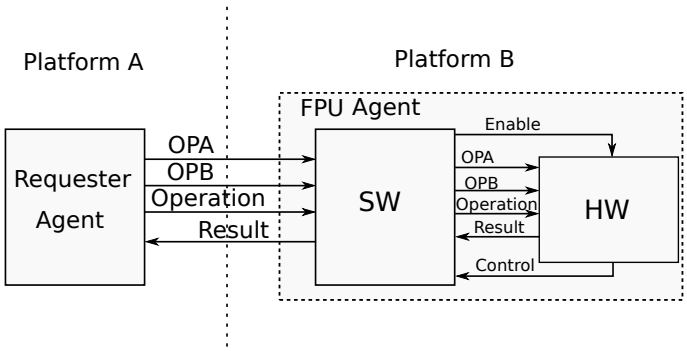

Figura 32: Versão implementada em *hardware* do agente FPU.

Um módulo FPU foi implementado em lógica programável e utilizado neste trabalho. A cópia das informações passadas pelo processador é feita pela instanciação

direta do módulo, como pode ser visto na Listagem 8.3, que mostra um simples mapeamento dos sinais para dentro do bloco *fpu_double*. Maiores detalhes sobre a implementação do módulo FPU podem ser encontrados no Anexo A.

```
1  -- fpu block
2  fpu: entity work.fpu_double
3  port map(
4   clk          =>  clk       ,
5   rst          =>  rst       ,
6   enable       =>  enable    ,
7   rmode        =>  rmode     ,
8   fpu_op       =>  fpu_op    ,
9   opa          =>  opa       ,
10  opb          =>  opb       ,
11  out_fp       =>  out_fp    ,
12  ready        =>  ready     ,
13  underflow    =>  underflow ,
14  overflow     =>  overflow  ,
15  inexact      =>  inexact   ,
16  exception    =>  exception ,
17  invalid      =>  invalid
18 );
```

Listagem 8.3: Mapeamentos das informações para o módulo fpu_double.

A tomada de decisão da migração do agente para o *hardware* foi feita baseada na abordagem onde o agente percebe a existência do FPGA na arquitetura e configura o *hardware* para que seja feita a migração. Esta abordagem foi discutida no Capítulo 6 onde é mostrado o fluxograma proposto de tomada de decisão de migração do agente para a arquitetura de *hardware*. O código que faz a cópia efetiva das informações do agente entre as suas implementações de *software* e *hardware* é mostrado na Listagem C.2, colocada em Anexo.

8.2 Agente Kalman

O agente Kalman é um estudo de caso de uma implementação de um filtro de Kalman de um sistema simples de segunda ordem. O agente migra entre diferentes nós carregando seu código e escopo e medidas são feitas para se analisar os custos associados à migração do agente entre as plataformas de *software* e de *hardware*. Nesta seção, primeiramente é explicado o que é o filtro de Kalman e derivadas as suas equações para um sistema de exemplo. Na seção 8.3 são analisados os resultados da migração do agente bem como são apresentados os gráficos de resultados da simulação do sistema em si.

8.2.1 Filtro de Kalman

O filtro de Kalman foi construído originalmente para uso em navegação aeroespacial, mas seu uso tem sido difundido pois ele tem utilidade em diversas aplicações. O filtro de Kalman é uma ferramenta que consegue estimar as variáveis de um processo e também prever os estados de um sistema. Do ponto de vista de uma modelagem no espaço de estados, o filtro de Kalman é um estimador de estados ideal. Ele tem sido usado em controle de veículos aéreos não tripulados (ZHOU et al., 2010), fusão de sensores e localização (ANJUM et al., 2010) e até mesmo estimação de cargas de baterias (DI DOMENICO; FIENGO; STEFANOPOULOU, 2008). Filtragem de

sinais é algo desejável em diversas aplicações de engenharia e sistemas embarcados, como por exemplo remover ruídos de sinais enquanto ainda mantém-se um nível de sinal aceitável.

Para remover-se o ruído, um sistema precisa ser descrito como um sistema de equações lineares, como pode ser visto na Equação 1, que mostra um sistema genérico a ser modelado. Nesta equação, A, B e C são matrizes que modelam o sistema no espaço de estados. Os valores representados por W_k e Z_k são respectivamente o ruído de processo e de medida.

$$\begin{aligned} x_{k+1} &= AX_k + BU_k + W_k \\ y_{k+1} &= CX_k + Z_k \end{aligned} \tag{1}$$

Em (SIMON, 2001) é modelado um sistema simples de segunda ordem no espaço de estados e aqui será utilizado o mesmo sistema para ter-se uma base confiável de comparação. Este sistema é descrito por um veículo que se move em linha reta com aceleração constante. O vetor de estados é descrito pela posição sua p e velocidade v. As Equações 2 e 3 mostram a descrição da velocidade e posição respectivamente. O vetor de estados é mostrado na Equação 4 e finalmente o sistema é descrito pela Equação 5.

$$v_{k+1} = v_k + Tu_k + v_k^n \tag{2}$$

$$p_{k+1} = p_k + Tv_k + \frac{1}{2}T^2u_k + p_k^n \tag{3}$$

$$x_k = \begin{bmatrix} p_k \\ v_k \end{bmatrix} \tag{4}$$

$$\begin{aligned} x_{k+1} &= \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} T^2/2 \\ T \end{bmatrix} u_k + w_k \\ y_k &= \begin{bmatrix} 1 & 0 \end{bmatrix} x_k + z_k \end{aligned} \tag{5}$$

O filtro de Kalman é um algoritmo iterativo onde em cada laço são calculados a atualização do vetor x estimado, descrita pela Equação 7, o erro estimado de covariância, descrito pela Equação 8 e também o ganho do filtro de Kalman, que é descrito pela Equação 6. De uma maneira geral, o sistema é modelado no espaço de estados e ele está sujeito à perturbações. O filtro vai estimar o estado do sistema, ou seja, sua posição e velocidade, para um tempo qualquer.

$$K_k = AP_kC^T(CP_kC^T + S_z)^{-1} \tag{6}$$

$$\hat{x}_{k+1} = (A\hat{x}_k + Bu_k) + K_k(y_{k+1} - C\hat{x}_k) \tag{7}$$

$$P_{k+1} = AP_kA^T + S_w - AP_kC^TS_z^{-1}CP_kA^T \tag{8}$$

8.3 Análise dos Resultados

Esta seção apresenta os resultados obtidos nos estudos de caso propostos nas seções anteriores. Aqui são feitas as medidas pertinentes para a validação da arquitetura e também estimativas de uso da arquitetura por agentes mais complexos. Inicialmente são mostrados os resultados obtidos no desenvolvimento do agente FPU. Os resultados deste primeiro estudo de caso servem como base para o segundo estudo de caso, onde são mostrados os resultados obtidos com o agente Kalman.

8.3.1 Agente FPU

A análise do agente FPU é dividida em duas partes. A primeira parte faz uma análise da migração deste agente entre diferentes plataformas, tanto em *software* quanto em *hardware*. Nesta etapa são medidos os tempos de migração dos agentes bem como o tamanho que eles ocupam para que a migração seja efetuada. São comparados os resultados dos agentes em *software* e em *hardware* através das medidas efetuadas.

A segunda parte faz uma análise do uso dos agentes, mostrando os resultados obtidos após a migração. Nesta etapa são feitas medidas de tamanho de mensagens que são trocadas entre os diferentes tipos de agentes para analisar-se o *overhead* imposto pelo *framework* do Jade e também para analisar o custo da cópia das informações para os agentes que estão sendo executados em *hardware*. São feitas também medidas de tempo de execução dos agentes em *software* e dos agentes em *hardware*. Nos agentes em *software*, as medidas são feitas diretamente no código, medindo-se o tempo gasto para se executar um determinado trecho de código. Já para as medidas em *hardware*, é utilizada uma ferramenta fornecida pela Xilinx para fazer a análise dos sinais internos ao FPGA.

8.3.1.1 Análise da migração do agente

Inicialmente ambos agentes encontram-se na plataforma A quando é feita a migração do agente FPU para a plataforma B. Neste momento, aproveita-se os métodos fornecidos pelo Jade que possibilitam a implementação de algum comportamento momentos antes da migração e logo após a migração para fazer-se as medidas necessárias para verificar-se o tamanho dos agentes e o tempo gasto na migração destes agentes. A Figura 33 ilustra esta situação.

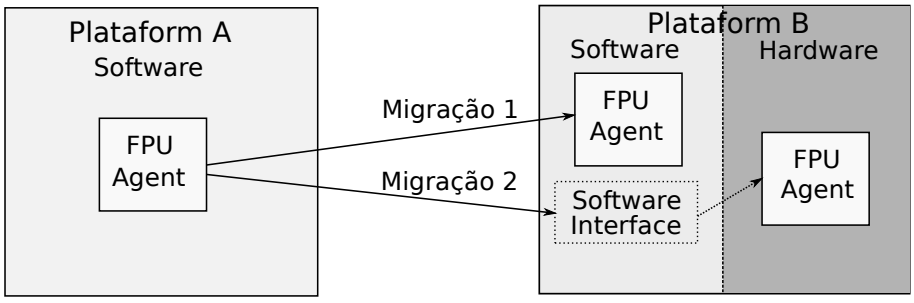

Figura 33: Migração do agente FPU entre diferentes plataformas.

Inicialmente o agente FPU migra da plataforma A, onde é executado puramente em *software* para uma plataforma B, onde inicialmente também é executado puramente em *software*. Na Figura 33 esta etapa é descrita pela *Migração 1*, onde o

agente FPU está migrando entre duas plataformas sendo executado em *software*. Já na etapa referenciada por *Migração 2*, é feita a migração do agente da plataforma de *software* para a plataforma de *hardware*. Durante este processo de migração, são feitas medidas para avaliar-se o tempo de migração em ambos os casos. Na *Migração 1*, os tempos são referenciados à arquitetura do Jade, visto que é uma migração puramente em *software*. Já a *Migração 2* contempla o fluxograma mostrado na Figura 26, que foi mostrado no Capítulo 6. A Tabela 3 mostra os resultados medidos neste procedimento.

Tabela 3: Medida de custo de envio de dados entre agentes em diferentes plataformas.

Medida	**Tempo**
Migração 1	5.3 ms
Migração 2[1]	5.6 ms
Migração 2[2]	20.4 s

As medidas apresentadas mostram os tempos médios observados para a programação do FPGA, leitura e escrita de um byte. A configuração do FPGA apresenta um tempo relativamente alto pois ele é reprogramado por completo. Não está sendo utilizada reprogramação parcial do FPGA neste estudo de caso. Este tempo médio foi observado fazendo-se mais de 100 reprogramações seguidas e medindo-se os tempos de configuração do FPGA. A classificação destes resultados dependem muito da aplicação pois está atrelada aos requisitos de onde o agente se encontra. Os tempos apresentados na Tabela 3 podem ser considerados satisfatórios para algumas aplicações, como por exemplo no uso de VANTs, uma vez que não existe necessariamente a reconfiguração periódica do FPGA, sendo isso feito somente em alguns momentos específicos. Um agente pode migrar entre plataformas de *software* com tempos relativamente baixos, na ordem de milisegundos, o que pode ser considerado satisfatório.

8.3.1.2 Análise do uso do agente

O agente Requester envia 3 informações ao agente FPU, que correspondem aos dois números e a operação que será feita. Cada número é composto de 8 bytes, pois descreve um número em ponto flutuante de precisão dupla, e a operação é composta de apenas 1 byte. Sendo assim, o tamanho mínimo das mensagens enviadas do agente Requester para o agente FPU é de 17 bytes. A resposta é constituída de apenas um número também composto por 8 bytes. A Tabela 4 mostra as medidas feitas na troca de mensagens entre os agentes.

Os resultados mostram uma diferença significativa entre o tamanho mínimo das mensagens e o que foi efetivamente medido. Essa diferença se deve ao fato do encapsulamento e serialização das mensagens feitas pelo Jade para que se cumpra o padrão de comunicação entre os agentes normatizados pela FIPA. Isso se reflete em um *overhead* significativo que é adicionado pela utilização desta plataforma de agentes.

Após o envio do comando, o agente FPU deve processar a informação e devolver o resultado da operação desejada. Desta maneira, foram implementadas as

[1]Medidas feitas com o FPGA já programado

[2]Medidas feitas incluindo-se a programação do FPGA

Tabela 4: Medida de custo de envio de dados entre agentes em diferentes plataformas.

Medida	**Valor**
Dados Saída Requester	560 Bytes
Dados Saída FPU	448 Bytes

versões tanto em *software* quanto em *hardware* deste agente conforme especificado pela IEEE-754 (IEEE, 2012). Na versão em *software* é utilizada a implementação proposta em (SCHOEBERL, 2005), onde é feito manualmente na própria linguagem Java a manipulação dos *bits* para a obtenção do resultado em ponto flutuante sem que se use a unidade em ponto flutuante que está disponível no processador. Foram feitas medidas de tempo sobre este módulo em cada operação e os resultados médios podem ser vistos na Tabela 5.

Tabela 5: Medidas de tempo do agente FPU em *software*.

Operação	**Duração**
Adição	1.3 μs
Subtração	1.4 μs
Multiplicação	50.3 μs
Divisão	340 μs

O tempo total entre o agente Requester enviar uma mensagem e receber a resposta é dependente da latência da rede, que em geral é muito maior do que os tempos apresentados na Tabela 5. Para todas as operações feitas, o tempo total ficou entre 2.5 e 3.2 milissegundos, sendo que independe da operação feita pois existe uma diferença significativa entre as grandezas do tempo para a troca de mensagens e os tempos de execução do agente FPU em *software*.

Na versão em *hardware* deste agente, as informações são passadas para o agente através da camada do Wrapper, que efetua escritas no espaço de memória reservada para o agente. São feitas no total 8 escritas para cada palavra de 64 *bits* e uma escrita para os sinais de *enable* e *operation*, indicando a habilitação do módulo e operação desejada, respectivamente.

A Tabela 6 resume os custos da cópia de informações entre o processador principal e o FPGA. Os tempos são referenciados para leitura e escrita de um único byte. A cópia de informações é feita através de um comando disparado pelo agente em java para o Wrapper, que por sua vez faz a cópia dos dados para a memória que está acessível pelo FPGA. Isto pode ser visto pela Listagem C.2, onde estão detalhadas as funções que acessam diretamente o *hardware* a partir do espaço de usuário.

Tabela 6: Medida de custo de escrita e leitura dos dados no *hardware*.

Medida	**Resultado**
Escrita de um byte	4.4 μ segundos
Leitura de um byte	4.2 μ segundos

As medidas de escrita e leitura de um byte também são médias, pois foram feitas 1000 escritas e 1000 leituras para chegar-se a estes valores. Com estas medidas é possível estimar-se o custo de migração de um agente qualquer para *hardware* baseado

na quantidade de informações que eles precisam trocar para que migração ocorra. Em geral, a migração requer pelo menos duas leituras ao *hardware* para descobrir se o FPGA já está programado, além do tempo necessário para a programação do FPGA. Os dados de tempo de escrita e leitura do FPGA servem também como base para se estimar o tempo de uso de um agente que requer troca de dados constantes entre processador e FPGA.

Uma operação completa do agente FPU em *hardware* necessita de 16 escritas de um byte, mais uma escrita da operação, e o resultado é lido com mais nove leituras de um *byte*, das quais 8 leituras são referentes ao resultado da operação desejada. O tempo total para efetuar qualquer operação do ponto de vista do Wrapper é muito parecido para qualquer operação, visto que os tempos de leitura e escrita são muito maiores que o tempo de execução da operação em *hardware*. Este tempo foi medido em aproximadamente 114 μs, referentes à 17 escritas e 9 leituras.

O tempo total para o envio de uma requisição e a sua resposta na versão em *hardware* deste agente foram medidos e os valores ficaram entre 3 e 3.6 milissegundos, aproximadamente. Novamente, os valores medidos tem em sua grande parte influência da troca de mensagens dependente da rede onde este módulo se encontra. A diferença no uso do agente FPU entre as versões de *software* e de *hardware* não foi muito significativa neste caso pois o tempo de processamento das informações em *hardware* associado à cópia das informações entre o processador e o FPGA foram muito parecidos com o tempo de processamento do mesmo agente na sua versão em *software*.

A Xilinx oferece uma maneira de monitorar sinais que são internos ao FPGA através de uma conexão JTAG. Uma ferramenta chamada *Chipscope* insere um bloco de lógica logo após o processo de síntese que é responsável por monitorar os sinais desejados e mostrá-los em uma interface semelhante ao que se teria em um osciloscópio. As Figuras 34, 35, 36 e 37 mostram as operações de adição, subtração, multiplicação e divisão respectivamente. Foram feitos diversos testes com diferentes números que eram gerados randomicamente pelo agente *Requester* que comprovaram a validade do módulo, além do arquivo de simulação, que foi fornecido junto com o código fonte.

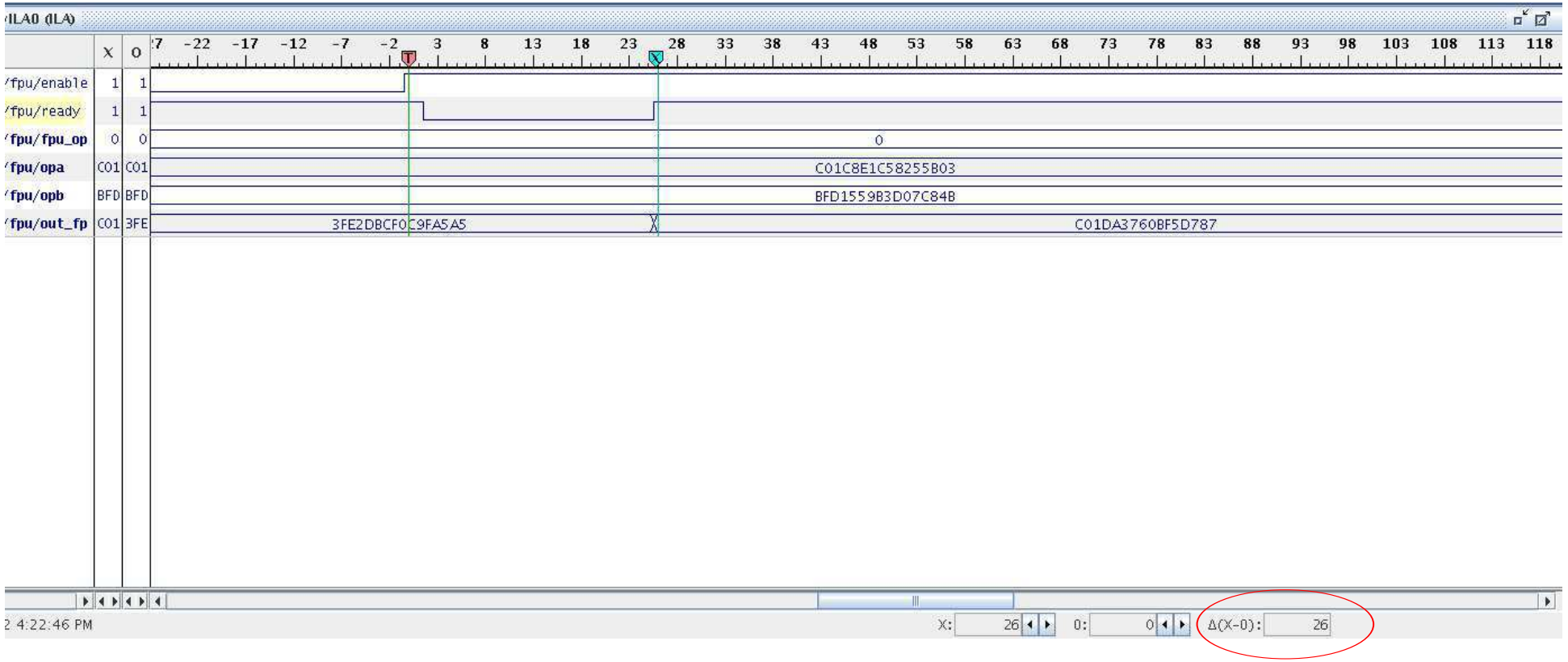

Figura 34: Operação de soma realizada no FPGA.

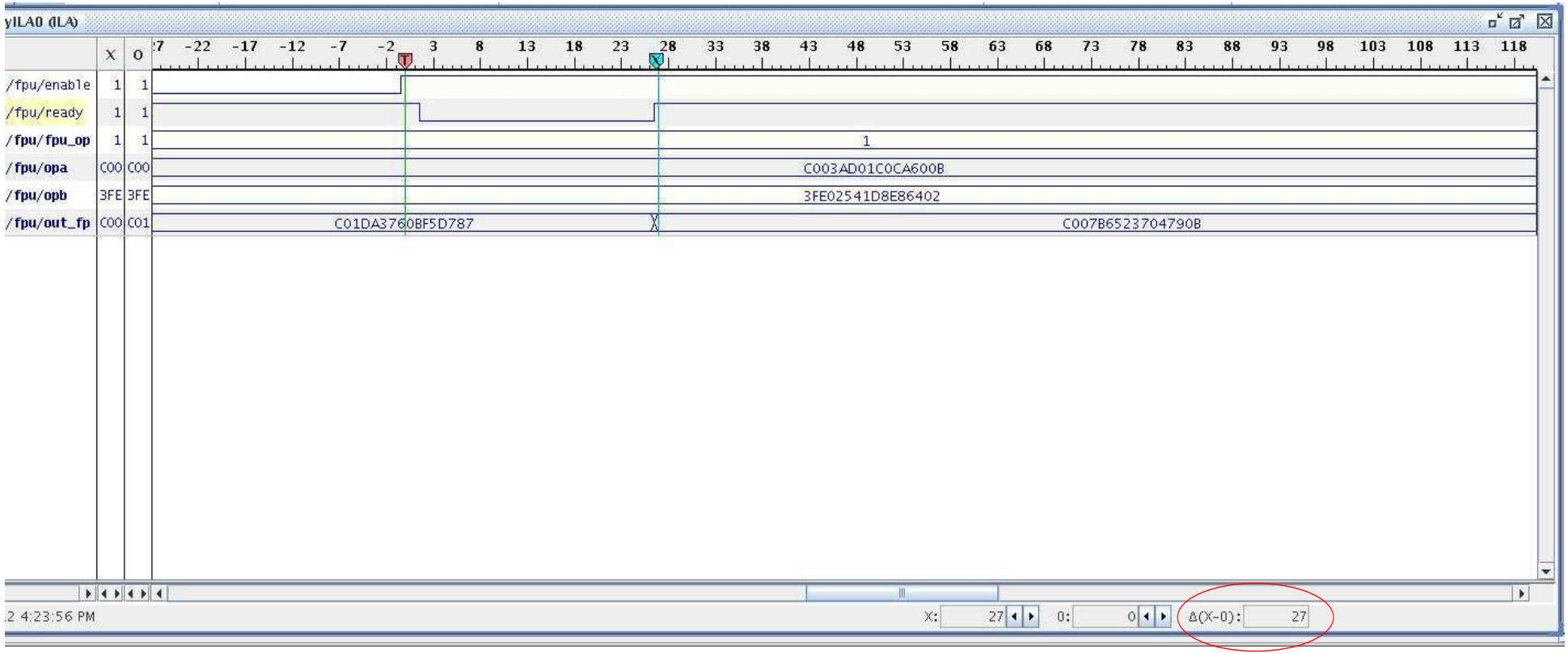

Figura 35: Operação de subtração realizada no FPGA.

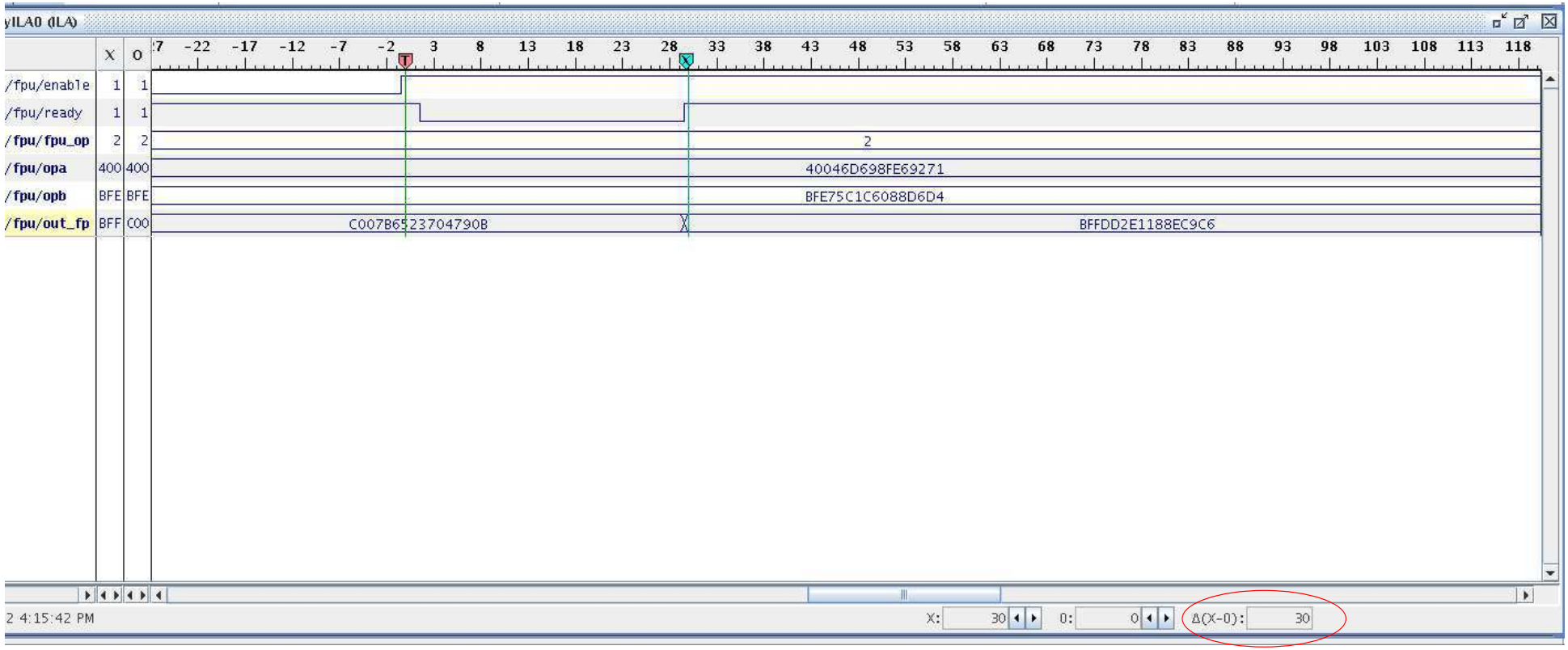

Figura 36: Operação de multiplicação realizada no FPGA.

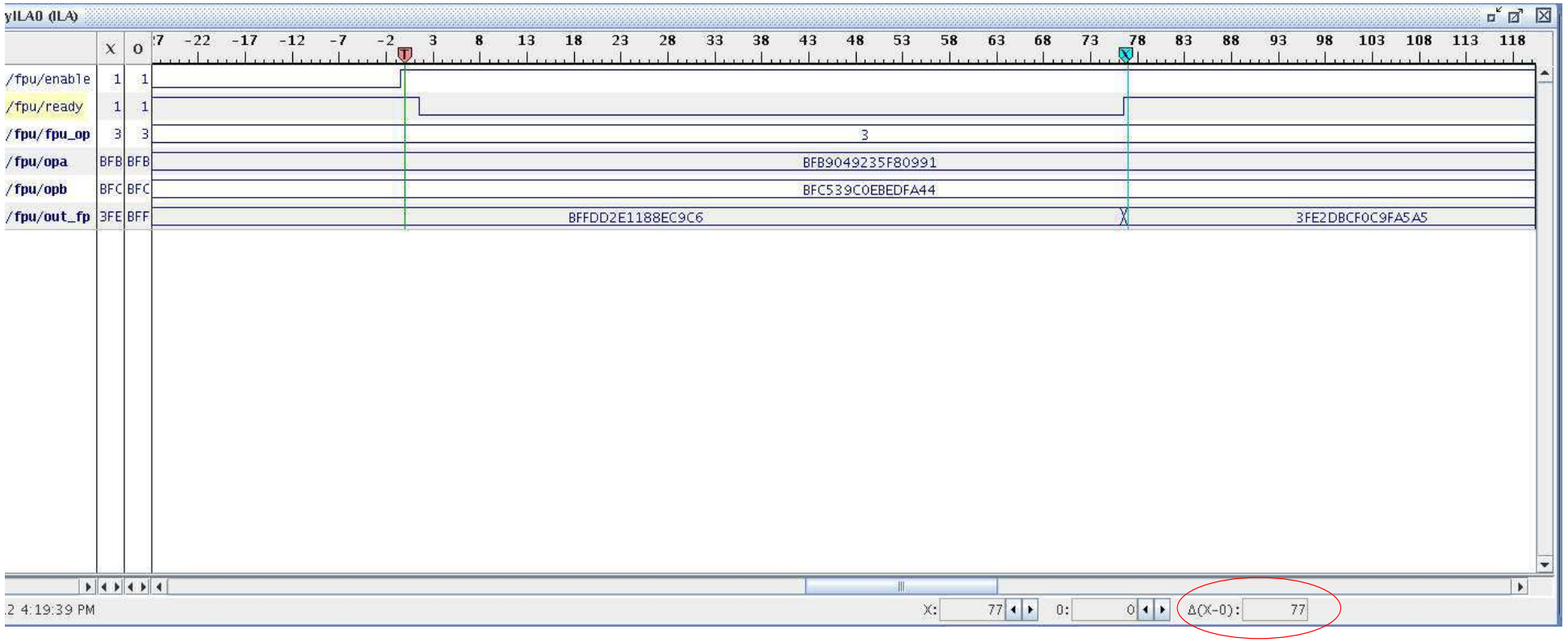

Figura 37: Operação de divisão realizada no FPGA.

A Tabela 7 mostra os valores de tempos dos sinais utilizados nos exemplos bem como a quantidade de ciclos de *clock* necessárias para fazer cada operação. Ainda, é mostrada uma estimativa de tempo para ser feita uma operação baseada no *clock* utilizada no bloco FPU do FPGA. As imagens mostram somente a quantidade de ciclos de*clock* necessárias para se fazer cada operação, sendo que o resultado pode ser visto circulado nas figuras. A placa de testes está utilizando um sinal de *clock* de 100MHz, sendo que ele ainda é expansível. Os resultados mostrados na Tabela 7 são referenciados à frequência de 100MHz.

Tabela 7: Tempos de cada operação para um *clock* de 100 MHz.

Operação	**Ciclos de Clock**	**Tempo**
Soma	26	260 ns
Subtração	27	270 ns
Multiplicação	30	300 ns
Divisão	77	770 ns

8.3.2 Agente Kalman

A análise do agente Kalman é feita em duas partes. A primeira aborda a simulação do filtro pelo agente Kalman, que fará a simulação em diferentes nós, e o objetivo deste primeiro estudo é verificar o custo de migração deste agente entre diferentes plataformas sendo executado em *software*. Após esta análise, é feita uma análise algébrica do filtro, para que se obtenha uma estimativa do custo de migração do agente Kalman para uma plataforma de *hardware*. Esta estimativa é feita baseada nos resultados obtidos pelo agente Kalman, visto que o agente Kalman utiliza um módulo FPU internamente.

8.3.2.1 Análise de migração do agente

Foi implementado um agente móvel com este filtro onde este sistema está sendo simulado. O agente simula este sistema por 100 segundos, em iterações com passo de 0.01 segundos e o veículo tem aceleração constante de $0.1m/s^2$. O código que implementa este agente pode ser visto na Listagem D.1. O agente móvel faz uma simulação completa em um nó e depois migra para outro nó, até que todos os nós da rede executem a simulação do filtro. Os dados mostrados na Tabela 8 mostram algumas medidas feitas neste agente durante a sua execução e migração.

Como resultado desta simulação, foram geradas as figuras de posição, mostrada na Figura 38, e velocidade, mostrada na Figura 39. Pode ser percebido em ambas as figuras um ruído em torno da linha contínua. Este ruído é o que foi inserido no processo e a linha contínua é a estimativa da variável mensurada.

Tabela 8: Resultados medidos do agente móvel Kalman.

Medida	**Resultado**
Tamanho do agente móvel	3634 Bytes
Tampo médio migração	8.35 ms
Tempo médio de execução	12729 ms
Utilização do processador	100%

O agente Kalman implementado em *software* resultou em 100% de utilização do processador durante a sua execução no sistema embarcado de testes utilizado sendo que o tempo médio de execução desta simulação ficou próxima dos 13 segundos. O custo de migração do agente é de 3634 *bytes*, medido-se o tráfego de dados no momento de migração do agente. Este custo de migração contém as informações trocadas pelo protocolo ACL especificado pela FIPA, ou seja, nem todas estas informações são necessárias na migração do agente para o *hardware*.

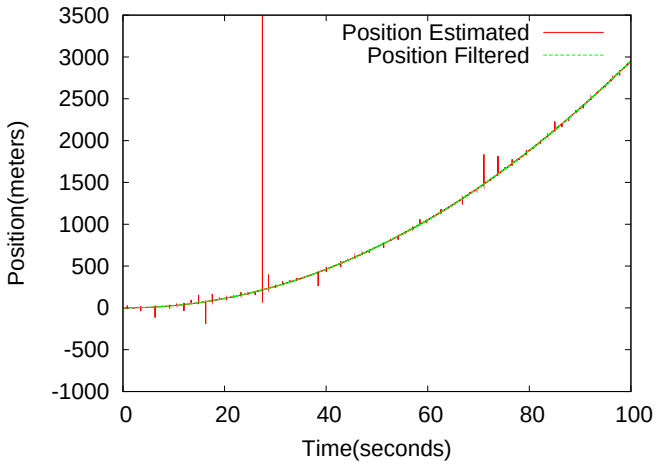

Figura 38: Posição estimada do veículo.

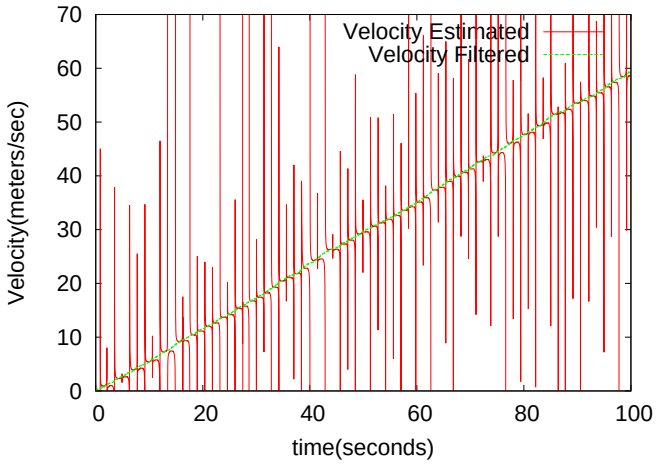

Figura 39: Velocidade estimada do veículo.

Estes resultados mostram que o agente Kalman utiliza todos os recursos do processador durante a sua execução. Isso afeta o desempenho do sistema, e outros agentes que estão sendo executados no processador durante a execução do agente Kalman, terão seu desempenho prejudicado. Por exemplo, caso existisse algum agente fazendo o controle de algum processo com algum tipo de requisito temporal, este controle seria naturalmente prejudicado. Como sugestão, outros agentes podem usar esta medida de uso do processador como base para motivar a migração.

8.3.2.2 Análise algébrica do filtro

A ideia de se fazer uma análise algébrica do filtro de Kalman serve para se analisar quantas operações elementares são necessárias para a implementação do mesmo. Estes resultados servem como base para um estudo de validade de migração do agente para *hardware*. Com base nestes dados e nos dados do agente FPU, é possível se estimar o custo computacional para a modelagem em lógica programável do agente Kalman.

$$\begin{aligned}
PC^T{}_{11} &= p_{11}c_{11} + p_{12}c_{12} \\
PC^T{}_{21} &= p_{21}c_{11} + p_{22}c_{12} \\
CPC^T{}_{11} &= c_{11}PC^T{}_{11} + c_{12}PC^T{}_{21} \\
CPC^TSz_{11} &= CPC^T{}_{11} + sz_{11} \\
CPC^TSzinv_{11} &= \frac{1}{CPC^TSz11} \\
APC^T{}_{11} &= a_{11}PC^T{}_{11} + a_{12}PC^T{}_{21} \\
APC^T{}_{21} &= a_{21}PC^T{}_{11} + a_{22}PC^T{}_{21} \\
k11 &= APC^T{}_{11}CPC^TSzinv11 \\
k21 &= APC^T{}_{21}CPC^TSzinv11
\end{aligned} \tag{9}$$

A partir da Equação 6 que define o ganho do filtro de Kalman, deriva-se a Equação 9, que mostra os passos feitos durante a implementação desta parte do filtro. Nesta equação são utilizadas 5 operações de soma, 12 multiplicações e uma divisão. Fazendo-se a mesma análise para as Equações 7 e 8, pode-se obter o total de operações necessárias para a implementação do filtro. A Tabela 9 resume estes valores.

Tabela 9: Operações básicas necessárias nas equações do filtro de Kalman.

Equação	**Adição**	**Subtração**	**Multiplicação**	**Divisão**
6	5	0	12	1
7	7	1	10	0
8	18	0	18	1
Total	30	1	40	2

A partir dos resultados mostrados na Tabela 7 e dos resultados mostrados na Tabela 9, estima-se o tempo para a execução das operações do filtro de Kalman simplesmente somando-se a quantidade de operações e multiplicando-se pelo tempo estimado. Assim teremos uma estimativa do pior caso implementado em lógica programável, pois não está se considerando o uso do paralelismo de operações. O tempo estimado é dado por $30 \times 260ns + 1 \times 270ns + 40 \times 300ns + 2 \times 770ns = 21510ns$, ou seja, para cada iteração do filtro, serão gastos $21510ns$. Para fins de comparação, na versão em *software* deste filtro são feitas 10000 interações, então, neste caso, este valor passa a ser de aproximadamente 200 milissegundos. Este valor não está considerando o *overhead* de leitura dos valores a cada interação, para isso deve-se adicionar mais 2 leituras de registradores de 64 *bits* a cada iteração, ou seja, são adicionados mais $10000 \times 16 \times 4.2\mu s = 672ms$, sendo que o tempo estimado final fica em aproximadamente 872 milissegundos. É estimado então que a versão

de *hardware* deste agente será mais de 15 vezes mais rápida do que o análogo em *software* considerando-se todo o período de simulação bem como o *overhead* de escrita e leitura.

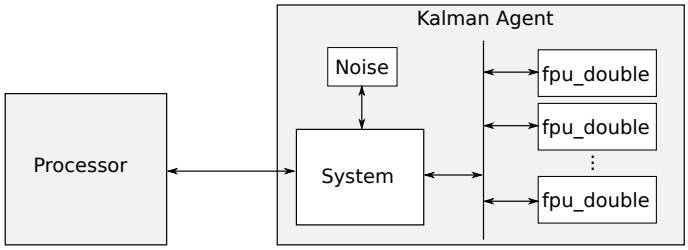

Figura 40: Implementação em hardware do agente Kalman.

A Figura 40 mostra uma possível estrutura para o agente Kalman, utilizando diversos blocos externos do módulo FPU, paralelizando-se assim as operações internas ao sistema utilizado. A análise feita neste capítulo considera o uso de somente uma instância do módulo FPU, e portanto estes resultados ainda podem ser melhorados. A análise da implementação de módulos mais complexos é motivação para outro tipo de estudo e ficaa como proposta para trabalhos futuros.

A Tabela 10 mostra um resumo de diversas operações medidas e estimadas para o agente Kalman. O Caso A é o resultado de uma migração entre dois agentes em *software* em diferentes nós enquanto que o Caso B é uma migração entre um agente em *software* e um agente em *hardware*. Vale a pena observar que o tamanho do agente é o mesmo, visto que o agente está migrando sem carregar junto o *bitstream* para programar o FPGA que é indicada pelo Caso C. Caso o agente não tivesse carregado junto o *bitstream*, ao chegar no nó destino, ele identificaria a falta do agente em *hardware* e executaria o algoritmo relatado no Capítulo 6, onde é mostrado um fluxograma para programação do agente em *hardware*. Os Casos D e E mostram os custos relacionados ao uso do agente em *software* e em *hardware*, respectivamente. Um agente Requester solicita as informações de simulação a um agente remoto que devolve os resultados sobre a forma de uma mensagem ACL.

Tabela 10: Resumo das medidas feitas com o agente Kalman.

Medida	**Tamanho**	**Tempo**
Caso A	3634 Bytes	8.35 ms
Caso B	3634 Bytes	9.18 ms
Caso C	340 KB	21.2 s
Caso D	5452 Bytes	12.74 s
Caso E [3]	5452 Bytes	5 ms

A implementação do agente Kalman em *hardware* se mostra viável quando o custo é o tempo de execução deste agente nas suas duas versões. O *overhead* de escrita e leitura de registradores do FPGA não inviabiliza a implementação deste agente, uma vez que estima-se que o tempo de execução será mais de 15 vezes menor do que na sua versão em *software*. Esta diferença pode ser ainda maior dependendo da maneira como se implementa o agente em *hardware*, usando-se por exemplo diversos *pipelines* durante as operações nas matrizes do agente Kalman.

[3]Este caso não foi implementado, logo o tempo é estimado.

8.4 Conclusão

Este capítulo descreveu estudos de caso que serviram para validar a arquitetura proposta neste trabalho. Inicialmente, foi analisado a comunicação entre agentes de *software* e *hardware* mostrando assim a transparência da comunicação entre estes dois tipos de agentes que estavam situados em diferentes plataformas físicas. Este estudo também avaliou os custos da migração do agente de *software* para *hardware* utilizando-se uma das abordagens discutidas na proposta deste trabalho.

Posteriormente foi analisado um agente um pouco mais complexo, pois utiliza em seu núcleo partes do estudo anterior. Este agente foi chamado de agente Kalman, pois ele implementa a simulação de um sistema de segunda ordem utilizando-se do filtro de Kalman para remover ruídos randômicos que são inseridos na simulação. Com este estudo foi possível analisar o custo de migração deste agente entre diferentes nós e também analisar a viabilidade de implementação de um agente deste tipo em uma plataforma de *hardware*.

Os resultados deste capítulo mostraram a viabilidade da implementação de agentes em *hardware* e também uma maneira de fazer eles se comunicarem com agentes externos utilizando-se a linguagem ACL fornecida pelo Jade. Estes resultados também mostraram que a arquitetura proposta permite migração de agentes junto com a reconfiguração de *hardware*. Foi mostrado uma medida de custo de troca de mensagens entre agentes e também uma medida do custo de migração do agente entre dois nós diferentes. Foi utilizado um módulo de operações em ponto flutuante para se construir um comportamento de um agente em *hardware* que se comunicasse com agentes em *software* de forma transparente. Os resultados deste estudo foram utilizados para se estimar o custo computacional e viabilidade de implementação de um agente que implementasse um filtro de Kalman. O agente Kalman se mostrou viável de ser implementado quando comparado à sua versão em *software* pois foi estimado que o custo de execução dele em *hardware* seria aproximadamente 15 vezes menor do que o custo de execução do agente em *software*.

O *overhead* imposto pelo *Wrapper* deve ser levado sempre em consideração na implementação dos agentes em *hardware*. Em um caso genérico, o *overhead* consiste nos tempos necessários para leitura e escrita de registradores do FPGA. Uma vez que a cópia das informações entre o processador passa pelo Wrapper, agentes que necessitam de muita troca de informações podem ser prejudicados pelos tempos mostrados na Tabela 6. Aliado a este fato, existe ainda a latência imposta pela rede, que em alguns casos pode ter valores elevados ao ponto de mascarar o acréscimo de tempo referente à cópia de dados.

Agentes que não utilizam muito o poder computacional de paralelismo oferecido pelo FPGA podem não ser viáveis pois o custo para se mapear os dados entre os agentes é alto. No exemplo do agente Kalman, o *overhead* da cópia dos dados foi cerca de 3 vezes maior do que o custo da execução do agente em si. A análise de viabilidade deve ser estudada caso a caso, pois o custo é dependente da aplicação e dos requisitos da mesma. A análise feita no agente Kalman não considera o paralelismo entre as operações e estas estimativas podem ser melhoradas utilizando-se *pipelines* nas operações que descrevem o filtro.

A proposta implementada mostrou-se viável pelos resultados apresentados nos estudos de caso deste capítulo. A migração de agentes entre *software* e *hardware* foi demonstrada utilizando-se o agente FPU onde resultados de migração e uso do agente foram aplicados. O agente Kalman foi utilizado para demonstrar a flexibilidade da

arquitetura com outros tipos de agentes que utilizam o recurso do paralelismo o FPGA. Os resultados também mostraram a transparência da comunicação entre os diferentes tipos de agentes, possibilitando que diversos agentes sejam criados e mantidos independente da sua natureza.

A migração de agentes para *hardware* é viável em praticamente qualquer situação que se tenha a implementação do agente em lógica programável. Entretanto não é todo o tipo de agente que justifica o esforço da sua implementação em *hardware*. Os exemplos mostrados neste capítulo mostraram que os custos são menores para agentes que utilizam mais os recursos do FPGA. Para agentes que fazem tarefas simples, como por exemplo o agente FPU, os custos de se executar em *software* ou em *hardware* são muito parecidos. Existem outros tipos de custos que poderiam ser analisados, como por exemplo o consumo de energia do dispositivo, mas este tipo de análise está vinculada ao tipo de requisito de cada proposta que utiliza esta arquitetura como base.

Um dos pontos fortes da arquitetura proposta neste trabalho é a possibilidade de migração de agentes entre *software* e *hardware* e também a comunicação transparente entre eles. Este tipo de abordagem aumenta a flexibilidade das RSSFs que estiverem utilizando uma arquitetura baseada nesta proposta. Uma das melhorias que já é possível visualizar é a diminuição dos tempos de escrita e leitura do FPGA pelo processador principal. Entretanto, isso é dependente da arquitetura do processador principal e pode variar para as diferentes implementações. Outro ponto a melhorar é o uso de reconfiguração parcial do FPGA, o que possibilita tempos reduzidos de configuração dos agentes em *hardware* bem como flexibiliza a configuração dos diferentes agentes de forma independente, isto é, permite-se configurar um agente interromper a execução de todos os outros agentes que estão implementados no único *bitstream* que programa o FPGA.

9 CONCLUSÕES

O uso de agentes móveis em redes de sensores sem fio traz flexibilidade à rede, uma vez que os agentes aumentam o dinamismo da rede permitindo a sua rápida reconfiguração. A reconfiguração dos dispositivos é necessária principalmente em ambientes onde os requisitos mudam ao longo do tempo, e então reconfigurar a rede dinamicamente permite que as tarefas que estão sendo executadas em cada nó acompanhem as necessidades impostas pelo meio ou pela aplicação.

Neste trabalho foi proposta uma arquitetura reconfigurável com agentes móveis sendo executados tanto em *softwre* ou em *hardware*. Esta arquitetura suporta a comunicação de agentes implementados em *software* com agentes implementados em *hardware* de forma transparente. A arquitetura suporta ainda a migração destes agentes de forma que um agente pode estar sendo executado em *software* em uma plataforma e passar a ser executado em *hardware* em outra plataforma após a sua migração.

A utilização do *framework* Jade permite a comunicação dos agentes implementados neste trabalho com qualquer outro agente que siga o padrão normatizado pela FIPA. Com isso, a arquitetura proposta dispõe de agentes que são capazes de serem inseridos em outros sistemas multi-agente o que torna o sistema mais flexível uma vez que ele pode ser facilmente acoplado a outras plataformas. O Jade ainda fornece um ambiente de fácil implementação de agentes, abstraindo os conceitos dos agentes às classes Java de seu *framework*.

Os agentes implementados utilizando o Jade comunicam-se com os agentes em *hardware* através de uma camada de abstração criada neste trabalho chamada de Wrapper de agentes. Este Wrapper é responsável por criar a interface necessária para que os agentes consigam migrar entre *software* e *hardware* e ainda permitir a comunicação entre eles. O Wrapper também é responsável por manter o sincronismo em baixo nível, garantindo a exclusão mútua de diversos agentes tentando acessar um mesmo dispositivo em *hardware*.

A proposta atual contempla o uso do Wrapper em *software*, entretanto seus conceitos são extensíveis ao *hardware*. A implementação do Wrapper em *hardware* demanda a implementação da camada de comunicação fornecida pelo Jade bem como todo o protocolo ACL dentro do FPGA. Este trabalho está propondo o uso do Wrapper em *software* somente e a sua implementação em *hardware* é uma proposta para trabalhos futuros.

Os agentes em *hardware* são implementados manualmente e não existe nenhum mecanismo automático para a geração das duas versões de agentes simultaneamente. Este trabalho não tem o foco em métodos para a criação dos agentes mas sim na criação do ambiente que propicia o uso de agentes tanto em *software* quanto em

hardware de forma transparente.

Os resultados dos casos de uso que foram implementados demonstraram a viabilidade da criação de agentes tanto em *software* quanto em *hardware*. Estes agentes são capazes de se comunicar uns com os outros independente do seu tipo de implementação. A arquitetura ainda suporta a migração destes agentes de forma que os agentes são capazes de decidir internamente sobre onde eles vão estar sendo executados, se em *software* ou se em *hardware*.

Este trabalho traz como principal contribuição o estudo da viabilidade da migração de agentes entre diferentes plataformas de *software* e de *hardware*, e também a possibilidade de comunicação entre estes agentes de forma transparente. Este estudo serve como base para desenvolvimento de arquiteturas de redes de sensores sem fio e de sistemas multi-agente que queiram utilizar recursos de *software* e de *hardware* concorrentemente. A arquitetura proposta também é um resultado importante, uma vez que ela contribui para o estado da arte no desenvolvimento de arquiteturas de redes de sensores sem fio e de sistemas multi-agente. A utilização da arquitetura proposta neste trabalho por outros sistemas propiciará o uso dos recursos do FPGA para executar os agentes em *hardware* mantendo a transparência necessária aos agentes independentemente da sua natureza.

Existe ainda a possibilidade de expansão deste trabalho e algumas ideias já são vislumbradas para o curto e médio prazo. A primeira modificação possível é a utilização da programação parcial do FPGA, que já foi discutida extensivamente ao longo deste trabalho. Isso não foi implementado devido aos prazos deste trabalho quando então optou-se por em um primeiro momento criar uma base teórica e prática da arquitetura para depois possibilitar a sua expansão. A programação parcial trará consigo uma melhoria nos resultados apresentados visto que o tempo de reconfiguração dos agentes em *hardware* deverá diminuir consideravelmente, ficando diretamente proporcional à complexidade do agente implementado em *hardware*, visto que isso se refletirá no tamanho do seu *bitstream*. Esta abordagem ainda trará a possibilidade de reconfiguração de agentes de forma independente, ou seja, será possível reconfigurar diferentes agentes em *hardware* sem que isso afete a execução de outros agentes que estão no FPGA, o que já é feito por alguns trabalhos mostrados no Capítulo 5 deste trabalho.

Outra possível melhoria é a eliminação do uso de um processador externo ao FPGA. É possível utilizar-se tanto soluções com processadores implementados em lógica programável, conhecidos como *soft-cores*, ou ainda aproveitar núcleos de processadores que já vem embarcados no FPGA, que é o caso do Virtex4 da Xilinx que contém um núcleo de um PowerPC. Este tipo de abordagem traz novas possibilidades de implementações. Uma primeira ideia seria fazer o suporte da máquina virtual Java para estes processadores e continuar utilizando-se da arquitetura do Jade. Caso isso não seja possível, será necessário um novo estudo de implementações de mais baixo nível do protocolo da FIPA, o que é possível de ser feito utilizando-se um *framework* chamado Mobile-C (MOBILEC, 2012). Este *framework* implementa o protocolo FIPA em uma linguagem de mais baixo nível eliminando-se a necessidade de utilização de máquinas virtuais nos processadores embarcados.

A arquitetura poderia ser ainda somente em *hardware*, o que demandaria a implementação do protocolo FIPA internamente ao FPGA. Isso demandaria o uso de FPGAs com mais recursos pois isso ocuparia mais espaço na lógica, o que poderia diminuir o espaço disponível para os agentes e então não seria viável a sus im-

plementação. Este tipo de abordagem demandaria um tempo maior de estudo de implementação também, o que pode ser fruto de outros estudos que fogem do escopo deste trabalho.

REFERÊNCIAS

AHMED, S.; RAJA, M. Y. A. Telemedic sensor networks and informatics for healthcare services. **International Symposium on High-Capacity Optical Networks and Enabling Technologies (HONET)**, Riyadh, p.67–73, Dec. 2009.

AKYILDIZ, I. et al. A survey on sensor networks. **Communications Magazine, IEEE**, Atlanta, v.40, n.8, p.102 – 114, Aug. 2002.

ALLGAYER, R. S. **Femtonode**: arquitetura de nó-sensor reconfigurável e customizável para rede de sensores sem fio. 2009. Mestrado em Controle e Automação — Universidade Federal do Rio Grande do Sul, Escola de Engenharia, Programa de Pós-Graduação em Engenharia Elétrica, Porto Alegre, 2009.

ANDRE, N. et al. Miniaturized wireless sensing system for real-time breath activity recording. **Sensors Journal, IEEE**, Louvain, v.10, n.1, p.178 –184, Jan. 2010.

ANJUM, M. et al. Sensor data fusion using unscented kalman filter for accurate localization of mobile robots. In: INTERNATIONAL CONFERENCE ON CONTROL AUTOMATION AND SYSTEMS (ICCAS), 2010., Jeju. **Proceedings...** Jeju:IEEE Conference Publications, 2010. v.1, n.2, p.947 –952.

BENSO, A. et al. Reconfigurable systems self-healing using mobile hardware agents. In: TEST CONFERENCE, 2005. ITC 2005. IEEE INTERNATIONAL, Austin. **Proceedings...** Austin:IEEE Conference Publications, 2005. v.1, n.2, p.9 – 476.

BRATMAN, M. E. **Intention, Plans, and Practical Reason**. Cambridge: Harvard University Press, 1987.

BUSE, D.; FENG, J.; WU, Q. Mobile agents for data analysis in industrial automation systems. In: IIEEE/WIC INTERNATIONAL CONFERENCE ON NTELLIGENT AGENT TECHNOLOGY, 2003., Macau. **Proceedings...** Macau:IEEE Conference Publications, 2003. v.2, n.1, p.60 – 66.

BUSE, D.; WU, Q. Mobile agents for remote control of distributed systems. **IEEE Transactions on Industrial Electronics**, Vaxjo, v.51, n.6, p.1142 – 1149, dec. 2004.

CEMIN, D. **Implementação do Wrapper de Agentes**. Disponível em: <https://github.com/davidcemin/wrapper>. Acesso em: 12 Mar. 2012.

CHEN, M. et al. Mobile agent based wireless sensor networks. **Journal of Computing**, Vancouver, v.1, n.1, Apr. 2006.

CHESS, D. et al. Itinerant agents for mobile computing. **Communications Surveys Tutorials, IEEE**, New York, v.3, n.3, p.34 –49, Apr. 2000.

CHINRUNGRUENG, J.; KAEWKAMNERD, S. Wireless magnetic sensor network for collecting vehicle data. In: SENSORS, 2009 IEEE, Pathumthani. **Proceedings...** Pathumthani:IEEE Conference Publications, 2009. p.1792 –1795.

CHIPCON. **Página do Datasheet do Módulo**. Disponível em: <http://www.cse.ohio-state.edu/siefast/nest/nest_webpage/datasheet/Chipcon20CC1000.pdf>. Acesso em: 08 Nov. 2011.

CORKILL., D. D. Blackboard Systems. **AI Expert**, San Antonio, p.40–47, Sep. 1991.

DI DOMENICO, D.; FIENGO, G.; STEFANOPOULOU, A. Lithium-ion battery state of charge estimation with a kalman filter based on a electrochemical model. In: IEEE INTERNATIONAL CONFERENCE ON CONTROL APPLICATIONS, 2008., San Antonio. **Proceedings...** San Antonio:IEEE Conference Publications, 2008. v.1, n.1, p.702 –707.

ENZ, C.; SCOLARI, N.; YODPRASIT, U. Ultra low-power radio design for wireless sensor networks. In: IEEE INTERNATIONAL WORKSHOP ON INTEGRATED CIRCUITS FOR WIDEBAND COMMUNICATION AND WIRELESS SENSOR NETWORKS, 2005., Singapore. **Proceedings...** Singapore:IEEE Conference Publications, 2005. v.2, n.1, p.1 – 17.

FILGUEIRAS, T.; LUNG, L. C.; OLIVEIRA RECH, L. de. Providing real-time scheduling for mobile agents in the JADE platform. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT/COMPONENT/SERVICE-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 2012, Beijing. **Proceedings...** Beijing:IEEE Conference Publications, 2012. p.8 –15.

FIPA. **FIPA ACL Message Structure Specification**. Disponível em: <http://www.fipa.org/specs/fipa00061/>. Acesso em: 21 Mar. 2012.

FOK, C.-L.; ROMAN, G.-C.; LU, C. Rapid development and flexible deployment of adaptive wireless sensor network applications. In: IEEE INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 2005., 25., Washington. **Proceedings...** Washington:IEEE Conference Publications, 2005. n.3, p.653 –662.

FOK, C.-L.; ROMAN, G.-C.; LU, C. Mobile agent middleware for sensor networks: an application case study. In: INTERNATIONAL SYMPOSIUM ON INFORMATION PROCESSING IN SENSOR NETWORKS, 2005., Los Angeles. **Proceedings...** Los Angeles:IEEE Conference Publications, 2005. n.2, p.382 – 387.

FREESCALE. **Página do Fabricante**. Disponível em: <http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC8314E>. Acesso em: 16 Jun. 2012.

FREITAS, E. P. de. **Cooperative context-aware setup and performance surveillance missions using static and mobile wireless sensor networks**. 2012. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2012.

GELERNTER, D. Generative communication in Linda. **ACM Trans. Program. Lang. Syst.**, New York, v.7, n.1, p.80–112, Jan. 1985.

GOMEZ-GUALDRON, J.; VELEZ-REYES, M.; COLLAZO, L. Self-reconfigurable electric power distribution system using multi-agent systems. In: IEEE ELECTRIC SHIP TECHNOLOGIES SYMPOSIUM, 2007, Arlington. **Proceedings...** Arlington:IEEE Conference Publications, 2007. v.2, n.2, p.180 –187.

HAMA, M. T. **Uma plataforma orientada a agentes para o desenvolvimento de software em veículos aéreos não tripulados**. 2012. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2012.

HAN, F. et al. Design of coal mine wireless sensor networks based on piezoelectric sensors for gas monitoring. In: INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, MANAGEMENT SCIENCE AND ELECTRONIC COMMERCE (AIMSEC), 2011, 2., Zhengzhou. **Proceedings...** Zhengzhou:IEEE Conference Publications, 2011. n.3, p.3879 –3882.

HIGHTOWER, J.; BORRIELLO, G. A survey and taxonomy of location systems for ubiquitous computing. **Technical Report UW-CSE**, Seattle, Aug. 2003.

HILL, J. L.; CULLER, D. E. Mica: a wireless platform for deeply embedded networks. **IEEE Micro**, Los Alamitos, v.22, p.12–24, Nov. 2002.

IEEE. **IEEE 754 Standard**. Dísponível em <http://grouper.ieee.org/groups/754/>Acesso em: 12 Mar. 2012.

JIN, J. she et al. Development of remote-controlled home automation system with wireless sensor network. **5th IEEE International Symposium on Embedded Computing**, Dalian, p.169–173, Oct. 2008.

KALMAN, R. E. A new approach to linear filtering and prediction problems. **Journal Of Basic Engineering**, Baltimore, v.82, n.Series D, p.35–45, 1960.

KANSAL, A. et al. Novel adaptive FPGA-based self-calibration and self-testing scheme with PN sequences for MEMS-based inertial sensors. In: IEEE INTERNATIONAL MIXED-SIGNALS, SENSORS AND SYSTEMS TEST WORKSHOP (IMS3TW), 2011, Santa Barbara. **Proceedings...** Santa Barbara:IEEE Conference Publications, 2011. n.4, p.120 –126.

KUORILEHTO, M.; HANNIKAINEN, M.; HAMALAINEN, T. A survey of application distribution in wireless sensor networks. **EURASIP Journal on Wireless Communications and Networking**, Tampere, p.774–788, 2005.

LANGE, D. B.; OSHIMA, M. Seven good reasons for mobile agents. **Commun. ACM**, New York, v.42, n.3, p.88–89, Mar. 1999.

LAUKKANEN, M.; HELIN, H.; LAAMANEN, H. Supporting nomadic agent-based applications in the FIPA agent architecture. In: AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, New York. **Proceedings...** New York:ACM Conference Publications, 2002. p.1348–1355. (AAMAS '02).

LEE, J. yun et al. The energy conversion system with piezoelectric effect for wireless sensor network. In: IEEE INTERNATIONAL CONFERENCE ON SUSTAINABLE ENERGY TECHNOLOGIES, 2008., Singapore. **Proceedings...** Singapore:IEEE Conference Publications, 2008. v.1, n.3, p.820 –824.

LEE, S. H. et al. Wireless sensor network design for tactical military applications : remote large-scale environments. **IEEE Military Communications Conference**, New Jersey, p.1–7, Oct. 2009.

LEVIS, P. et al. TinyOS: an operating system for sensor networks ambient intelligence. In: WEBER, W.; RABAEY, J. M.; AARTS, E. (Ed.). **Ambient Intelligence**. Berlin/Heidelberg: Springer Berlin Heidelberg, 2005. p.115–148.

LI, Z. et al. An urban traffic control system based on mobile multi-agents. In: IEEE INTERNATIONAL CONFERENCE ON VEHICULAR ELECTRONICS AND SAFETY, 2006., Beijing. **Proceedings...** Beijing:IEEE Conference Publications, 2006. n.6, p.103 –108.

LIU, A. et al. An energy-efficient MAC protocol based on routing information for wireless sensor networks. **IEEE Wireless Communications and Networking Conference**, Zhengzhou, p.458–462, March 2007.

LUNDGREN, D. **Opencores Project Web Site**. Disponível em: <http://opencores.org/project,fpu_double>. Acesso em: 12 Mar. 2012.

MAXFIELD, C. **The Design Warrior's Guide to FPGAs**. Orlando: Academic Press, Inc., 2004.

MENG, Y. An agent-based mobile robot system using configurable SOC technique. In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 2006., Orlando. **Proceedings...** Orlando:IEEE Conference Publications, 2006. v.1, n.1, p.3368 –3373.

MICRON. **Página do Fabricante**. Disponível em: <http://www.micron.com/products/dram/rldram.html>. Acesso em: 05 Nov. 2011.

MILLER, K.; SURESH, S. Monitoring patient health using policy based agents in wireless body sensor mesh networks. In: WORLD CONGRESS ON NATURE BIOLOGICALLY INSPIRED COMPUTING, 2009., Coimbatore. **Proceedings...** Coimbatore:IEEE Conference Publications, 2009. v.1, n.1, p.503 –508.

MOBILEC. **Página do projeto Mobile C**. Disponível em: <http://www.mobilec.org/>. Acesso em: 16 Ago. 2012.

MüLLER, I. et al. Namimote:a low-cost sensor node for wireless sensor networks. In: ANDREEV, S.; BALANDIN, S.; KOUCHERYAVY, Y. (Ed.). **Internet of Things, Smart Spaces, and Next Generation Networking**. St. Petersburg, Russia: Springer Berlin / Heidelberg, 2012. p.391–400. (Lecture Notes in Computer Science, v.7469).

NAJI, H. Sensor fusion using hardware agents, an implementation example. In: IEEE INTERNATIONAL CONFERENCE ON INDUSTRIAL INFORMATICS, 2005., Perth. **Proceedings...** Perth:IEEE Conference Publications, 2005. v.1, n.3, p.378 – 383.

NESTINGER, S. et al. Mobile agent-based remote vision sensor fusion. In: IEEE/ASME INTERNATIONAL CONFERENCE ON MECHTRONIC AND EMBEDDED SYSTEMS AND APPLICATIONS, 2008., Beijing. **Proceedings...** Beijing:IEEE Conference Publications, 2008. v.1, n.1, p.482 –487.

NEWMAN, R.; KEMP, J. Developing wireless sensor nodes for real-world applications. In: IEEE CONFERENCE ON LOCAL COMPUTER NETWORKS, 2007., Dublin. **Proceedings...** Dublin: IEEE Conference Publications, 2007. n.2, p.858 –863.

NGUYEN, P.; SCHAU, V.; ROSSAK, W. Performance comparison of some message transport protocol implementations for agent community communication. In: INTERNATIONAL CONFERENCE ON INNOVATIVE INTERNET COMMUNITY SERVICES, Berlin. **Proceedings...** Berlin:IEEE Conference Publications, 2011. v.P.186, p.193–204.

O'BRIEN, D. et al. Design and implementation of optical wireless communications with optically powered smart dust motes. **Selected Areas in Communications, IEEE Journal on**, Oxford, v.27, n.9, p.1646 –1653, Dec. 2009.

PARK, C.; DING, Y.; BYON, E. Collaborative data reduction for energy efficient sensor networks. **IEEE International Conference on Automation Science and Engineering.**, San Antonio, p.442–447, Aug. 2008.

PEIXOTO, J. A. **Desenvolvimento de sistemas de automação da manufatura usando arquiteturas orientadas a serviço e sistemas multi-agentes**. 2012. Mestrado em Controle e Automação — Universidade Federal do Rio Grande do Sul, Escola de Engenharia, Programa de Pós-Graduação em Engenharia Elétrica, Porto Alegre, 2012.

PONCI, F.; DESHMUKH, A. A mobile agent for measurements in distributed power electronic systems. In: INSTRUMENTATION AND MEASUREMENT TECHNOLOGY CONFERENCE, 2008., Vancouver. **Proceedings...** Vancouver:IEEE Conference Publications, 2008. n.3, p.870 –875.

RFM. **Página do Datasheet do Módulo**. Disponível em: <http://www.rfm.com/products/data/tr1000.pdf>. Acesso em: 08 Nov. 2011.

RUSSELL, S.; NORVIG, P. **Artificial Intelligence**: a modern approach. 2nd.ed. New Jersey: Prentice-Hall, Englewood Cliffs, 2003.

SAAIM, E. H. M. et al. Applying mobile agent in distributed speech recognition using JADE. In: INTERNATIONAL CONFERENCE ON COMPUTERS, COMMUNICATIONS, SIGNAL PROCESSING WITH SPECIAL TRACK ON BIOMEDICAL ENGINEERING, 2005., Kuala Lumpur. **Proceedings...** Kuala Lumpur:IEEE Conference Publications, 2005. n.1, p.55 –59.

SADIK, S. et al. Policy based migration of mobile agents in disaster management systems. In: INTERNATIONAL CONFERENCE ON EMERGING TECHNOLOGIES, 2006., Alberta. **Proceedings...** Alberta:IEEE Conference Publications, 2006. n.1, p.224 –229.

SANCHEZ, A. et al. A low cost and high efficient acoustic modem for underwater sensor networks. In: OCEANS, 2011 IEEE, Kona. **Proceedings...** Kana:IEEE Conference Publications, 2011. n.1, p.1 –10.

SCHNEIDER, J.; NAGGATZ, M.; SPALLEK, R. Implementation of architecture concepts for hardware agent systems. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER AND INFORMATION TECHNOLOGY, 2007., Fukushima. **Proceedings...** Fukushima:IEEE Conference Publications, 2007. n.1, p.823 –828.

SCHOEBERL, M. **JOP**: a java optimized processor for embedded real-time systems. 2005. Doctor thesis — Vienna University of Technology, Vienna.

SEDCOLE, N. P. et al. Modular partial reconfiguration in virtex FPGAs. In: FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS, Belfast. **Proceedings...** Belfast:IEEE Conference Publications, 2005. n.1, p.211–216.

SIMON, D. Kalman Filtering. **Embedded Systems Programming**, San Francisco, p.72–79, 2001.

SUN, R.; PETERSON, T. Learning in reactive sequential decision tasks: the clarion model. In: IEEE INTERNATIONAL CONFERENCE ON NEURAL NETWORKS, 1996., Tuscaloosa, AL. **Proceedings...** Tuscaloosa:IEEE Conference Publications, 1996. v.2, n.1, p.1073 –1078 vol.2.

SUNSPOT. **Página com Informações do Projeto**. Disponível em: <http://www.sunspotworld.com/>. Acesso em: 16 Out. 2011.

TILAB. **Jade Website**. Disponível em: <http://jade.tilab.com>. Acesso em: 10 Mar. 2012.

VEERASINGAM, S. et al. Design of wireless sensor network node on zigBee for temperature monitoring. In: INTERNATIONAL CONFERENCE ON ADVANCES IN COMPUTING, CONTROL, TELECOMMUNICATION TECHNOLOGIES, 2009., Trivandrum. **Proceedings...** Trivandrum:IEEE Conference Publications, 2009. n.1, p.20 –23.

WOOLRIDGE, M.; WOOLDRIDGE, M. J. **Introduction to Multiagent Systems**. New York: John Wiley & Sons, Inc., 2001.

XBOW. **Crossbow Website**. Disponível em: <http://www.xbow.com/asset-tracking/technology/index.html>. Acesso em: 21 Out. 2011.

XILINX. **Manual de Configuração do FPGA**. Disponível em: <http://www.xilinx.com/support/documentation/user_guides/ug360.pdf>. Acesso em: 16 Jun. de 2012.

YANG, G.-Z. **Body Sensor Networks**. Secaucus: Springer-Verlag New York, 2006.

ZHOU, Q.-L. et al. Dead reckoning and Kalman filter design for trajectory tracking of a quadrotor UAV. In: IEEE/ASME INTERNATIONAL CONFERENCE ON MECHATRONICS AND EMBEDDED SYSTEMS AND APPLICATIONS, 2010., Qingdao. **Proceedings...** Qingdao:IEEE Conference Publications, 2010. n.2, p.119 –124.

ZURICH, E. **BTNode Website**. Disponível em: <http://www.btnode.ethz.ch/>. Acesso em: 21 Out. 2011.

APÊNDICE A LISTAGENS DOS AGENTES

```
1module FIPA {
2 typedef string URL;
3 struct Property {
4  string keyword;
5  any value;
6 };
7 struct AgentID {
8  string name;
9  sequence<URL>      addresses;
10  sequence<AgentID>  resolvers;
11  sequence<Property> userDefinedProperties;
12 };
13 typedef sequence<AgentID> AgentIDs;
14  struct DateTime {
15  short year;
16  short month;
17  short day;
18  short hour;
19  short minutes;
20  short seconds;
21  short milliseconds;
22  char  typeDesignator;
23 };
24 struct ReceivedObject {
25  URL by;
26  URL from;
27  DateTime date;
28  string id;
29  string via;
30 };
31 typedef sequence<Property> TransportBehaviourType;
32 typedef sequence<AgentID,1> OptAgentID;
33 typedef sequence<DateTime,1> OptDateTime;
34 typedef sequence<TransportBehaviourType,1>
     OptTransportBehaviourType;
35 typedef sequence<ReceivedObject,1> OptReceivedObject;
36 struct Envelope {
37  AgentIDs                  to;
38  OptAgentID                from;
39  string                    comments;
40  string                    aclRepresentation;
41  long                      payloadLength;
42  string                    payloadEncoding;
43  OptDateTime               date;
```

```
44   AgentIDs                    intendedReceiver;
45   OptReceivedObject           received;
46   OptTransportBehaviourType   transportBehaviour;
47   sequence<Property>          userDefinedProperties;
48  };
49  typedef sequence<Envelope> Envelopes;
50  typedef sequence<octet> Payload;
51  struct FipaMessage {
52   Envelopes messageEnvelopes;
53   Payload   messageBody;
54  };
55  interface MTS {
56   oneway void message(in FipaMessage aFipaMessage);
57  };
58 };
```

Listagem A.1: Classe que encapsula as mensagens no ACC

```
1 typedef struct fipa_acl_message_s
2 {
3   enum fipa_performative_e performative;
4   struct fipa_agent_identifier_s *sender;
5   struct fipa_agent_identifier_set_s *receiver;
6   int receiver_num;
7   struct fipa_agent_identifier_set_s *reply_to;
8   struct fipa_string_s *content;
9   struct fipa_expression_s *language;
10  struct fipa_expression_s *encoding;
11  struct fipa_expression_s *ontology;
12  enum fipa_protocol_e protocol;
13  struct fipa_expression_s *conversation_id;
14  struct fipa_expression_s *reply_with;
15  struct fipa_expression_s *in_reply_to;
16  struct fipa_DateTime_s *reply_by;
17 } fipa_acl_message_t;
```

Listagem A.2: Exemplo da estrutura de dados ACL em linguagem C

APÊNDICE B WRAPPER DE AGENTES

```
1#include <stdio.h>
2#include <stdlib.h>
3#include <sys/stat.h>
4
5#include "temperature.h"
6#include "tcputils.h"
7#include "packetAPI.h"
8#include "ethstat.h"
9
10//#define HOSTAPP_DEBUG
11#ifdef HOSTAPP_DEBUG
12#include "debug_on.h"
13#else
14#include "debug_off.h"
15#endif
16
17enum {
18 PRINT_PACK = 0x01,
19 TEMP_GET   = 0x02,
20#ifdef PC_CONFIG
21 TEMP_PC_SET = 0x03,
22#endif /*PC_CONFIG*/
23 ETHSTAT_GET = 0x04,
24} WRAPPER_SERVICES;
25
26typedef struct {
27 unsigned char a;
28 unsigned char b;
29 unsigned char c;
30 unsigned char d;
31}st_a;
32
33#ifndef PC_CONFIG
34static unsigned int byteInvertEndianess(unsigned int b)
35{
36 unsigned int ret;
37 unsigned char tmp[sizeof(ret)];
38 unsigned char *ptr = (unsigned char *)&b;
39 unsigned int sz = sizeof(b) - 1;
40 int i;
41
42 for (i = 0; i < sizeof(b); i++) {
43  printd("tmp[%d] = 0x%x 0x%x\n", i, ptr[sz-i], ptr[i]);
44  tmp[i] = ptr[sz-i];
```

```
45 }
46
47 memcpy(&ret, tmp, sizeof(ret));
48
49 return ret;
50}
51#endif
52
53static void tempPackAssemble(st_tcpserverData *pack, st_temp
      temp)
54{
55 pack->header.command = TEMP_GET;
56 pack->header.size = sizeof(temp);
57
58 memcpy(pack->data, &temp, sizeof(temp));
59}
60
61#ifdef PC_CONFIG
62static void tempPackDisassemble(st_temp *temp,
     st_tcpserverData pack)
63{
64 memcpy(temp, pack.data, sizeof(*temp));
65}
66#endif
67
68static void ethstatPackAssemble(st_tcpserverData *pack,
     st_ethstat st)
69{
70 pack->header.command = ETHSTAT_GET;
71 pack->header.size = sizeof(st);
72
73 memcpy(pack->data, &st, sizeof(st));
74}
75
76static int treatClientPack(st_tcpserverData pack, int sockfd
     , struct sockaddr_in sa, struct sockaddr_in sac)
77{
78 int ret = 0;
79 st_a a;
80
81 a.a = 0;
82 a.b = 0;
83 a.c = 0;
84 a.d = 1;
85
86 switch(pack.header.command){
87  case PRINT_PACK:
88   printf("test\n\r");
89   socket_tcp_send(sockfd, &a, sizeof(a));
90   break;
91  case TEMP_GET: {
92   st_temp localtemp;
93   st_tcpserverData answer;
94
95   printd("GET TEMP!\n");
96   bzero(&answer, sizeof(answer));
97   bzero(&localtemp, sizeof(localtemp));
98
```

```
 99    /*get the temperature*/
100    temperature_get(&localtemp);
101
102    /*mount the packet*/
103    tempPackAssemble(&answer, localtemp);
104
105    /*send data*/
106    tcpSendPack(sockfd, sa, &answer);
107    break;
108   }
109#ifdef PC_CONFIG
110   case TEMP_PC_SET: {
111    st_temp localtemp;
112
113    bzero(&localtemp, sizeof(localtemp));
114
115    printd("SET PC TEMP!\n");
116    /*process received packet*/
117    tempPackDisassemble(&localtemp, pack);
118
119    /*set the temperature*/
120    temperature_PC_set(localtemp);
121    break;
122     }
123#endif /*PC_CONFIG*/
124   case ETHSTAT_GET: {
125    st_ethstat st;
126    const char *dev = "eth1"; /*FIXME*/
127    st_tcpserverData answer;
128
129    bzero(&st, sizeof(st));
130    ethstat_get_txrx(&st, dev);
131
132#ifndef PC_CONFIG
133    st.tx_bytes = byteInvertEndianess(st.tx_bytes);
134    st.rx_bytes = byteInvertEndianess(st.rx_bytes);
135#endif
136    printf("tx: %d rx: %d\n", st.tx_bytes, st.rx_bytes);
137
138    ethstatPackAssemble(&answer, st);
139    tcpSendPack(sockfd, sa, &answer);
140    break;
141   }
142   default:
143    printd("Invalid code %d\n", pack->header.command);
144    break;
145  }
146
147  return ret;
148}
149
150#ifndef PC_CONFIG
151
152static int fpgaProgram(void)
153{
154  unsigned char a;
155  int b;
156  int ret;
```

```
157 const char fname[] = "image.bin.gz"; /*FIXME*/
158 int tries=10;
159 struct stat buf;
160 double t = 0;
161 int i;
162
163 if (fpga_init() != 0 || fpga_open_all() != 0) {
164  fprintf(stderr, "error initializing fpga api!\n");
165  return -1;
166 }
167
168 if(stat(fname, &buf)) {
169  fprintf(stderr, "error in stat..\n");
170  return -1;
171 }
172
173 printf("Programming fpga.. please wait...\n");
174 t = getTimeMilisec();
175 while((ret=fpga_prog(fname, 0, 2)) < 0 &&(--tries))
176  fprintf(stderr, "error programming fpga: %d %d\n", ret,
        tries);
177 printf("t = %3.2f\n\r", getTimeMilisec() - t);
178
179 if(ret < 0)
180  return -1;
181 t = getTimeMilisec();
182 for (i = 0; i < 1000; i++)
183  b = fpga_read_byte(0, 0x101, &a);
184 printf("tread = %3.2f\n\r", getTimeMilisec() - t);
185
186 printf("ret: %d value: 0x%x\n", b, a);
187
188 return 0;
189}
190
191#endif /*PC_CONFIG*/
192
193int main(int argc, char *argv[])
194{
195 unsigned int port;
196 st_temp_shared th;
197 st_tcpserver server;
198 pthread_t tcplisten, temperature;
199 pthread_attr_t tcplisten_at, temperature_at;
200
201 if(argc < 2) {
202  fprintf(stderr, "Usage ./hostapp <port>\n");
203  return -1;
204 }
205
206#ifndef PC_CONFIG
207 if(fpgaProgram() < 0)
208  return -1;
209 th.thermal_status = &embThermalStatus;
210#else
211 th.thermal_status = &pcThermalStatus;
212#endif
213
```

```
214 port = atoi(argv[1]);
215 printf("port: %d\n", port);
216
217 server.treat_clientpack = &treatClientPack;
218 server.port = port;
219
220 __PTHREAD_CREATE_FUNC_(tcplisten, tcplisten_at, tcpListener
       , server);
221 __PTHREAD_CREATE_FUNC_(temperature, temperature_at,
       temp_thread, th);
222
223 pthread_join(tcplisten, NULL);
224 pthread_join(temperature, NULL);
225
226 return 0;
227}
```

Listagem B.1: Núcleo da implementação do Wrapper.

APÊNDICE C MÓDULO FPU

```
1 import java.util.*;
2 import java.io.*;
3 import java.net.*;
4 import java.lang.Math.*;
5
6 import jade.core.*;
7 import jade.core.Agent;
8 import jade.core.*;
9 import jade.core.behaviours.*;
10 import jade.lang.acl.*;
11 import jade.content.*;
12 import jade.content.lang.*;
13 import jade.content.lang.sl.*;
14 import jade.content.onto.basic.*;
15 import jade.domain.*;
16 import jade.domain.JADEAgentManagement.*;
17
18 import def.*;
19
20 public class FpuAgent extends Agent
21 {
22  @Override
23   protected void setup()
24   {
25    addBehaviour(new FpuAgentBehaviour(this));
26   }
27 }
28
29 class FpuAgentBehaviour extends SimpleBehaviour
30 {
31  double opa, opb, result;
32  int operation;
33  int txtmp, rxtmp;
34
35  /* Constructor */
36  public FpuAgentBehaviour(Agent agent)
37  {
38   super(agent);
39   this.opa = 0;
40   this.opb = 0;
41   this.result = 0;
42   this.operation = 0;
43   this.txtmp = 0;
44   this.rxtmp = 0;
```

```
45  System.out.println("name: " + agent.getLocalName() + "@" +
        agent.getHap());
46 }
47
48 public boolean done()
49 {
50  return false;
51 }
52
53 @Override
54  public void action()
55  {
56   MessageTemplate mt = MessageTemplate.MatchPerformative(
        ACLMessage.REQUEST);
57   ACLMessage aclmessage = myAgent.receive(mt);
58   FpuAnswerDataStruct f = new FpuAnswerDataStruct();
59
60   if(aclmessage != null) {
61    try{
62     RequesterDataStruct req = (RequesterDataStruct)
          aclmessage.getContentObject();
63     this.opa = req.getOpa();
64     this.opb = req.getOpb();
65     this.operation = req.getOperation();
66
67     treatOperation();
68     f.setResult(this.result);
69
70     treatCmd(def.Agents.GET_ETH_STAT_BEFORE);
71     answerRequest(f);
72     treatCmd(def.Agents.GET_ETH_STAT_AFTER);
73    } catch (Exception e) {
74     e.printStackTrace();
75    }
76   }
77   else
78    this.block();
79  }
80
81 void treatOperation()
82 {
83  switch(this.operation) {
84   case def.Agents.FPU_ADD:
85    this.result = double_add(this.opa,this.opb);
86    break;
87
88   case def.Agents.FPU_SUB:
89    this.result = double_sub(this.opa,this.opb);
90    break;
91
92   case def.Agents.FPU_MUL:
93    this.result = double_mul(this.opa,this.opb);
94    break;
95
96   case def.Agents.FPU_DIV:
97    this.result = double_div(this.opa,this.opb);
98    break;
99
```

```
100    default:
101     System.out.println("Error receiving command!");
102     this.result = 0;
103     break;
104   }
105 }
106
107 void answerRequest(FpuAnswerDataStruct f)
108 {
109  ACLMessage aclmsg = new ACLMessage(ACLMessage.INFORM);
110  AID r = new AID("req_agent", AID.ISLOCALNAME);
111
112  System.out.println("Result is "+f.getResult());
113  aclmsg.addReceiver(r);
114  try {
115   aclmsg.setContentObject(f);
116  } catch(Exception e) {
117   e.printStackTrace();
118  }
119  this.myAgent.send(aclmsg);
120 }
121
122 private void treatCmd(int cmd)
123 {
124  String ip = def.Agents.DEFAULT_IP;
125  int port = def.Agents.DEFAULT_PORT;
126  WrapperServices wp = new WrapperServices();
127
128  switch(cmd) {
129   case def.Agents.GET_ETH_STAT_BEFORE:
130    wp.ethStatsGet(ip, port);
131    this.txtmp = wp.ethTxGet();
132    this.rxtmp = wp.ethRxGet();
133    break;
134   case def.Agents.GET_ETH_STAT_AFTER:
135    wp.ethStatsGet(ip, port);
136    int tx_bytes = wp.ethTxGet() - this.txtmp;
137    int rx_bytes = wp.ethRxGet() - this.rxtmp;
138    System.out.println("TX: "+ tx_bytes);
139    System.out.println("RX: "+ rx_bytes);
140    break;
141   default:
142    break;
143  }
144 }
145}
```

Listagem C.1: Agente FPU implementado em *software*.

```
1#define REG_BASE    0x0200 /*Endereço base*/
2#define REG_LEN     256    /*Número de registradores*/
3
4#define DATA_A_RESERVED_REG  REG_BASE + 0x00
5#define DATA_B_RESERVED_REG  REG_BASE + 0x01
6#define FPU_CTRL    REG_BASE + 0x02 /* enable/reset */
7#define FPU_OP_REG    REG_BASE + 0x03 /*3 bits*/
8#define OPA_0_7_REG    REG_BASE + 0x04
9#define OPA_8_15_REG   REG_BASE + 0x05
10#define OPA_16_23_REG   REG_BASE + 0x06
```

```
11#define OPA_24_31_REG   REG_BASE + 0x07
12#define OPA_32_39_REG   REG_BASE + 0x08
13#define OPA_40_47_REG   REG_BASE + 0x09
14#define OPA_48_55_REG   REG_BASE + 0x0A
15#define OPA_56_63_REG   REG_BASE + 0x0B
16#define OPB_0_7_REG    REG_BASE + 0x0C
17#define OPB_8_15_REG   REG_BASE + 0x0D
18#define OPB_16_23_REG   REG_BASE + 0x0E
19#define OPB_24_31_REG   REG_BASE + 0x0F
20#define OPB_32_39_REG   REG_BASE + 0x10
21#define OPB_40_47_REG   REG_BASE + 0x11
22#define OPB_48_55_REG   REG_BASE + 0x12
23#define OPB_56_63_REG   REG_BASE + 0x13
24#define FP_RES_0_7_REG   REG_BASE + 0x14
25#define FP_RES_8_15_REG   REG_BASE + 0x15
26#define FP_RES_16_23_REG  REG_BASE + 0x16
27#define FP_RES_24_31_REG  REG_BASE + 0x17
28#define FP_RES_32_39_REG  REG_BASE + 0x18
29#define FP_RES_40_47_REG  REG_BASE + 0x19
30#define FP_RES_48_55_REG  REG_BASE + 0x1A
31#define FP_RES_56_63_REG  REG_BASE + 0x1B
32#define FPU_SIGNALS_REG   REG_BASE + 0x1C
33
34typedef union {
35 struct {
36  unsigned char res1   : 3;
37  unsigned char reset  : 1;
38  unsigned char res2   : 3;
39  unsigned char enable : 1;
40 } bit;
41 unsigned char reg;
42} st_fpu_ctrl;
43
44typedef union {
45 struct {
46  unsigned char res1 : 5;
47  unsigned char op   : 3;
48 } bit;
49 unsigned char reg;
50} st_fpu_operations;
51
52typedef union {
53 struct {
54  unsigned char res   : 7;
55  unsigned char ready : 1;
56 } bit;
57 unsigned char reg;
58} st_fpu_signals;
59
60typedef union {
61 unsigned char byte[sizeof(double)];
62 double reg;
63} st_double_reg;
64
65static int double_reg_write(double reg, unsigned char op)
66{
67 int i;
68 st_double_reg r;
```

```
69
70  unsigned int add[2][8] = {
71  {
72   OPA_0_7_REG,
73   OPA_8_15_REG,
74   OPA_16_23_REG,
75   OPA_24_31_REG,
76   OPA_32_39_REG,
77   OPA_40_47_REG,
78   OPA_48_55_REG,
79   OPA_56_63_REG
80  },
81  {
82   OPB_0_7_REG,
83   OPB_8_15_REG,
84   OPB_16_23_REG,
85   OPB_24_31_REG,
86   OPB_32_39_REG,
87   OPB_40_47_REG,
88   OPB_48_55_REG,
89   OPB_56_63_REG
90  }
91 };
92
93 bzero(&r, sizeof(st_double_reg));
94 r.reg = reg;
95
96 for (i = 0; i < sizeof(double); i++) {
97  unsigned char val = r.byte[sizeof(double)-1-i];
98  fpga_write_byte_now(0, add[op][i], val);
99 }
100 return 0;
101}
102
103static int double_reg_read(double *reg)
104{
105 int i;
106 st_double_reg r;
107
108 unsigned int add[8] = {
109  FP_RES_0_7_REG  ,
110  FP_RES_8_15_REG ,
111  FP_RES_16_23_REG,
112  FP_RES_24_31_REG,
113  FP_RES_32_39_REG,
114  FP_RES_40_47_REG,
115  FP_RES_48_55_REG,
116  FP_RES_56_63_REG
117 };
118
119 bzero(&r, sizeof(st_double_reg));
120
121 for (i = 0; i < sizeof(double); i++) {
122  unsigned char val = 0;
123  fpga_read_byte(0, add[i], &val);
124  printf("reg[%d]: 0x%x add 0x%x\n", i, val, add[i]);
125  r.byte[sizeof(double)-1-i] = val;
126 }
```

```
127 *reg = r.reg;
128
129 return 0;
130}
131
132int fpu_double_map(double opa, double opb, int operation)
133{
134 st_fpu_ctrl fpu_ctrl;
135 st_fpu_operations fpu_op;
136 st_fpu_signals fpu_sign;
137 double res = 0.0;
138
139 bzero(&fpu_ctrl, sizeof(st_fpu_ctrl));
140 bzero(&fpu_op, sizeof(st_fpu_operations));
141 bzero(&fpu_sign, sizeof(st_fpu_signals));
142
143 /*write reg1*/
144 double_reg_write(opa, 0);
145
146 /*write reg2*/
147 double_reg_write(opb, 1);
148
149 /*write op*/
150 fpu_op.bit.op = operation;
151 fpga_write_byte(0, FPU_OP_REG, fpu_op.reg);
152
153 /*enable = 1*/
154 fpu_ctrl.bit.enable = 1;
155 fpga_write_byte_now(0, FPU_CTRL, fpu_ctrl.reg);
156
157 fpga_read_byte(0, FPU_SIGNALS_REG, &fpu_sign.reg);
158 /*read ready*/
159 while (!fpu_sign.bit.ready)
160  fpga_read_byte(0, FPU_SIGNALS_REG, &fpu_sign.reg);
161
162 /*read result*/
163 double_reg_read(&res);
164
165 /*enable = 0*/
166 fpu_ctrl.bit.enable = 0;
167 fpga_write_byte_now(0, FPU_CTRL, fpu_ctrl.reg);
168
169 return 0;
170}
```

Listagem C.2: Mapeamento de dados do processador para o espaço de memória do FPGA.

APÊNDICE D FILTRO DE KALMAN

```
1 import java.lang.Math;
2 import java.util.Random;
3
4 import java.io.Writer;
5 import java.io.*;
6 import java.io.FileWriter;
7
8 import Jama.Matrix;
9
10 import org.hyperic.sigar.Cpu;
11 import org.hyperic.sigar.CpuPerc;
12 import org.hyperic.sigar.CpuInfo;
13 import org.hyperic.sigar.FileSystem;
14 import org.hyperic.sigar.Mem;
15 import org.hyperic.sigar.ProcMem;
16 import org.hyperic.sigar.ProcCpu;
17 import org.hyperic.sigar.CpuPerc;
18 import org.hyperic.sigar.CpuTimer;
19 import org.hyperic.sigar.Sigar;
20 import org.hyperic.sigar.SigarException;
21
22 /**
23  * \brief Cpu information class
24  */
25 class cpuInfo {
26  private double [] perc;
27
28  public cpuInfo(int size) {
29   perc = new double[size];
30  }
31
32  public void setPerc(CpuPerc pcpu, int ts) {
33   double p = pcpu.getCombined();
34   perc[ts] = p;
35  }
36  public void setPercDiffLast(CpuPerc pcpu, int ts) {
37   double pl = getCurrPerc(ts);
38   double p = pcpu.getCombined();
39   perc[ts] = p-pl;
40  }
41  public double getCurrPerc(int ts) {
42   return perc[ts];
43  }
44  public double [] getPercPtr() {
```

```
45   return perc;
46  }
47 }
48
49 /**
50  * \brief  Class to evaluate cpu timing
51  */
52 class cpuTimer {
53  private double [] us;
54
55  public cpuTimer(int size) {
56   us = new double[size];
57  }
58
59  public void setPerc(CpuTimer pcpu, int ts) {
60   double p = pcpu.getLastSampleTime();
61   this.us[ts] = p;
62  }
63  public double getCurrPerc(int ts) {
64   return this.us[ts];
65  }
66  public double [] getPercPtr() {
67   return this.us;
68  }
69 }
70
71 /**
72  * \brief Class to evaluate memory usage
73  */
74 class memInfo {
75  private double [] res;
76  private double [] shr;
77  private double [] siz;
78  private double [] maj;
79  private double [] min;
80  private double [] pag;
81
82  public memInfo(int size) {
83   res = new double[size];
84   shr = new double[size];
85   siz = new double[size];
86   maj = new double[size];
87   min = new double[size];
88   pag = new double[size];
89  }
90
91  public void setMem(ProcMem pm, int ts) {
92   this.min[ts] = pm.getMinorFaults();
93   this.pag[ts] = pm.getPageFaults();
94   this.res[ts] = pm.getResident() / 1024;
95   this.shr[ts] = pm.getShare() / 1024;
96   this.siz[ts] = pm.getSize() / (1024*1024);
97  }
98
99  public double [] memInfoResPtrGet() {
100  return this.res;
101  }
102  public double [] memInfopfaultPtrGet() {
```

```
103  return this.pag;
104 }
105 public double [] memInfoShrPtrGet() {
106  return this.shr;
107 }
108 public double [] memInfoSizPtrGet() {
109  return this.siz;
110 }
111}
112
113
114/**
115 * \brief  Main Class: Kalman filter.
116 */
117public class Kalman {
118 static double ACCELNOISE = 1.0;
119 static double MEASNOISE = 0.2;
120
121 static void kalman(int duration, double dt)
122 {
123  double sizeaux = (double)duration/dt;
124  int size = (int)sizeaux+1;
125  Sigar sigar = new Sigar();
126  long pid = sigar.getPid();
127  CpuPerc pcpu = null;
128  ProcMem pm = new ProcMem();
129  CpuTimer pct = new CpuTimer();
130
131  try {
132   pcpu = sigar.getCpuPerc();
133  }
134  catch(SigarException se) {
135   se.printStackTrace();
136  }
137
138  try {
139   pm.gather(sigar, pid);
140  } catch (SigarException se) {
141   se.printStackTrace();
142  }
143
144  CpuInfo[] cpuinfo = null;
145  try {
146   cpuinfo = sigar.getCpuInfoList();
147  } catch (SigarException se) {
148   se.printStackTrace();
149  }
150
151  for (int i = 0; i < cpuinfo.length; i++) {
152   System.out.println("CPU " + i + ": " + cpuinfo[i].toMap()
       );
153  }
154
155  Matrix a = new Matrix(2,2); /*transition matrix*/
156  Matrix b = new Matrix(2,1); /*input matrix*/
157  Matrix c = new Matrix(1,2); /*measurement matrix*/
158  Matrix x = new Matrix(2,1); /*state vector*/
159  Matrix xhat = new Matrix(2,1); /*state estimate*/
```

```
160
161  /*Initial values*/
162  a.set(0, 0, 1);
163  a.set(0, 1, dt);
164  a.set(1, 0, 0);
165  a.set(1, 1, 1);
166
167  b.set(0, 0, Math.pow(dt,2)/2);
168  b.set(1, 0, dt);
169
170  c.set(0, 0, 1);
171  c.set(0, 1, 0);
172
173  /*auxiliary matrix*/
174  Matrix processNoise = new Matrix(2,1);
175  Matrix mrand = new Matrix(2,1);
176  Matrix s = new Matrix(1,1);
177  Matrix k = new Matrix(1,1);
178  Matrix innovation = new Matrix(1,1);
179  Matrix y = new Matrix(1,1);
180  Matrix measNoise = new Matrix(1,1);
181
182  /*Covariance of the estimation error*/
183  Matrix P = new Matrix(2,2);
184
185  Matrix covZ = new Matrix(1,1);
186  covZ.set(0, 0, Math.pow(MEASNOISE,2)); /*Measurement error
         covariance*/
187
188  Matrix covW = new Matrix(2,2); /*process noise covariance
       */
189  covW.set(0, 0, Math.pow(dt,4)/4);
190  covW.set(0, 1, Math.pow(dt,3)/2);
191  covW.set(1, 0, Math.pow(dt,3)/2);
192  covW.set(1, 1, Math.pow(dt,2));
193  covW = covW.times(Math.pow(ACCELNOISE,2));
194
195  double [] pos = new double[size];
196  double [] poshat = new double[size];
197  double [] posmeas = new double[size];
198  double [] vel = new double[size];
199  double [] velhat = new double[size];
200
201  cpuInfo ci = new cpuInfo(size);
202  memInfo mi = new memInfo(size);
203  cpuTimer ctu = new cpuTimer(size);
204
205  double xx;
206  double yy;
207
208  double xy = System.nanoTime();
209  Random rdx = new Random(System.currentTimeMillis());
210  xx = rdx.nextDouble();
211  yy = rdx.nextDouble();
212  double xya = xx*yy;
213  double xy1 = System.nanoTime() - xy;
214
215  System.out.println("t: " +xy1);
```

```
216
217  long ab = System.currentTimeMillis();
218  for(int t=0; t < size; t++) {
219   double u;
220   Random rd = new Random(System.currentTimeMillis());
221
222   /*acceleration is contant: 1*/
223   u = 0.1;
224
225   /*Simulate the linear system*/
226   mrand.set(0, 0, Math.pow(dt,2)/2*rd.nextDouble());
227   mrand.set(1, 0, dt*rd.nextDouble());
228
229   processNoise = mrand.times(ACCELNOISE);
230   //x = a*x + b*u + processNoise;
231   x = a.times(x);
232   x.plusEquals(b.times(u));
233   x.plusEquals(processNoise);
234
235   /*Simulate the noisy measurement*/
236   measNoise.set(0,0, MEASNOISE * rd.nextDouble());
237   //y = c*x + measNoise;
238   y = c.times(x);
239   y.plusEquals(measNoise);
240
241   /*Extrapolate the most recent state estimate to the
        present time*/
242   //xhat = a*xhat + b*u;
243   xhat = a.times(xhat);
244   xhat.plusEquals(b.times(u));
245
246   /*Form the innovation vector*/
247   //innovation = y - c*xhat;
248   innovation = y.minus(c.times(xhat));
249
250   /*Compute the covariance and the innovation*/
251   //s = c*P*c.t() + covZ;
252   Matrix saux = new Matrix(1,2);
253   saux = c.times(P);
254   s = saux.times(c.transpose());
255   s.plusEquals(covZ);
256
257   /*Form the Kalman gain matrix*/
258   //k = a*P*c.t() * s.inv();
259   Matrix kaux1 = new Matrix(2,1);
260   Matrix kaux2 = new Matrix(2,2);
261   kaux2 = a.times(P);
262   kaux1 = kaux2.times(c.transpose());
263   k = kaux1.times(s.inverse());
264
265   /*Update the state estimate*/
266   //xhat = xhat + k*innovation;
267   xhat.plusEquals(k.times(innovation));
268
269   //CPU
270   ci.setPerc(pcpu, t);
271   ctu.setPerc(pct, t);
272   //mem
```

```
273    mi.setMem(pm, t);
274
275    /*Compute the covariance of the estimation error*/
276    //P = a*P*a.t() - a*P*c.t() * s.inv() * c*P*a.t() + covW;
277    Matrix Poriginal = new Matrix(2,2);
278    Poriginal = P;
279    P = a.times(P);
280    P = P.times(a.transpose()); // a*P*a'
281    Matrix Paux1 = new Matrix(2,2);
282       Paux1 = a.times(Poriginal); // a*P*c
283
284    Matrix Paux2 = new Matrix(2,1);
285    Paux2 = Paux1.times(c.transpose());// a*P*c'
286    Paux2 = Paux2.times(s.inverse()); // a*P*c'*inv(s)
287
288    Paux1 = c.times(Poriginal); // c*P
289    Paux1.times(a.transpose()); // c*P*a'
290    Paux1 = Paux2.times(Paux1); // a*P*c'*inv(s)*c*P*a'
291    Paux1.plusEquals(covW); // a*P*c'*inv(s)*c*P*a'+ covW
292
293    P.minusEquals(Paux1);
294
295    pos[t] = x.get(0,0);
296    posmeas[t] = y.get(0,0);
297    poshat[t] = xhat.get(0,0);
298    vel[t] = x.get(1,0);
299    velhat[t] = xhat.get(1,0);
300
301   }
302   long bc = System.currentTimeMillis();
303   long d = bc - ab;
304   System.out.println("time: " +  d);
305
306   kalman_savetofile("data/pos.dat", pos, dt);
307   kalman_savetofile("data/posmeas.dat", posmeas, dt);
308   kalman_savetofile("data/poshat.dat", poshat, dt);
309   kalman_savetofile("data/vel.dat", vel, dt);
310   kalman_savetofile("data/velhat.dat", velhat, dt);
311
312   kalman_savetofile("data/perc.dat", ci.getPercPtr(), dt);
313   kalman_savetofile("data/res.dat", mi.memInfoResPtrGet(),
         dt);
314   kalman_savetofile("data/pfault.dat", mi.
         memInfopfaultPtrGet(), dt);
315   kalman_savetofile("data/shr.dat", mi.memInfoShrPtrGet(),
         dt);
316   kalman_savetofile("data/siz.dat", mi.memInfoSizPtrGet(),
         dt);
317   kalman_savetofile("data/ctu.dat", ctu.getPercPtr(), dt);
318  }
319
320  static void kalman_savetofile(String fname, double [] d,
        double dt)
321  {
322   BufferedWriter bufferedWriter = null;
323   try {
324    //Construct the BufferedWriter object
325    bufferedWriter = new BufferedWriter(new FileWriter(fname)
```

```
        );
326    int tmax = d.length;
327    double currentTime = 0;
328    double v = 0;
329
330    for (int t = 0; t < tmax; t++) {
331     currentTime = t*dt;
332     bufferedWriter.write(Double.toString(currentTime));
333     bufferedWriter.write("\t");
334     bufferedWriter.write(Double.toString(d[t]));
335     bufferedWriter.write("\n");
336    }
337   } catch (FileNotFoundException ex) {
338    ex.printStackTrace();
339   } catch (IOException ex) {
340    ex.printStackTrace();
341   } finally {
342    //Close the BufferedWriter
343    try {
344     if (bufferedWriter != null) {
345      bufferedWriter.flush();
346      bufferedWriter.close();
347     }
348    } catch (IOException ex) {
349     ex.printStackTrace();
350    }
351   }
352 }
353
354 public static void main(String[] args)
355 {
356  kalman(100,0.01);
357 }
358}
```

Listagem D.1: Implementação do filtro de Kalman em Java.

APÊNDICE E ARQUITETURA DE HARDWARE

Este capítulo descreve a arquitetura de hardware deste trabalho onde é descrito em detalhes as conexões pertinentes entre os blocos de *hardware* utilizados. Neste capítulo é também descrita a maneira de programação do FPGA e o canal de comunicação entre o processador e o FPGA, passando por um mapeamento em memória dos blocos implementados dentro do FPGA.

Todo desenvolvimento foi feito sobre um hardware com as seguintes características mostradas na Tabela 11.

Tabela 11: Características físicas do sistema embarcado de testes.

Processador Principal	Freescale MPC8314E - PowerPC e300.
Memória	128 MBytes SDRAM.
FPGA	Xilinx Virtex6 LXT 130.
NAND Flash	1 Gbit.

A Figura 41 mostra as ligações físicas pertinentes entre os dispositivos que compõem a arquitetura deste projeto. O processador principal ligado diretamente à uma memória externa e ao FPGA. A ligação entre o FPGA e o processador é feita por um barramento que está ligado ao *localbus controller* (LBC) do processador utilizado (FREESCALE, 2012).

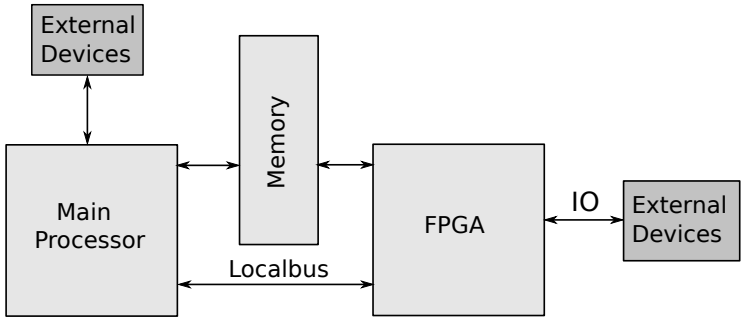

Figura 41: Conexões entre processador, memória e FPGA.

E.1 Mapeamento em memória

Os dispositivos implementados em lógica programável no FPGA são mapeados em memória do ponto de vista do processador. O LBC provê a interface aos peri-

féricos de forma que o trabalho do *driver* de escrita e leitura é somente atuar sobre este controlador. Todo o controle de sinalização entre processador e FPGA é feito através deste bloco diretamente em hardware.

O LBC provê diferentes maneiras de criar interfaces entre dispositivos que são conectados a ele, o que permite a implementação de sistemas de memória com requisitos de temporização bem específicos. As três principais interfaces utilizadas por este controlador são o *General Purpose Chip-select Machine* (GPCM), a máquina SDRAM e as *User Programmable Machines* (UPMs). Quando uma transação de acesso a um dispositivo de memória é entregue ao LBC, o endereço de memória é comparado para ativar o *chip select* da máquina que estará sendo utilizada.

A máquina SDRAM provê uma interface para SDRAMs usando técnicas específicas para se obter alta performance sobre um canal multiplexado de dados e endereços. A máquina GPCM provê uma interface para dispositivos mais simples e de baixa performance, como por exemplo memórias e outros dispositivos que podem ser mapeados em memória. Ele é de baixa performance pois não suporta acesso em rajadas (*bursting access*). Por esta razão, a máquina GPCM é usada primariamente na fase de *boot loading* do sistema e também em dispositivos mapeados em memória de baixa performance. A máquina UPM suporta praticamente qualquer outro tipo de periférico, inclusive os que têm requisitos de alta performance. É necessário entretanto uma pequena lógica de cola para alguns sinais deste barramento quando utilizado neste modo.

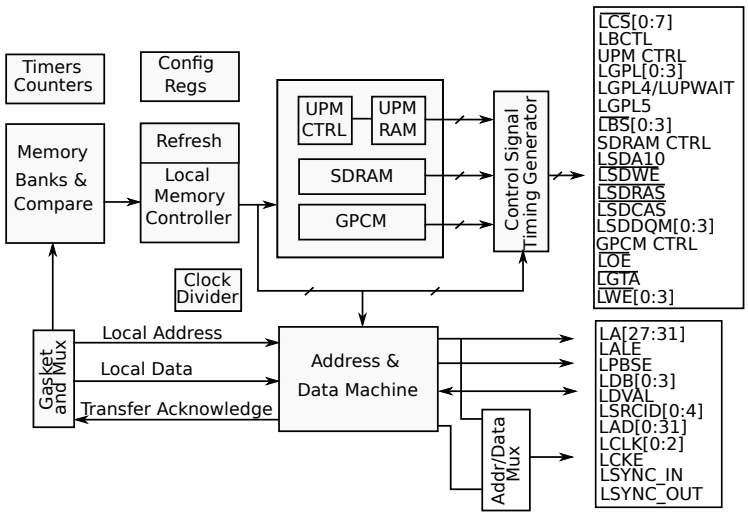

Figura 42: Diagrama de blocos do controlador *localbus*.

A Figura 42 mostra o diagrama de blocos do LBC (FREESCALE, 2012). O FPGA é acessado através das UPMs deste bloco sendo que o *driver* de acesso a este bloco do processador já é fornecido pelo fabricante do processador. Este código pode ser encontrado em arch/powerpc/sysdev/fsl_lbc.c diretamente no Kernel do Linux.

Do ponto de vista do FPGA, é necessário ler os sinais das ligações com o processador e registrá-los conforme a necessidade. Foi criado um multiplexador de leitura e escrita na arquitetura topo do *design* em lógica programável de forma que é possível selecionar o endereço desejado no sinal do barramento de endereços do *localbus* e a partir da operação desejada, leitura ou escrita, a lógica acessa a memória no endereço desejado. Desta forma é possível que o processador principal veja módulos implementados em lógica programável sobre a forma de endereços mapeados em

memória.

E.2 Programação do FPGA

O FPGA é programado através do processador principal e para isso é utilizado o modo *slave serial*, conforme o manual de configuração do dispositivo (XILINX, 2012). São utilizados alguns pinos de GPIO para fazer a comunicação serial entre o processador e o FPGA. A Figura 43 mostra as conexões físicas entre processador e FPGA para fins de programação deste dispositivo.

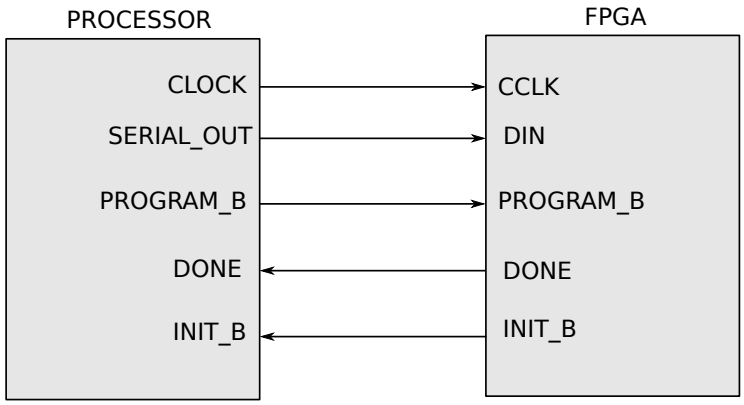

Figura 43: Conexões necessárias para programação do FPGA.

São necessários alguns pinos de entrada e saída do processador para a configuração do FPGA e para isso são utilizados os terminais de *General Purpose Input-Output* (GPIO) do processador. Através destes pinos é feita a comunicação do processador com o FPGA para fins de enviar-se o *bitstream* ao FPGA. Os dados saem do processador no terminal indicado por SERIAL_OUT sendo que um bit é transferido a cada pulso de *clock*. Isso fica mais claro na Figura 44. É necessário dar um pulso nos sinais PROGRAM_B e INIT_B antes de se começar a enviar os dados. Uma vez que o FPGA está programado, o sinal DONE é ativado, indicando ao processador o fim da programação.

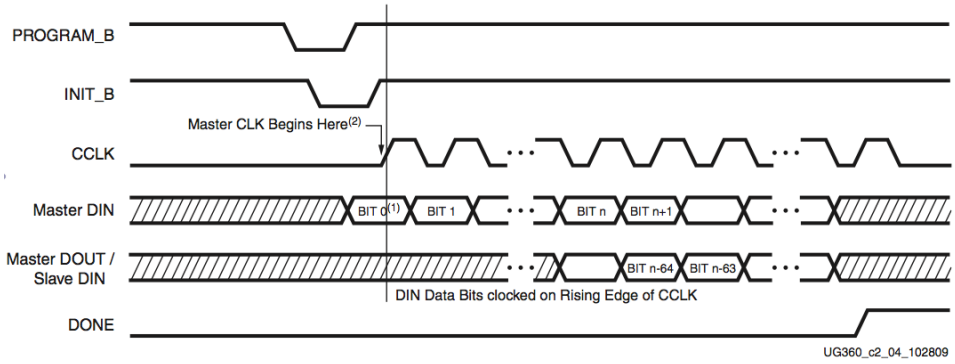

Figura 44: Sinais utilizados na programação do FPGA.

O código mostrado na listagem 7.2 mostra um conjunto de funções utilizadas na programação do FPGA. Estas funções foram criadas a nível do Kernel do Linux e algumas funções são exportadas ao espaço de usuário para que a programação possa

ser chamada de qualquer usuário do sistema. As funções de acesso aos pinos de GPIO do processador podem ser encontradas diretamente no Kernel do Linux no arquivo arch/powerpc/sysdev/gpio.c.

ANEXO A MODULO DE OPERAÇÕES EM PONTO FLUTUANTE

O módulo fpu_double (LUNDGREN, 2012) implementa o IEEE-754 (IEEE, 2012), que define o padrão para a aritmética em ponto flutuante. Este módulo implementa as 4 operações básicas que são adição, subtração, multiplicação e divisão. A Figura 45 ilustra a arquitetura implementada.

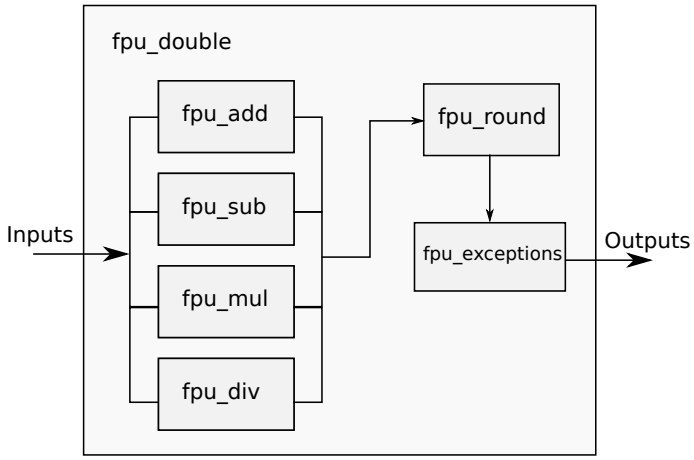

Figura 45: Componentes do módulo FPU.

A implementação do módulo define 7 sinais de entrada e 7 sinais de saída. A arquitetura topo do módulo pode ser vista na Figura 46.

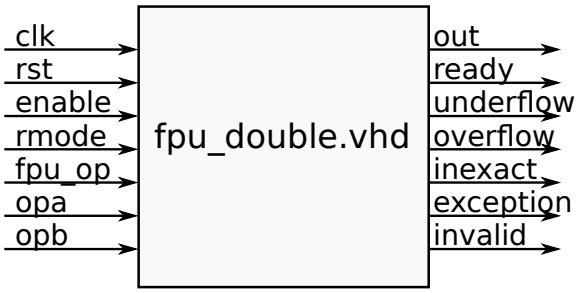

Figura 46: Arquitetura topo do módulo FPU.

As entradas do módulo são:

- clk: *Clock* do módulo. Todos os blocos internos sincronizam o clock com esta entrada.

- rst: *Reset* assíncrono. Sinal global que reinicia todos os registradores do módulo.
- enable: Habilita/desabilita o funcionamento do módulo.
- rmode: Modo de arredondamento, sendo que é possível selecionar um arredondamento para o número ímpar mais próximo, para zero, para cima ou para baixo.
- fpu_op: Modo de operação.
- opa: Valor de 64 bits do primeiro número.
- opb: Valor de 64 bits do segundo número.

As saídas do módulo são:

- out: Resultado da operação em 64 bits.
- ready: Sinal que indica que a operação já está pronta.
- underflow: Sinal que indica um *underflow* na operação desejada.
- overflow: Sinal que indica um *overflow* na operação desejada.
- inexact: Sinal que indica que o resultado da operação não é exato.
- exception: Sinal que indica que ocorreu alguma exceção durante o cálculo.
- invalid: Sinal que indica que o resultado é inválido.