Rogério Esteves Salustiano

NÚCLEO DE SIMULAÇÃO COMPUTACIONAL BASEADO NA SINCRONIZAÇÃO POR BARREIRAS COM APLICAÇÃO EM REDE DE SENSORES SEM FIO

Campinas
2014

UNICAMP

Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação

Rogério Esteves Salustiano

NÚCLEO DE SIMULAÇÃO COMPUTACIONAL BASEADO NA SINCRONIZAÇÃO POR BARREIRAS COM APLICAÇÃO EM REDE DE SENSORES SEM FIO

Tese de doutorado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos exigidos para a obtenção do título de Doutor em Engenharia Elétrica. Área de concentração: Eletrônica, Microeletrônica e Optoeletrônica.

Orientador: **Prof. Dr. Carlos Alberto dos Reis Filho**

Este exemplar corresponde à versão final da tese defendida pelo aluno **Rogério Esteves Salustiano** e orientada pelo **Prof. Dr. Carlos Alberto dos Reis Filho**

Campinas
2014

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Elizangela Aparecida dos Santos Souza - CRB 8/8098

Sa38n

Salustiano, Rogério Esteves, 1978-
Núcleo de simulação computacional baseado na sincronização por barreiras com aplicação em redes de sensores sem fio / Rogério Esteves Salustiano. – Campinas, SP : [s.n.], 2014.

Orientador: Carlos Alberto dos Reis Filho.
Tese (doutorado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Simulação por computador. 2. Redes de sensores sem fio. 3. Sincronização. 4. Framework (Programa de computador). I. Reis Filho, Carlos Alberto dos,1950-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Computer simulation core based on barrier synchronization with application in wireless sensor networks
Palavras-chave em inglês:
Computer simulation
Wireless sensor networks
Synchronization
Framework (Computer program)
Área de concentração: Eletrônica, Microeletrônica e Optoeletrônica
Titulação: Doutor em Engenharia Elétrica
Banca examinadora:
Carlos Alberto dos Reis Filho [Orientador]
Dilvan de Abreu Moreira
João Henrique Kleinschmidt
Edmundo Roberto Mauro Madeira
Nelson Luis Saldanha da Fonseca
Data de defesa: 27-06-2014
Programa de Pós-Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - TESE DE DOUTORADO

Candidato: Rogério Esteves Salustiano

Data da Defesa: 27 de junho de 2014

Título da Tese: "Núcleo de Simulação Computacional Baseado na Sincronização por Barreiras com Aplicação em Rede de Sensores sem fio"

Prof. Dr. Carlos Alberto dos Reis Filho (Presidente): _____

Prof. Dr. Dílvan de Abreu Moreira: _____

Prof. Dr. João Henrique Kleinschmidt: _____

Prof. Dr. Edmundo Roberto Mauro Madeira: _____

Prof. Dr. Nelson Luís Saldanha da Fonseca: _____

Resumo

O principal objetivo deste trabalho é a proposta e a implementação de um núcleo de simulação do tipo *event-driven* baseado em agentes destinado ao estudo e previsão de desempenho de Redes de Sensores Sem Fio. O núcleo de simulação, BaSS – *Barrier Synchronization Simulator*, foi desenvolvido a partir do modelo básico de sincronização por barreiras, cujas funcionalidades foram ampliadas com a inclusão de temporização e interrupção nos eventos de sincronização, sendo esta a contribuição mais relevante do trabalho. Sendo parte de um projeto com metas mais ambiciosas, em particular a busca de estruturas de Redes de Sensores Sem Fio mais eficientes, contando para isto com o envolvimento de múltiplas expertises, dentre as quais projeto de circuitos integrados, eficiência energética e protocolos de comunicação, o presente trabalho foi motivado pela necessidade de uma ferramenta de simulação suficientemente flexível para acomodar os dispositivos que estariam sendo desenvolvidos e usados nas novas Redes de Sensores Sem Fio. Como resultado, considera-se que os frutos tangíveis deste trabalho são: primeiro, um *framework* que pode ser utilizado no desenvolvimento de simuladores do tipo *event-driven* com aplicações diversas e, segundo, um simulador com aplicação específica em Redes de Sensores Sem Fio, que permite a modelagem comportamental dos seus elementos, tais como as propriedades dos sensores, as características de transmissão e recepção e as cargas das baterias. O simulador tem uma interface gráfica que permite a visualização dinâmica da rede e a inclusão de recursos de avaliação e controle da simulação.

Abstract

The main objective of this work is the proposal and the implementation of an event-driven simulation kernel based on agents for the study and performance prediction of Wireless Sensor Networks The simulation kernel, BaSS - Barrier Synchronization Simulator, was developed based on the barrier synchronization model, whose features were expanded with the inclusion of interruptions and timers in the synchronization of events, which is the most important contribution of this work. As a part of a project with more ambitious purposes, particularly pursuing the development of more efficient wireless sensor networks, counting for this with the involvement of diverse expertises, including design of integrated circuits, energy efficiency and communication protocols, the herein presented work was motivated by the need of a flexible simulation tool to accommodate the devices that would be developed and used in the new wireless sensor networks. As a result, it is understood that the tangible contributions of this work are: first, a framework that can be used in the development of any kind of event-driven simulators and, second, a simulator with a specific application in Wireless Sensor Networks, which allows the behavioural modelling of their elements, such as sensors proprieties, transmission and reception characteristics and batteries charges. The simulator has a graphical interface that allows the dynamic visualization of the network and the use of evaluation resources and controls of the simulation.

"A auto-satisfação é inimiga do estudo.
Se queremos realmente aprender alguma coisa,
devemos começar por libertar-nos disso.
Em relação a nós próprios devemos ser insaciáveis
na aprendizagem e em relação aos outros,
insaciáveis no ensino".
Mao Tse-Tung

“Não sei o que possa parecer aos olhos do mundo
mas aos meus pareço apenas ter sido como
um menino brincando à beira-mar, divertindo-me
com o fato de encontrar de vez em quando um seixo
mais liso ou uma concha mais bonita que o normal,
enquanto o grande oceano da verdade permanece
completamente por descobrir à minha frente”.
Isaac Newton

Aos meus pais,
Pedro Martins Salustiano e
Maria Albertina Esteves Salustiano,
sempre presentes e me apoiando.

Agradecimentos

Agradeço a Deus por ter me dado saúde, força, ânimo e potencial criativo para trilhar essa árdua caminhada de estudo e pesquisa.

Ao professor Reis pela confiança em me orientar mais uma vez e paciência diante a minha tendência em fazer diversas atividades ao mesmo tempo.

Aos colegas, funcionários e professores da FEEC e do Instituto de Computação (IC) pelo companheirismo, incentivo e transmissão de conhecimento. Agradeço ao colega de laboratório Felipe Moura Miranda pelo auxílio no teste do simulador e, em especial, a Vilson Rodrigo Mognon, pelos ensinamentos de eletrônica que engrandeceram minha formação.

As amigas da sala 13 da pós-graduação do Instituto de Geociências da UNICAMP pelas reflexões sobre astronomia, vida acadêmica e educação, temas sempre presentes nas nossas conversas. Em especial, Flávia Requeijo e Cissa Nascimento.

Aos amigos Rafael Cataldo e Flávia Callefo, pelo companheirismo em todos os momentos.

A UNICAMP pela sua estrutura física e organizacional. Nessa “Terra do Nunca” eu pude vivenciar uma realidade ímpar que contribuiu de forma inigualável para a minha formação profissional e pessoal. Hoje entendo o verdadeiro significado desse apelido.

Ao CNPq e a FAPESP pelo auxílio financeiro.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

A/D – Analógico/Digital
API – Application Programming Interface
ASIC – Aplication-Specific Integrated Circuit
BaSS – Barrier Synchronization Simulator
BS – Base Station
CCR – Corner Cubic Reflector
CDMA – Code Division Multiple Access
CPU – Central Processing Unit
CRC – Cyclic Redundancy Check
DSN – Distributed Sensor Networks
DTD – Document Type Definition
FDMA – Frequency Division Multiple Access
FIFO – First-In-First-Out
GPS – Global Positioning System
GUI – Graphical User Interface
H-WSN – Heterogeneous Wireless Sensor Network
I^2C – Inter-Integrated Circuit
IoT – Internet of Things
IP – Internet Protocol
JAR – Java Archive
JDK – Java Development Kit
JRE – Java Runtime Environment
JVM – Java Virtual Machine
MAC – Medium Access Control
MANET – Mobile Ad Hoc Network
NN – Network Node
OSI – Open Systems Interconnection
PN – Processing Node
RPC – Remote Procedure Call
RSSF – Redes de Sensores Sem Fio
RSSI – Received Signal Strength Indication
SN – Sensor Node
SPI – Serial Peripheral Interface
TDMA – Time Division Multiple Access
UML – Unified Modeling Language

VHDL – VHSIC Hardware Description Language
WLAN – Wireless Local Area Network
WMN – Wireless Multihop Networks
WSN – Wireless Sensor Network
WSN-BaSS – Wireless Sensor Network - Barrier Synchronization Simulator
XML – eXtended Markup Language

1
Introdução

"If we knew what it was we were doing,
it would not be called research, would it?".
Albert Eintein

O desenvolvimento tecnológico das últimas décadas promoveu um grande avanço na comunicação sem fio, no processamento de dados e na exatidão, precisão e diversidade de sensores. Componentes microeletrônicos com dimensões cada vez menores e de menor consumo energético permitiram a materialização das Redes de Sensores Sem Fio.

As pesquisas acadêmicas e aplicações comerciais voltadas para esse tipo de rede vêm crescendo de forma relevante, principalmente diante os desafios e as oportunidades na sua utilização.

Um dos principais desafios das Redes de Sensores Sem Fio é garantir sua eficiência e o seu tempo de vida diante as limitações energéticas dos nós sensores. O desenvolvimento de dispositivos de baixo consumo (*ultra low-power*) e a concepção de estratégias de comunicação e processamento eficientes são fundamentais para aumentar o tempo de operação dessas redes, viabilizando, assim, a sua utilização.

Dentre as inúmeras aplicações que irão se beneficiar das Redes de Sensores Sem Fio de longa duração, pode-se citar o monitoramento ambiental, a agricultura de precisão, o monitoramento da saúde e de esportes, aplicações militares e a manutenção de infraestruturas

civis.

O estudo do comportamento e da eficiência energética das Redes de Sensores Sem Fio não é uma tarefa simples de ser realizada diante a sua escalabilidade e topologia diversificada. Dessa forma, é de suma importância a utilização de ferramentas computacionais de simulação para a previsão do desempenho dessas redes em diversas condições de operação.

Este trabalho de tese foi motivado pela necessidade de uma ferramenta de simulação de Redes de Sensores Sem Fio que permitisse a incorporação de dispositivos eletrônicos ainda em fase de desenvolvimento, cuja variabilidade requer uma grande flexibilidade descritiva.

Apesar dos esforços realizados pela comunidade científica no estudo das Redes de Sensores Sem Fio utilizando simuladores, a proposta desta tese se justifica, a partir do conhecimento do autor da literatura disponível, pela ausência de uma ferramenta de simulação voltada para Redes de Sensores Sem Fio que seja flexível e abrangente o suficiente para permitir o estudo adequado, sistemático e comparativo, dessas redes.

Os objetivos desta tese, portanto, envolvem prover um conjunto de soluções para problemas relacionados à simulação de Redes de Sensores Sem Fio, considerando os seguintes aspectos:

- Controle autônomo da sincronização dos objetos simulados e dos mecanismos que determinam o tempo de vida da rede;
- Possibilidade de representação do comportamento do hardware e do software envolvidos na dinâmica operacional dos nós, rádios, sensores e baterias;
- Flexibilidade nos mecanismos de interação entre os objetos simulados, permitindo uma maior fidelidade representativa;
- Possibilidade de representação de variáveis ambientais dinâmicas;
- Facilidade de incorporação de ferramentas analíticas e produtoras de relatórios junto ao simulador;
- Modularidade na descrição dos objetos e de seus comportamentos, com o objetivo de organizar e reaproveitar a programação;
- Facilidade de criação de redes com topologias diferentes para a realização de testes de desempenho criados a partir de combinações de protocolos de comunicação e componentes de hardware distintos.
- Possibilidade de acompanhamento da simulação através de um ambiente gráfico enquanto a mesma está em execução.

Para alcançar esses objetivos, foi desenvolvido um *framework* de simulação de propósito geral denominado BaSS (*Barrier Synchronization Simulator*) e, a partir dele, um simulador dedicado à Redes de Sensores Sem Fio, o WSN-BaSS.

As contribuições desta tese são as seguintes:

- A caracterização do cenário atual de estudo, desenvolvimento e aplicação das Redes de Sensores Sem Fio através de uma visão crítica da literatura especializada;
- Uma análise dos simuladores utilizados nos estudos das Redes de Sensores Sem Fio e as limitações associadas aos seus usos;
- Proposta de um conjunto de aprimoramentos no modelo de sincronização por barreiras (prioridades, interrupções e temporizadores), ampliando, assim, a possibilidade de controle e interação entre os objetos de uma simulação;
- Implementação de um núcleo de simulação do tipo event-driven baseado em agentes através de um framework na linguagem de programação Java, tendo com base o modelo de sincronização por barreiras aprimorado;
- Proposta e implementação de um simulador para Rede de Sensores Sem Fio desenvolvido na linguagem de programação Java.

O texto desta tese está organizado em seis capítulos, conforme os conteúdos indicados a seguir.

O capítulo 2 é essencialmente teórico, apresentando as definições de Redes de Sensores Sem Fio encontradas na literatura, as possíveis classificações e as arquiteturas de hardware (projetos dos nós sensores) e software (sistemas operacionais e protocolos) comumente utilizados nesse tipo de rede. Além disso, o capítulo 2 descreve o ciclo de vida de uma Rede de Sensores Sem Fio, o consumo de energia dos nós sensores e apresenta 17 critérios diferentes para a definição do tempo de vida da rede. Aplicações e desafios no desenvolvimento das Redes de Sensores Sem Fio finalizam o capítulo.

O capítulo 3 relaciona as 16 principais ferramentas de simulação utilizadas no estudo das Redes de Sensores Sem Fio, apresentando uma breve descrição das suas funcionalidades e as suas limitações. Baseado nas implementações desses simuladores e nos conceitos explorados no capítulo 2, o capítulo 3 ainda apresenta uma lista das principais características desejáveis em um simulador de Rede de Sensores Sem Fio.

No capítulo 4 é proposto um núcleo de simulação de propósito geral e são explicados os aprimoramentos realizados no modelo de sincronização por barreiras. O *framework* resultante da implementação desse núcleo na linguagem de programação Java, o

BaSS (*Barrier Synchronization Simulator*), é apresentado de forma detalhada, assim como os testes realizados e as análises de desempenho.

O capítulo 5 apresenta o WSN-BaSS, simulador para Redes de Sensores Sem Fio desenvolvido na linguagem de programação Java. Baseado na arquitetura de sincronização do BaSS, a concepção e implementação do WSN-BaSS são explicadas em detalhes.

O capítulo 6 finaliza o texto da tese com as conclusões obtidas do estudo das Redes de Sensores sem Fio e do desenvolvimento do simulador apresentado. Possíveis extensões deste trabalho são sugeridas.

2
Rede de Sensores Sem Fio

"O mestre disse a um dos seus alunos: Yu, queres saber em que consiste o conhecimento? Consiste em ter consciência tanto de conhecer uma coisa quanto de não a conhecer. Este é o conhecimento..."
Confúcio

A pesquisa em rede de sensores foi considerada uma das seis áreas de grande desafio em pesquisa de acordo com o relatório do *Workshop* em Estudos Fundamentais de Redes de 2003, realizado na Universidade da Virginia (EUA) e patrocinado pela *National Science Foundation*[1] [AMM03]. Redes de comunicação sem fio também foi um tema citado nesse relatório como de grande relevância para a solução de problemas do mundo real.

Desde então, as Redes de Sensores Sem Fio, cujas primeiras pesquisas já estavam em andamento desde a década de 70, passaram a despertar um grande interesse tanto nas pesquisas acadêmicas como em aplicações comerciais.

Com o desenvolvimento e a convergência das tecnologias de circuitos digitais, comunicação sem fio e MEMS (*Microelectromechanical Systems*) nas últimas décadas, as Redes de Sensores Sem Fio puderam ser materializadas, alguns problemas inerentes da sua estrutura foram tratados e novos desafios emergiram. Alguns desses desafios continuam até o presente, promovendo oportunidades de investigação tanto em *hardware* como em *software*

[1] *National Science Foundation* é a agência do governo dos EUA responsável por promover programas e projetos de educação em Ciência e Engenharia.

embarcado e protocolos de comunicação.

Este capítulo, essencialmente descritivo, faz uma revisão bibliográfica das Redes de Sensores Sem Fio, iniciando pela sua definição e características, aplicações e possíveis classificações. Em seguida é feita uma coletânea dos principais elementos de hardware e software presentes nesse tipo de rede, dando ênfase aos protocolos de comunicação.

O ciclo de vida das WSN, o tempo de vida da rede e o gasto de energia dos nós sensores são os temas centrais deste capítulo.

As aplicações das Redes de Sensores Sem Fio e os principais desafios no desenvolvimento das WSN são discutidos nos últimos itens.

2.1 – Definição e características

As Redes de Sensores Sem Fio (WSN – *Wireless Sensor Networks*)[2] podem ser definidas de maneira geral da seguinte maneira [HAE04][AKA05][DAI05]:

> *Uma Rede de Sensores Sem Fio consiste em um conjunto de nós sensores espalhados em uma área geográfica de interesse que colaboram mutuamente na execução de tarefas e utilizam para isso uma rede de comunicação sem fio.*

Contudo, a literatura especializada em WSN apresenta outras definições similares para o termo no que tange aos elementos de rede envolvidos, mas que divergem em relação à natureza e configuração da rede. Cada autor estende a definição geral apresentada acordo com o foco do seu trabalho. A descrição de Redes de Sensores Sem Fio realizada a seguir é um apanhado de características encontradas na literatura que, juntas, permitem compreender os aspectos organizacionais e comportamentais desse tipo de rede.

Os elementos de rede comuns em todas as Redes de Sensores Sem Fio são: **nós sensores** (SN – *Sensor Nodes*) e **estações rádio base** (BS – *Base Stations*). Tanto os nós sensores como as estações rádio base são **nós de rede** (NN – *Network Nodes*). Os nós sensores também são denominados *motes*, termo utilizado para designar pequenos dispositivos eletrônicos dotados de poucos recursos [RAJ03a]. Outros termos comumente utilizados para designar uma estação rádio base são: nó sorvedouro (*sink node*), nó de processamento (PN –

[2] No decorrer do texto o acrônimo inglês WSN será utilizado para referenciar Redes de Sensores Sem Fio, ao invés do acrônimo português RSSF, encontrado em alguns artigos e livros publicados na língua portuguesa. Essa decisão é decorrente da já constatada não popularização na utilização de outros acrônimos "traduzidos" para o português, tais como SIDA (Síndrome da Imunodeficiência Adquirida) para AIDS (*Acquired Immunodeficiency Syndrome*) e ADN (Ácido Desoxirribonucleico) para DNA (*Deoxyribonucleic Acid*), uma vez que possuem utilização limitada ao país da língua, não sendo de grande difusão como na língua inglesa, de abrangência mundial.

Processing Node) [CHA08] e nós de monitoração [SIL04].

Os nós sensores de uma WSN dão dotados de um ou mais sensores de naturezas diversas (temperatura, umidade, pressão, luminosidade, etc.) e são limitados tanto em tamanho como em capacidade de processamento, armazenamento de dados, disponibilidade de energia (baterias pequenas e de baixa capacidade) e potência de transmissão do rádio (dezenas a centenas de metros). As estações rádio base geralmente dispõem de maior quantidade de recursos e fonte de energia, utilizando baterias de maior capacidade ou até mesmo valendo-se da rede elétrica convencional, dependendo da disponibilidade na sua localização.

Apesar das Redes de Sensores Sem Fio serem redes compostas por dispositivos com capacidades computacionais, elas são consideravelmente distintas das redes tradicionais. Uma WSN é constituída de uma ou mais estações rádio base e de dezenas a milhares de nós sensores. O papel de um nó sensor na rede é trabalhar em conjunto com os outros nós sensores para transmitir os dados dos seus sensores até uma estação rádio base. As estações rádio base, por sua vez, têm como principal função receber os dados dos nós sensores e disponibilizá-los para os usuários através da conexão com computadores ou realizando a interligação da WSN com outras redes (Ethernet, 3G, Wi-Fi, etc.).

A Figura 2.1 ilustra uma WSN com onze nós de rede, sendo dez nós sensores e uma estação rádio base. A transmissão dos pacotes de dados ocorre via comunicação sem fio entre os nós da rede (nós sensores e estação rádio base) e, neste exemplo, a estação rádio base (BS) encaminha os pacotes da WSN para uma rede IEEE 802.3.

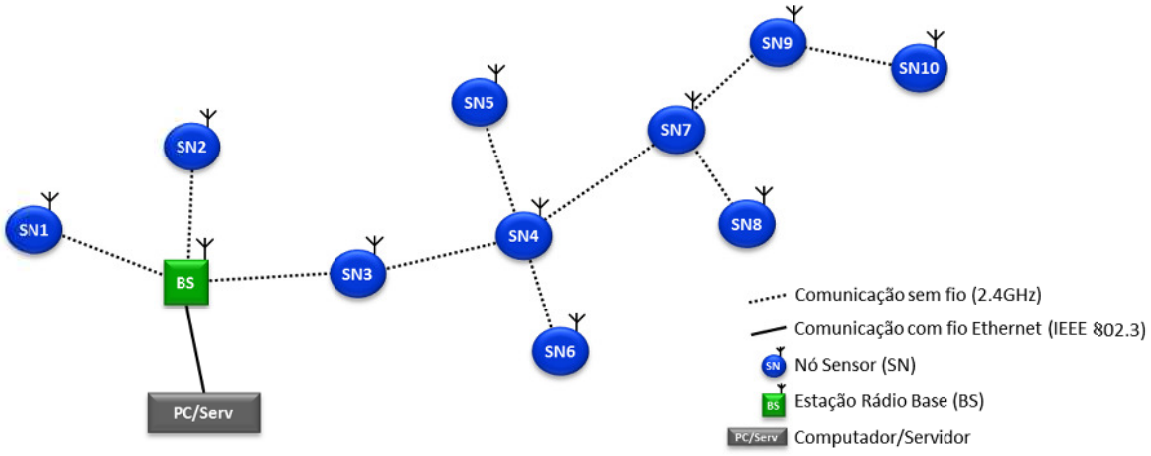


Figura 2.1 – Exemplo de uma Rede de Sensores Sem Fio: dez Nós Sensores (SN) transmitem seus dados via comunicação sem fio para uma Estação Rádio Base (BS); a BS envia os dados por uma rede Ethernet com fio para um Computador/Servidor, que disponibiliza os dados para o usuário.

Em algumas aplicações, as estações rádio base podem estar localizadas longe da infraestrutura de outras redes. Nesses casos, as estações rádio base são responsáveis por

coletar e armazenar os dados provenientes dos nós sensores e, posteriormente, os dados podem ser transferidos para outros equipamentos.

A transmissão dos dados dos nós sensores até uma estação rádio base raramente ocorre com uma única transmissão (*one-hop*). Devido à potência de transmissão do rádio dos nós sensores ser baixa, seu alcance é limitado, fazendo com que as transmissões da maioria dos nós sensores da rede não seja recebida por nenhuma estação rádio base diretamente. Para resolver este problema, a maioria dos protocolos de comunicação[3] desenvolvidos para WSN utilizam o esquema *multihop* (saltos múltiplos), no qual a transmissão de pacotes do nó origem ao nó destino utiliza nós intermediários como retransmissores, conforme ilustrado na Figura 2.2.

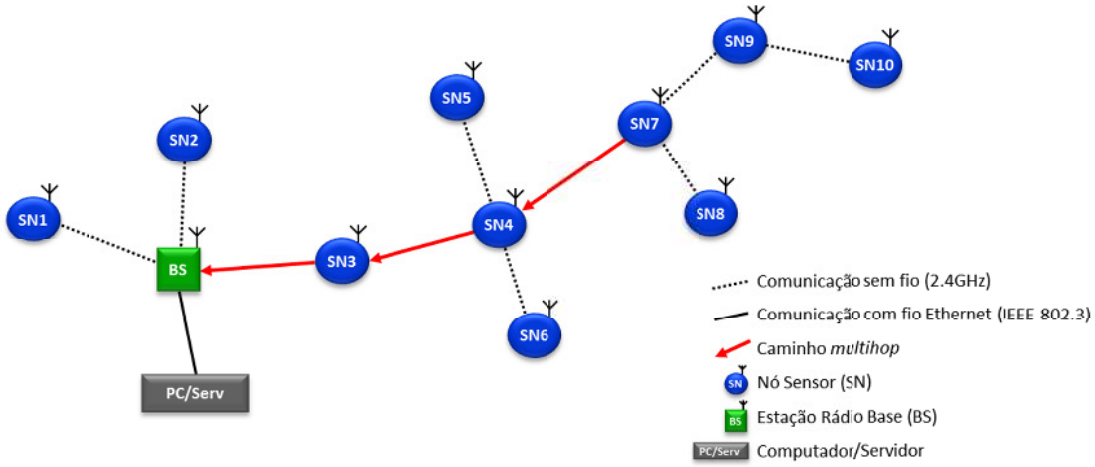


Figura 2.2 – Transmissão de pacote utilizando *multihop*. A transmissão de um pacote partindo do Nó Sensor SN7 para Estação Rádio Base (BS) não é possível ser realizada diretamente, pois a potência de transmissão do rádio de SN7 não é suficiente para sensibilizar o receptor da BS. Utilizando o esquema *multihop*, o pacote de dados originado em SN7 passa sequencialmente pelos nós intermediários SN4 e SN3 até chegar à BS.

Como a maioria dos protocolos de comunicação desenvolvidos para WSN são *multihop*, [SHA04] as denomina como Redes *Multihop* Sem Fio (WMN – *Wireless Multihop Networks*).

A necessidade de trabalho em conjunto (sinergia) dos nós sensores faz com que uma WSN seja mais parecida com um Sistema Distribuído do que com uma rede tradicional, uma vez que nas redes tradicionais os nós interagem no sentido de requisitar serviços oferecidos por outros nós e não no sentido colaborativo de execução de tarefas. De fato, as Redes de Sensores Sem Fio foram classificadas como Sistemas Distribuídos por [BOR04], ou

[3] Protocolo de comunicação é o conjunto de mensagens de controle e pacotes de dados que são trocados entre as unidades de comunicação, além das regras que definem o momento em que os dados devem ser transmitidos ou recebidos [SIL00].

como é definido em [ELS04], uma nova classe de Sistemas Distribuídos. [YUA04] utiliza a denominação Redes Distribuídas de Sensores (DSN – *Distributed Sensor Networks*) para designar as Redes de Sensores Sem Fio.

Uma característica importante que permite classificar as WSN como Sistemas Distribuídos é o fato dos usuários raramente estarem interessados nos dados de um único nó, mas requisitarem informações de um determinado fenômeno físico medido pela rede, que pode incluir a combinação dos dados de diversos nós sensores [BOU04].

Auto-organização é outra característica que deve ser suportada pelas Redes de Sensores Sem Fio, isto é, os nós devem cooperar entre si para executarem uma determinada funcionalidade sem a interferência do ser humano [PAP04]. Neste sentido, a auto-organização (e a automanutenção [ALK04]) envolve desde o estabelecimento de rotas para os pacotes, a detecção de falhas, o gerenciamento de energia, a combinação de dados, etc.

A identificação dos nós sensores na rede através de um número (ID) ou endereço de rede (endereço IP, por exemplo) é algo não factível em Rede de Sensores Sem Fio devido à quantidade de nós sensores. E ainda, como é ressaltado em [ALK04], na maior parte das aplicações de WSN o dado em si é mais importante do que se saber qual nó o transmitiu. Apesar disso, [HE_03] e [LAZ05] destacam que a localização do dado é uma informação importante, mas esta localização não é necessariamente a coordenada geográfica do nó, podendo ser mais adequado delimitar regiões espaciais a partir das quais os dados são produzidos.

Em comparação com as Redes *Ad Hoc*[4] Sem Fio, as Redes de Sensores Sem Fio se diferenciam principalmente devido à (1) distribuição densa dos nós, (2) alteração na topologia devido à possibilidade de falhas nos nós e (3) limitação nos recursos dos nós sensores (energia, potência de transmissão, memória e processamento). Além dessas diferenças, redes *ad hoc* tipicamente apresentam nós com características heterogêneas, enquanto que WSN possuem nós sensores do mesmo tipo; redes *ad* hoc são móveis e WSN apresentam pouca mobilidade (são praticamente estacionárias na maioria das aplicações); Redes de Sensores Sem Fio sempre possuem as estações rádio base como destino final dos pacotes de dados, enquanto que redes *ad hoc* não possuem nós concentradores análogos [COL03][SAN05][TUB03].

A Tabela 2.1 apresenta um resumo comparativo, segundo [WAN04], entre as Redes

[4] *Ad Hoc* é uma expressão em latim que significa “feita para este propósito”. A utilização deste termo em redes refere-se a não necessidade de uma infraestrutura (roteadores, pontos de acesso sem fio, etc.) para gerenciar o encaminhamento dos pacotes transmitidos. Tal gerenciamento é realizado de modo dinâmico, de acordo com a conectividade existente na rede.

de Sensores Sem Fio e as Redes *Ad Hoc* Sem Fio. Deve-se lembrar, contudo, que alguns projetos de WSN não utilizam redes e/ou protocolos que admitam todas as características identificadas nessa tabela e, além disso, essa diferenciação não é observada por outros autores, que classificam as WSN como um tipo particular de Rede *Ad Doc* Sem Fio [BOK04][XIA07][VU_09]. A classificação de WSN como sendo uma MANET (*Mobile Ad Hoc Network*), por exemplo, feita em [RUI04b] e [YI_03], atribui às Redes de Sensores Sem Fio a característica de serem redes móveis.

Tabela 2.1 – Comparação entre WSN e Redes *Ad hoc* Sem Fio [WAN04]

Característica	WSN	*Ad hoc* sem fio
Número de nós	Grande (centenas a milhares)	Pequeno a médio
Densidade dos nós	Alta	Relativamente baixa
Redundância de dados	Alta	Baixa
Suprimento de energia	Baterias insubstituíveis	Baterias recarregáveis
Taxa de dados	Baixas (1 a 100 Kps)	Alta
Mobilidade dos nós	Baixa	Pode ser muito alta
Direção de fluxos	Predominantemente unidirecional	Bidirecional
Encaminhamento de pacotes	Muitos para um (centrado em dados)	Fim-a-fim (centrado em endereços)
Natureza de consultas	Baseada em atributos	Baseada em nós
Disseminação de consultas	*Broadcast*	Nó a nó ou *broadcast*
Endereçamento	Ausência de ID global único	ID global único
Ciclo de atividade (*duty cycle*)	Pode ser tão baixo quanto 1%	Alto

Pelo fato dos nós sensores na maioria das WSN possuírem características homogêneas e apresentarem-se distribuídos densamente [BRO06], há uma grande redundância dos dados transmitidos a partir dos nós pertencentes a uma área específica da rede. Apesar dessa característica ser ruim sob a perspectiva de desperdício de recursos de rede (uma vez que os nós sensores produzem praticamente os mesmos dados), ela é importante na medida em que garante a qualidade dos dados, pois a redundância pode ser utilizada para confirmação dos valores das grandezas medidas. A alta densidade de nós também permite que um maior número de rotas possa ser estabelecido entre os nós sensores, permitindo um escalonamento

na participação dos nós nas retransmissões *multihop*, e, por consequência, economizando a energia dos nós sensores. Além disso, a alta densidade de nós sensores garante uma maior tolerância a falhas, pois múltiplas rotas aumentam a conectividade da rede [ALK04].

A característica de homogeneidade dos nós sensores presentes em uma WSN também não é consenso entre os autores. [BOU09] define as Redes de Sensores Sem Fio Heterogêneas (H-WSN - *Heterogeneous Wireless Sensor Network*), justificando a sua heterogeneidade pelo fato dela melhorar a escalabilidade da rede, permitir um balanceamento mais adequado na distribuição energética entre os nós (os nós podem possuir baterias com carga inicial distintas, de acordo com as sua função na rede, por exemplo), possuir nós com funcionalidades específicas na rede, permitir a incorporação de novas aplicações com a adição de novos nós à rede já estabelecidas, entre outros benefícios.

A taxa de transferência de dados (*throughput)* das WSN é bem menor se comparada a das redes do tipo Bluetooth (IEEE 802.15.1) e WLAN (IEEE 802.11). Comumente as taxas de transferência das Redes de Sensores Sem Fio são da ordem de 1 bit/s ou mais baixas [CAL03].

Grande parte dos artigos publicados em Redes de Sensores Sem Fio, dentre eles [SIL00], [RUI04a], [ALK04] e [SLI04], considera que no projeto de rede, tanto os elementos de *hardware* como os de *software* embarcado e protocolos de comunicação são extremamente dependes da aplicação. Contudo, há autores que contrariam a definição de que as WSN são desenvolvidas visando aplicações específicas, pelo menos no que tange ao *software* embarcado. Os projetos de Sistemas Operacionais para Redes de Sensores Sem Fio [DUL02][HUA07] indicam essa tendência de troca de aplicação dos nós sensores não somente antes das redes terem sido inseridas no ambiente, mas também na substituição do software embarcado dos nós sensores quando a rede já se encontra em operação (reprogramação *on the fly*) [HAN05]. Apesar disso, [BOK04] defende que na maioria das aplicações, uma vez que os nós foram distribuídos no ambiente, a aplicação da WSN é imutável.

Nas aplicações em que os nós sensores são distribuídos em ambientes de difícil acesso ou onde a localização exata dos nós não é sabida, a substituição ou o recarregamento das baterias torna-se uma tarefa muito difícil, senão impossível [WAN04][ZAY07][KRO08]. O fato da energia no nó sensor ser não renovável pode ser utilizado como um argumento a favor da ideia de que as WSN são desenvolvidas com foco na sua aplicação, uma vez que a seleção da bateria para o nó sensor (carga inicial) seria feita de acordo com a aplicação, a necessidade energética do protocolo de comunicação e o tempo de operação esperado para a rede.

A organização dos nós, isto é, como e quando os nós deverão se comunicar também pode ser considerado um quesito que depende da aplicação [PAP04]. O fato das WSN

serem auto-organizáveis não as tornam adaptáveis a qualquer tipo de aplicação, mas sim à relação estabelecida entre os nós de acordo com o posicionamento relativo desses no ambiente e proximidade com as estações rádio base. A auto-organização também pode ser vista como um atributo específico da aplicação.

O tráfego de dados em uma Rede de Sensores Sem Fio é baixo, sendo esta uma característica associada à própria natureza da rede e à necessidade de minimizar o gasto energético, deveras relacionado ao custo de transmissão de pacotes. Apesar disso, em aplicações que o disparo no envio de pacotes é determinado pela detecção de eventos (valores registrados pelos sensores fora de uma faixa determinada), há momentos de picos de transmissão, caracterizando essas WSN como tendo baixo ciclo de atividade, e, por consequência, alta latência na transmissão dos dados [ZAY07].

A segurança dos dados é um fator importante em algumas WSN, principalmente nas voltadas a aplicações militares [SLI04]. Contudo, grande parte dos protocolos de comunicação desenvolvidos para Redes de Sensores Sem Fio não leva em consideração a criptografia dos dados, uma vez que ela requer uma alta capacidade de processamento dos nós sensores, aumentado, assim, o consumo de energia por mensagem enviada.

A partir do exposto na literatura e sintetizado nas linhas antecedentes, com pertinência ao contexto desta tese, pode-se definir Redes de Sensores Sem Fio a partir das características relacionadas aos nós sensores, estações rádio base e estrutura da rede, apresentados na Figura 2.3.

2.2 – Classificações das Redes de Sensores Sem Fio

O fato das Redes de Sensores Sem Fio serem dependentes da aplicação não permite uma caracterização geral dessas redes que englobe todas as suas propriedades e funcionalidades. A caracterização apresentada na Figura 2.3 é superficial, sendo insuficiente para precisar uma determinada WSN. Contudo, é possível classificar as Redes de Sensores Sem Fio de acordo com alguns critérios gerais de redes e outros específicos para WSN, delimitando, assim, suas particularidades de forma mais rigorosa.

A lista mais completa encontrada na literatura com critérios para classificação de WSN está em [RUI03], trabalho no qual são consideradas quatro categorias gerais de classificação: (1) configuração dos nós e da rede, (2) periodicidade no sensoriamento, (3) tipo de comunicação e (4) tipo de processamento. Segue a classificação apresentada em [RUI03] acrescida de alguns itens encontrados em outros trabalhos que se encaixam adequadamente nas quatro categorias enunciadas, além de comentários pertinentes a cada classificação.

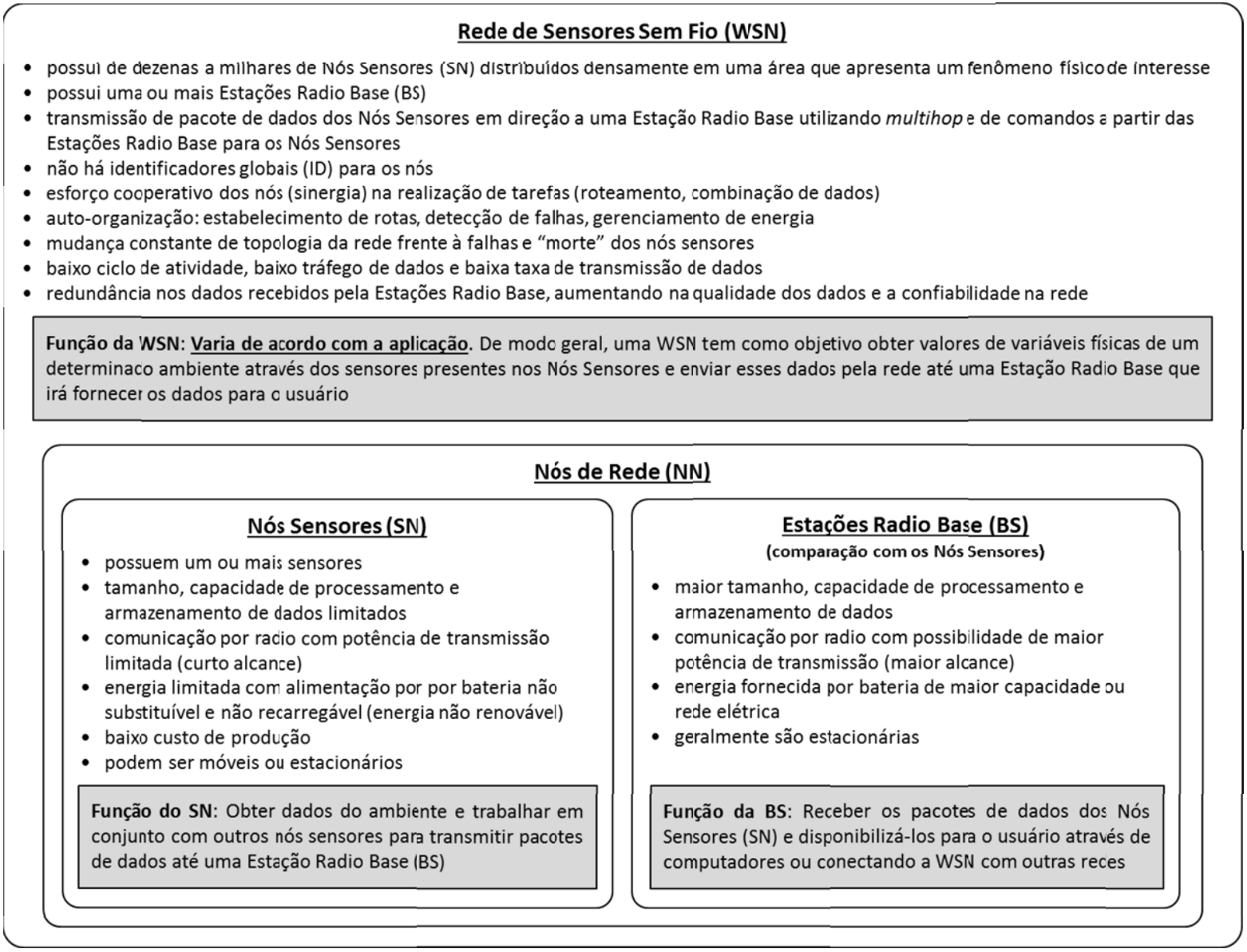


Figura 2.3 – Caracterização das Redes de Sensores Sem Fio.

A **configuração** de uma WSN pode ser classificada tanto do ponto de vista da natureza dos nós sensores (composição e mobilidade), como do ponto de vista da rede (organização, densidade e distribuição):

Composição: As redes podem ser **homogêneas** ou **heterogêneas**. No primeiro caso, todos os nós apresentam a mesma configuração de *hardware*, podendo, contudo, executar *softwares* diferentes; no segundo, os *hardwares* dos nós de rede são distintos (H-WSN na denominação de [BOU09]).

Mobilidade: Uma WSN pode ser **estacionária**, isto é, os nós sensores não mudam a sua localização durante o tempo de operação da rede, permanecendo fixos no local onde foram implantados. Nas Redes de Sensores Sem Fio **móveis** os nós se deslocam durante o tempo de operação da rede (MANET na denominação de [RUI04b] e [YI_03]).

Organização: A organização **hierárquica** prevê a divisão da rede em grupos (*clusters*) e a eleição de um dos nós membros de cada grupo como líderes (*cluster head*), podendo haver hierarquias inferiores dentro de cada grupo. Na organização **plana** todos os nós possuem a mesma função na rede e não há organização de grupos.

Densidade: Contrariando a definição de grande parte dos autores que as Redes de Sensores Sem Fio são densas por natureza, [RUI03] considera que as WSN podem apresentar distribuição **balanceada** (concentração e distribuição dos nós por unidade de área ideal segundo o objetivo funcional da rede), **densa** (alta concentração de nós por unidade de área) e **esparsa** (baixa concentração de nós por unidade de área).

Distribuição: As WSN podem apresentar distribuição **regular** (distribuição uniforme) ou **irregular** (poucos nós em determinadas áreas e muitos nós em outras). A classificação de distribuição dos nós sensores de forma irregular pode ser evidenciada em redes cuja implantação dos nós no meio é realizada de forma aleatória, podendo existir na área de interesse regiões com alta densidade de nós e outras com baixa densidade.

Com relação à **periodicidade de sensoriamento**, isto é, a decisão de quando os dados devem ser lidos dos sensores, uma WSN pode ser classificada de três formas:

Periódica: A coleta de dados é realizada em intervalos regulares pré-determinados (***Periodic Data Generation***).

Reativa: Os dados são coletados quando há a ocorrência de eventos de interesse

ou há uma solicitação da rede. As WSN reativas também são denominadas como ***event-driven*** [DEM06].

Contínua: Nesse tipo de coleta de dados, os nós (também conhecidos como nós com coleta de dados em **Tempo real**), obtém a maior quantidade de dados possível no menor intervalo de tempo que o seu *hardware* consegue realizar.

Considerando o **tipo de comunicação** (estratégia do protocolo e meio) estabelecido entre os nós, as Redes de Sensores Sem Fio podem ser classificadas segundo a disseminação dos dados, o tipo de conexão, a alocação do canal e o fluxo de dados, como segue.

Disseminação: A disseminação dos dados pode ser **programada** (em determinado horário ou em intervalos estabelecidos), **contínua** (os dados são enviados continuamente, assim que são adquiridos pelo nó sensor), **sob demanda** (os dados são disseminados quando há alguma solicitação da rede) ou **sob eventos** (os dados são transmitidos quando há a ocorrência de eventos de interesse, isto é, os valores lidos pelos sensores estão dentro de faixas de interesse pré-estabelecidas).

Tipo de Conexão: No tipo **simétrico**, todas as conexões, com exceção as das estações rádio base, possuem o mesmo alcance; nas conexões **assimétricas** os alcances dos rádios de cada nó são distintos. Esta classificação pode estar relacionada ao tipo de composição da rede, uma vez que redes homogêneas sempre teriam caráter simétrico. Contudo, a simetria pode ser algo questionável, uma vez que, mesmo que todos os nós sensores possuam o mesmo tipo de rádio e estes transmitam com a mesma potência e possuam a mesma sensibilidade de recepção, a presença de obstáculos e interferências localizadas alteraria o alcance dos rádios. O termo “alcance”, utilizado por [RUI03], torna a classificação das WSN por tipo de conexão algo teórico, que deve ser utilizado com cuidado ao empregá-lo em redes reais. Poder-se-ia empregar o termo “conectividade[5]” ao invés de “tipo de conexão”, utilizando a classificação **constante** para redes nas quais os nós sensores mantêm fixas a potência de transmissão e a sensibilidade de recepção e **variável** para redes cujos nós sensores alteram a potência de transmissão e a sensibilidade de recepção ao longo do tempo.

Alocação de Canal: A largura de banda pode ser **estática** ou **dinâmica**. No

[5] O termo conectividade está relacionado à capacidade ou possibilidade de um dispositivo em operar em ambiente de rede; já o termo conexão refere-se à ligação estabelecida entre dispositivos com objetivo de transferir dados (*Fonte*: Dicionário Houaiss da Língua Portuguesa). Na classificação apresentada como “tipo de conexão”, a palavra “alcance” foi utilizada corretamente, relacionando-se à “conexão”, mas ao se utilizar “potência de transmissão e sensibilidade de recepção”, o termo “conectividade” seria o mais apropriado, pois indica uma possibilidade de se estabelecer comunicação.

primeiro caso, a banda disponível para transmissão é dividida em faixas iguais, tantas quantas forem a quantidade de nós presentes na rede. Essa divisão pode ser feita em frequência (FDMA - *Frequency Division Multiple Access*), no tempo (TDMA - *Time Division Multiple Access*) ou por código (CDMA - Code *Division Multiple Access*). No caso da banda dinâmica, há disputa do canal entre os nós da rede, gerando conflitos. A referência [HAY08] pode ser utilizada para consulta mais detalhada sobre os protocolos de alocação de canal.

Fluxo de dados: Utilizando a política ***unicast***, os dados dos nós sensores são enviados diretamente para a estação rádio base sem a participação de nós intermediários (esquema de transmissão *one-hop*); no fluxo de dados por ***multicast***, somente os nós de um determinado grupo (*cluster*) devem receber os dados; no fluxo ***flooding***, todos os vizinhos de um nó sensor recebem os dados e os retransmitem para os seus vizinhos, de forma recursiva; no fluxo por ***gossiping*** somente alguns nós escolhidos pelo transmissor devem receber as mensagens; e no fluxo ***bargaining*** somente um nó transmite se houver requisição dos seus dados por outro nó (requer um processo de negociação). O fluxo de dados está diretamente relacionado com o protocolo de comunicação e muitas vezes este é desenvolvido utilizando mais de um tipo de fluxo de dados.

Com relação ao **processamento de dados**, os nós sensores das Redes de Sensores Sem Fio podem executar processamentos localizados e/ou distribuídos:

Localizada: O processamento dos dados é realizado localmente, sem a interação com os dados de outros nós. Os pacotes de dados recebidos de outros nós sensores são apenas retransmitidos, sem modificação ou aproveitamento dos dados recebidos.

Infraestrutura: As informações são processadas de forma a permitir o controle de acesso ao meio, estabelecer rotas, realizar a eleição de líderes, descobrir a localização dos nós, etc. Este tipo de processamento está relacionado com a organização dos nós sensores em rede.

Correlação: Nesse tipo de processamento os nós trabalham de forma conjunta na execução de tarefas para melhoria na qualidade dos dados ou otimização no pacote de dados. Os procedimentos de correlação mais comuns são: a fusão de dados, contagem, filtragem, compressão e agregação.

Além dos critérios classificativos apresentados, outros dois podem ser adicionados à lista, apesar de não se enquadrarem nas quatro categorias definidas em [RUI03]. Estes dois critérios adicionais levam em consideração a aplicação da Rede de Sensores Sem Fio.

O primeiro critério adicional está ligado a questões de **operação** da rede, podendo uma WSN ser orientada a **consulta** (*Querying Applicatons*) ou a **tarefas** (*Tasking Applications*) [SHE04]. Esse critério é de suma importância, pois tem grande impacto no projeto de protocolos de comunicação para WSN, na medida em que define o sentido do fluxo dos pacotes na rede (não apenas o fluxo dos dados pelo nó, como é classificado em "fluxo de dados"). Em redes orientadas a consulta, o fluxo de pacotes é bidirecional: as estações rádio base enviam comandos de consulta para os nós sensores e estes enviam seus dados (o resultado da consulta) para uma estação rádio base. Em redes orientadas a tarefas, o fluxo de pacotes é unidirecional, partido dos nós sensores em direção às estações rádio base.

O outro critério adicional refere-se ao modo de **implantação** dos nós sensores na área de interesse. A inserção dos nós no ambiente pode ser realizada de forma **aleatória**, na qual os nós sensores são "jogados" no ambiente sem determinação à priori do local específico em que eles irão realizar o sensoriamento, ou de forma **planejada**, posicionando os nós em locais pré-determinados [ALK04]. Esse critério também é extremamente relevante no projeto de uma WSN, uma vez que os protocolos de auto-organização da rede cujos nós sensores são distribuídos aleatoriamente são mais complexos de serem concebidos e testados. A distribuição manual dos nós sensores permite o estabelecimento de padrões geométricos favoráveis à comunicação (rede satisfatoriamente conectada) e o estabelecimento de uma cobertura geográfica equilibrada, além de, na maior parte dos casos, excluir a necessidade de determinação da localização dos nós.

2.3 – Arquitetura das Redes de Sensores Sem Fio

Uma Rede de Sensores Sem Fio é composta tanto por elementos de hardware como por elementos de software. Os elementos de hardware são os dispositivos eletrônicos capazes de realizar processamento dos dados (microcontroladores), o sensoriamento (sensores), o armazenamento dos dados (memórias), o fornecimento de energia aos nós (baterias) e o estabelecimento da comunicação (rádio e antena).

Os elementos de software são os responsáveis pelo controle das atividades dos nós e da rede de forma geral, isto é, são os elementos lógicos que coordenam o acionamento dos elementos de hardware para a realização de uma determinada tarefa. Incluem-se nos elementos de software das WSN os protocolos de comunicação e possíveis sistemas operacionais embarcados nos nós.

O conjunto hardware-software determina a eficiência de todo o sistema da rede, tanto do ponto de vista funcional como do ponto de vista de economia de energia.

2.3.1 – Arquitetura de hardware

Os hardwares desenvolvidos nos projetos de nós sensores para as WSN apresentam configurações diversificadas e comumente não estão atrelados a nenhuma aplicação específica. De forma geral, os microcontroladores, conversores A/D e o rádio e a antena utilizados para a comunicação compõem uma unidade fixa em cada projeto de nó sensor, sendo as placas contendo os dispositivos sensores (temperatura, luminosidade, posição, umidade, vibração, pressão, velocidade, massa, concentração química, estresse mecânico, etc.) acopladas aos nós sensores de acordo com as necessidades da aplicação.

O tamanho dos nós depende da tecnologia e dos componentes utilizados no projeto do hardware. Alguns nós sensores são desenvolvidos em circuitos customizados (ASIC – *Aplication-Specific Integrated Circuit*), como é o caso do *Smart Dust* [HSU98][DUS02], apresentando uma área de poucos milímetros quadrados. Outros são desenvolvidos a partir da confecção de placas de circuitos impresso com circuitos integrados, como é o do μAmps [MIC02b] e Mica2Dot [MIC02].

Muitos hardwares para Redes de Sensores Sem Fio foram desenvolvidos e permanecem em contínuo processo de aperfeiçoamento. Destacam-se os seguintes projetos:

- **Projeto Mica Motes** [MIC02] (Universidade de Berkeley);
- **Projeto Smart Dust** [DUS02] (Universidade de Berkeley);
- **Projeto μAmps** [MIC02b] (MIT - *Massachusetts Institute of Technology*);
- **Projeto WINS** [WIN01] (UCLA - Universidade da Califórnia);
- **Projeto MANTIS** [MAN03] (Universidade do Colorado);
- **Projeto BEAN** [VIE04] (UFMG – Universidade Federal de Minas Gerais);
- **Projeto PicoRadio** [PIC03] (Universidade de Berkeley);
- **Projeto CodeBlue** [COD04] (*Harward Sensor Network Lab*).

A Figura 2.4 apresenta alguns desses hardwares. A Tabela 2.2 compara alguns nós utilizados em projetos de WSN, relacionando o tipo de rádio, microcontrolador, quantidade de memória e sistema operacional utilizado.

Apesar da disparidade de tecnologias e soluções adotadas nos projetos, a arquitetura de hardware para nós sensores de Redes de Sensores Sem Fio segue, de maneira geral, o mesmo padrão. A Figura 2.5 apresenta os blocos funcionais comuns em arquiteturas de nós para WSN.

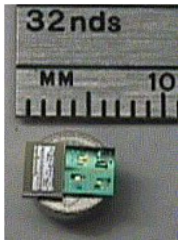
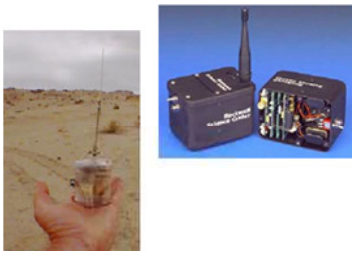
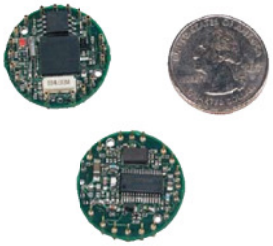
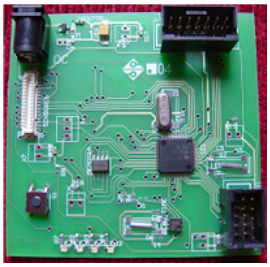
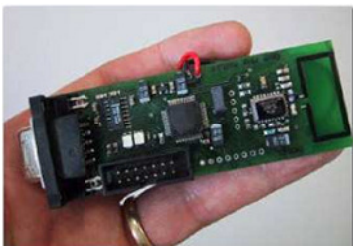
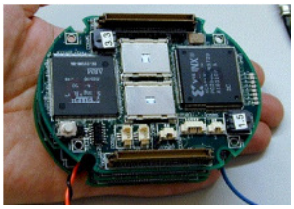
 **Smart Dust** (Universidade de Berkeley)	 **WINS** (UCLA)	 **Mica2Dot** (Universidade de Berkeley)
 **BEAN** (Universidade Federal de Minas Gerais)	 **EYES** (Universidade de Twente)	 **µAMPS** (MIT)
 **Mica Z** (Universidade de Berkeley)	 **PicoRadio** (Universidade de Berkeley)	 **CodeBlue (TelosB)** (Harward Sensor Network Lab)

Figura 2.4 – Hardwares (motes) desenvolvidos para Redes de Sensores Sem Fio[6].

[6] Referência das imagens:
Smart Dust - *http://robotics.eecs.berkeley.edu/~pister/SmartDust/*
WINS - *http://www.comp.lancs.ac.uk/~kristof/research/papers/phd/2003/*
Mica2Dot - *https://www.eol.ucar.edu/rtf/facilities/isa/internal/CrossBow/DataSheets/mica2dot.pdf*
BEAN - *http://homepages.dcc.ufmg.br/~mmvieira/publications/bean.pdf*
EYES - *http://www.utwente.nl/ctit/research/sro/old/ubricks/interviews/law.doc/*
µAMPS - *http://www-mtl.mit.edu/researchgroups/icsystems/uamps/research/sensor_net.shtml*
Mica Z - *http://www.snm.ethz.ch/snmwiki/Projects/MicaZ*
PicoRadio - *http://www.eecs.berkeley.edu/IPRO/Summary/Old.summaries/01abstracts/flb.3.html*
CodeBlue (TelosB) - *http://fiji.eecs.harvard.edu/CodeBlue*

Tabela 2.2 – Nós Sensores desenvolvidos para Redes de Sensores Sem Fio

Nó Sensor	Ano	Rádio	Microcontrolador	Memória de programa	Sistema Operacional
WeC [WEC09]	1998	TR1000	AT90LS8535	32 kB	TinyOS
PicoNode [RAB00]	2000	(proprietário)	StrongARM (SA-1100)	4 MB	n/d
Mica [MIC13a]	2001	TR1000	ATmega 103L	512 kB	TinyOS
WINS [WIN01]	2001	Connexant RDSSS9M	StrongARM (SA-1100)	4 MB	µC
Mica2Dot [MIC13b]	2003	CC1000	ATmega128L	128 kB	TinyOS
Mantis (Nymphs) [MAN03]	2003	CC1000	ATMEGA 128L	64 kB	MantisOS
Mica Z [MIC13c]	2004	CC2420 (ZigBee)	ATmega128L	128 kB	TinyOS
TelosB [TEL04]	2004	CC2420 (ZigBee)	MSP430F1611	48 kB	TinyOS
BEAN [VIE04]	2004	CC1000	MSP430F169	4 Mbit	YatOs
µAMPS [UAM13]	2004	LMX3162	StrongARM (SA-1100)	512 kB	eCos (Linux Red Hat)
IMote1 [IMO08]	2005	TC2001 (Bluetooth)	ARM7 TDMI	512 kB	TinyOS
EYES [GEO11]	2005	TR1001	MSP430F149	8 MBit	PeerOS
Tmote Sky [TMO13]	2006	CC2420	MSP430 F1611	1 MB	TinyOS
BTnode3 [BTN07]	2006	CC1000	ATMEGA 128L	256 kB	TinyOS
IMote2 [NAC08]	2007	CC2420 (ZigBee)	PXA271 XScale	32 MB	TinyOS
Shimmer [BUR10]	2008	CC2420 (ZigBee)	MSP430F1611	48kB	TinyOS
Cookies [COO13]	2008	ETRX2 TELEGESIS	MPS430	4kB	n/d
Egs [KO_10]	2010	CC2520 (Bluetooth)	ARM Cortex M3	128 kB	TinyOS

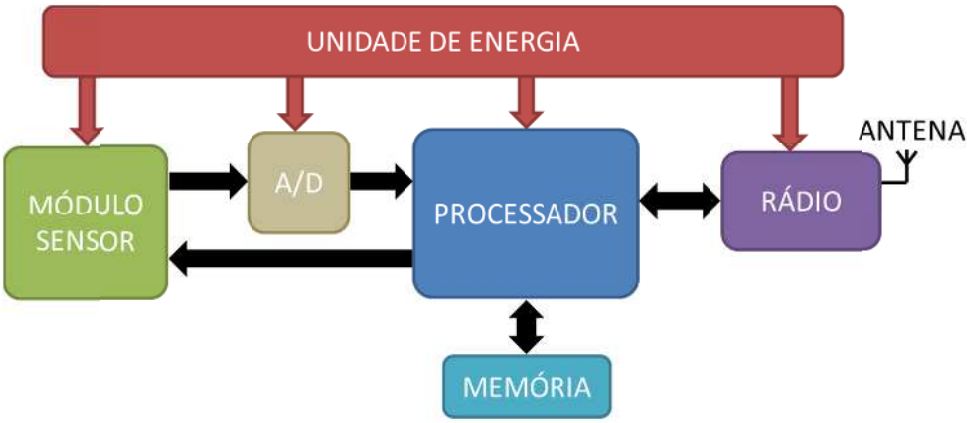


Figura 2.5 – Arquitetura de um nó sensor. Adaptação de [PAP04].

A unidade de energia consiste em uma bateria que alimenta todos os demais módulos do sistema. O módulo sensor possui um conjunto de sensores que transformam sinais analógicos de energia em energia elétrica (transdutores) que por sua vez são convertidos em sinais digitais pelo conversor A/D. O processador (geralmente um microcontrolador) realiza a as atividades programadas para a realização das tarefas do nó sensor, processando os dados recebidos pelos sensores, armazenando em memória e acionando o módulo de rádio para transmissão e recepção de mensagens. O conjunto rádio-antena é responsável, portanto, pela comunicação entre os nós sensores e/ou a estação radio base.

Considerações

Partindo da definição que uma WSN é formada por um grande número de nós sensores, para que o seu projeto seja viável, o custo de produção dos nós sensores deve ser baixo.

A produção de nós sensores em larga escala utilizando componentes de hardware integrados em um mesmo circuito (ASIC) pode reduzir o custo de produção de uma WSN, mas a determinação em tempo de projeto dos tipos de sensores presentes nos nós sensores se faz necessária. Além disso, pode haver restrições de produção de nós sensores pelas limitações tecnológicas existentes na produção de sensores integrados aos circuitos.

A maioria dos projetos de hardware para os nós das WSN ainda está voltada para aplicações experimentais, podendo ainda ser considerados *testbeds*. Apesar disso, o Smart Dust [DUS02] é o projeto que mais se assemelha ao conceito abstrato de nó sensor no que diz respeito ao seu tamanho (miniaturização), sendo necessário produzi-lo com os sensores adequados para cada tipo de aplicação.

O projeto CodeBlue [COD04], por sua vez, possui aplicação específica na área

médica, no qual o nó sensor TelosB [TEL04] é utilizado como núcleo de processamento, memória e transmissão de dados para se criar no corpo do paciente uma rede de sensores (oxímetro de pulso e eletrocardiograma, por exemplo) capaz de monitorar variáveis vitais do paciente. Contudo, o tamanho do nó TelosB (aproximadamente 5 cm por 2 cm) não facilita a sua fixação ao corpo do paciente, gerando desconforto.

Além da tecnologia existente atualmente para a fabricação de circuitos ASIC não contemplar todas as necessidades de um nó sensor (fragilidade física, vulnerabilidade eletromagnética e susceptibilidade a destruição pela exposição a micro-ondas), ainda existe o problema do tamanho das baterias necessárias para alimentar esses circuitos durante um longo período. O tamanho das baterias ainda é o maior determinante do tamanho do nó para a maioria das aplicações.

Circuitos *ultra-low power* podem ser a solução para a demanda energética dos nós sensores, reduzindo, assim, o tamanho das baterias necessárias ao seu funcionamento por longos períodos. Essa tecnologia ainda se apresenta em fase de desenvolvimento e adaptação à realidade das Redes de Sensores Sem Fio, com é o caso do projeto de uma fonte de tensão V_{DD} destinada a nós sensores [LIN13].

2.3.2 – Arquitetura de software

Apesar do projeto do hardware dos nós sensores apresentar uma grande influência no seu consumo energético, o software em execução, responsável pelo processamento dos dados e pela coordenação dos demais módulos presentes nos nós, tem um papel fundamental no consumo de energia. Por mais que os dispositivos de hardware sejam otimizados em relação ao consumo de energia, a quantidade de vezes e o tempo que eles e seus subsistemas permanecem ligados influenciam diretamente no montante de energia gasto para se realizar uma determinada operação no nó ou na rede.

Os itens seguintes descrevem os principais componentes de software presentes em uma rede de sensores sem fio: o **sistema operacional** (responsável pelo gerenciamento das atividades no nó sensor) e o **protocolo de comunicação** (responsável pela integração entre os nós e a composição da rede).

2.3.2.1 – Sistemas operacionais para Redes de Sensores Sem Fio

Um sistema operacional (OS – *Operating System*) exerce duas funções distintas em um sistema [TAN97]: (1) permite o acesso aos dispositivos de hardware de maneira mais simples que o acesso direto por funções de acesso à memória (o OS fornece funções de fácil

manipulação que são utilizadas pelos softwares para executarem as funcionalidades dos hardwares); e (2) realiza o controle de acesso aos dispositivos de hardware (escalonando o acesso aos dispositivos, com o objetivo de evitar conflitos entre processos que tentam executar funções do hardware simultaneamente).

A primeira função de um sistema operacional pode ser entendida como o OS sendo uma extensão do hardware, ou seja, o OS é uma máquina virtual. A segunda função caracteriza o SO como um gerente de recursos [TAN97].

Em uma Rede de Sensores Sem Fio, o sistema operacional deve exercer as mesmas funcionalidades de um sistema operacional convencional. Contudo alguns aspectos particulares das WSN devem ser levados em consideração [ALM04]: limitação da quantidade de memória disponível para o OS e para os softwares; a configuração dos nós deve ser feita de forma automática, visto que a quantidade de nós sensores inviabiliza sua configuração manual; e a quantidade de energia é limitada nos nós, sendo, portanto, indispensável o uso eficiente dos recursos de hardware.

A Tabela 2.3 apresenta os principais sistemas operacionais desenvolvidos ou em processo de desenvolvimento para Redes de Sensores Sem Fio, comparando suas principais características.

O sistema operacional TinyOS [TIN06] possui seu código aberto e vem sendo desenvolvido, testado e aplicado pela Universidade de Berkeley desde 1999. Inicialmente foi aplicado à plataforma WeC [POL04b], passando pelos *motes* Mica [MIC02] a partir de 2001 e em 2012 foi lançada a sua versão mais recente (versão 2.1.2). Ele possui um ambiente de desenvolvimento aberto e uma linguagem de programação dedicada denominada nesC [GAY03].

A arquitetura do TinyOS é baseada em componentes, podendo cada aplicação especificar o conjunto de componentes que utiliza. O modelo de execução de código do TinyOS pode ser tanto orientado a tarefas (pré-programadas) como a eventos (disparadas por sensoriamento do ambiente ou eventos internos dos nós). Ambos os modelos são concorrentes e não-preemptivos, não havendo, portanto, operações de bloqueio de execução de código. O escalonador do TinyOS é do tipo FIFO (*First-In-First-Out*) e não há regras de prioridades na fila de execução.

A partir de 2006, a versão 2.0 do TinyOS [TIN06] inclui um gerenciador de energia que faz com que os nós entrem em estado de SLEEP (“dormir”) quando não estão processando e/ou comunicando dados. Os nós permanecem em estado vigilante para poder realizar o sensoriamento.

Tabela 2.3 – Comparação entre os Sistemas Operacionais para Redes de Sensores Sem Fio

	TinyOS [TIN06]	**SOS** [HAN05]	**MANTIS** [BHA05]	**Contiki** [DUN04]	**YATOS** [ALM04]	**Nano-QPlus** [PAR06]	**EYES** [DUL02]
Programação dinâmica	✓	✓	✓	✓			✓
Escalonamento/ agendamento com prioridades			✓	✓		✓	✓
Execução em tempo real			✓			✓	
Módulo de gerenciamento de energia	✓	✓	✓	✓	✓	✓	✓
Multithread			✓	✓		✓	
TCP/IP				✓			
Linguagem de programação	nesC	C	C	C	C	C	C
Modelo de execução	Eventos e Tarefas	Módulos	Tarefas	Tarefas	Eventos e Tarefas	Tarefas	Eventos

O sistema operacional SOS [HAN05] foi desenvolvido pela Universidade da Califórnia. Este sistema apresenta um escalonamento cooperativo que compartilha o processador entre seus módulos, enfileirando (pelo esquema FIFO, com dois níveis de prioridade) as mensagens dos módulos para execução posterior. Não existem, portanto, processos neste sistema operacional.

O kernel[7] do SOS não pode ser substituído ou atualizado através de comunicação pelo ar. Contudo, a programação dos nós (*Dynamic Code*) é bem flexível e pode ser substituída através da troca de mensagens. O SOS não possui proteção de acesso à memória, mas um coletor de lixo (*Garbage Collector*) está presente.

O MANTIS OS [BHA05] é um projeto da Universidade de Colorado para sistemas operacionais de redes sem fio que incorpora a execução de *threads*. Ele é programado por tarefas com escalonamento preemptivo.

O módulo de economia de energia do MANTIS é baseado na política de diminuição do ciclo de operação (*duty cicle*) de acordo com seu estado. Além disso, o MANTIS OS

[7] *Kernel* é o núcleo ou cerne do sistema operacional. A sua função principal é controlar a execução dos aplicativos, estabelecendo uma ponte entre eles e o processamento real de dados feito no nível de hardware.

disponibiliza reprogramação dinâmica, atualizando o código do sistema operacional por completo nos nós sensores.

O sistema operacional para WSN Contiki [DUN04] é um projeto do Instituto de Ciência da Computação Suíço (*Swedish Institute of Computer Science*). Similarmente ao MANTIS OS, o Contiki implementa *threads* de forma preemptiva, mas ao contrário daquele, as *threads* são alocadas na memória no momento da sua criação de forma a não compartilhar espaço com as demais *threads* do sistema. O Contiki é baseado em programação por eventos.

O YATOS (*Yet Another Tiny Operating System*) [ALM04] é o primeiro sistema operacional para rede de sensores sem fio desenvolvido no Brasil. Juntamente com o nó sensor BEAN [VIE04], o YATOS faz parte de um projeto Sensor Net da Universidade Federal de Minas Gerais (UFMG).

Podendo ser dirigido por eventos e/ou tarefas, o YATOS classifica os eventos de acordo com a frequência com que são executados: aperiódicos (sem uso de temporizadores, acionados através de interrupções), periódicos (ocorrem em intervalos de tempos definidos) e "tiro único" (eventos programados para ocorrer uma única vez, necessitando de um temporizador). As tarefas, por sua vez, são executadas seguindo a lista de prioridade presente no escalonador de tarefas. Tal escalonador é cooperativo, isto é, a tarefa (trecho de código executável) que está com a posse do processador decide quando liberá-lo para que outras possam executar seu código.

O YATOS possui um sistema de economia de energia que, entre outras políticas, coloca o rádio em modo de baixa energia enquanto o sistema está processando os dados.

O sistema operacional Nano-QPlus [PAR06] é um projeto desenvolvido na Coréia do Sul e possui as seguintes características: permite a execução de *threads*, o escalonamento é realizado com prioridade, a execução das operações ocorre em tempo real, o processamento é baseado em tarefas e possui um sistema de reprogramação dinâmica.

O EYES (*Energy Efficient Sensor Networks*) [DUL02] é o sistema operacional do projeto europeu homônimo que esteve ativo durante os anos de 2002 a 2005. O objetivo do projeto EYES foi o desenvolvimento de uma WSN com características de auto-organização e eficiência na utilização de energia.

O sistema operacional EYES foi desenvolvido pela Universidade de Twente na Holanda. Suas características são: processamento baseado em tarefas, processamento FIFO com prioridades e um sistema de gerenciamento de energia simples que garante que todo o sistema entre em estado de SLEEP quando nenhuma atividade está sendo realizada no nó sensor.

Um mecanismo inteligente de computação distribuída foi programado no sistema operacional EYES. O gerenciador de recursos do sistema verifica se há memória, energia ou até mesmo velocidade compatível para executar uma determinada tarefa e caso o nó sensor não disponha dos recursos necessários, ele envia uma mensagem para outros nós adjacentes efetuarem tal tarefa. Existe, portanto, uma forma de RPC (*Remote Procedure Call* – Chamada de Procedimento Remoto) implementada no sistema operacional EYE.

Considerações

O conceito de sistema operacional em execução nos nós sensores muitas vezes pode ser substituído pelo de software embarcado, pois a sua implementação não precisa ser tão genérica para suportar diferentes dispositivos e rodar aplicações variadas. Na maior parte das vezes, softwares flexíveis requerem mais memória e maior capacidade e tempo de processamento, elementos que diminuem o tempo de vida do nó sensor.

Os recursos disponíveis nos sistemas operacionais vão muito além das necessidades organizacionais e de processamento de um nó sensor. A substituição dos programas e do próprio sistema operacional através de mensagens não é uma característica relevante para as WSN, uma vez que, por definição, elas são desenvolvidas de acordo com a aplicação a que se destinam. Portanto, se a aplicação da WSN é específica e os nós sensores possuem um tempo de vida limitado, a proposta de reprogramação do seu conteúdo de software não só contradiz a característica da rede em ser dependente da aplicação, como requer uma quantidade maior de memória (necessária à reprogramação) e utiliza uma quantidade de energia adicional para a transmissão das mensagens de substituição do software.

A característica de alguns dos sistemas operacionais apresentados em possuírem tanto a programação orientada a eventos como a tarefas também recai sobre o fato das WSN serem desenvolvidas para aplicações específicas. Uma vez que uma rede com comportamento orientado somente a eventos é desenvolvida, por exemplo, a parte do sistema operacional que cuida das tarefas deve ser desativada completamente, caso contrário uma quantidade de energia extra estaria sendo utilizada nos nós sensores.

Os sistemas operacionais aqui apresentados estão sempre associados a uma ou mais plataformas de desenvolvimento específicas de WSN. Essas plataformas são experimentais e, por isso, permitem a utilização de diversos tipos de dispositivos sensores, sendo utilizadas também com aplicações variadas. O conceito e as implementações flexíveis de um sistema operacional são, portanto, pertinentes neste contexto. Para o desenvolvimento específico de uma WSN com aplicação definida, a utilização de um software embarcado

dedicado seria o procedimento mais adequado.

2.3.2.2 – Protocolos de comunicação para Redes de Sensores Sem Fio

Os protocolos de comunicação utilizados em redes *ad hoc* não são indicados para Redes de Sensores Sem Fio por requererem muito espaço em memória e possuírem cabeçalhos demasiadamente extensos (refletindo em um gasto energético elevado devido ao tempo necessário para transmissão e recepção de mensagens). Por isso, alguns protocolos foram desenvolvidos especialmente para as WSN.

O agrupamento dos protocolos de comunicação para WSN conforme as camadas de rede do modelo OSI (*Open Systems Interconnection*), conforme sugestão de [RUI04a], pode ser realizado com o objetivo de distinguir as suas funcionalidades, isto é, classificar os protocolos de acordo com a especificidade de cada camada.

A Tabela 2.4 apresenta alguns protocolos de comunicação utilizados em Redes de Sensores Sem Fio, seguindo a divisão de camadas do modelo OSI. A literatura apresenta uma quantidade vasta de protocolos destinados para WSN, mas apenas os mais utilizados são apresentados aqui, uma vez que não é o foco deste trabalho o conhecimento aprofundado dos mesmos.

Tabela 2.4 – Protocolos para as Redes de Sensores Sem Fio

Camada	Protocolos
Física	Transmissão por rádio frequência, ótica, infravermelho
Enlace	S-MAC, T-MAC, ARC, TRAMA, TSCH
Rede	Flooding, Gossiping, Direct Difusion, Rumour Routing, SPIN, LEACH, PEGASIS, TEEN, SPEED, SAR, GEAR
Transporte	PFSQ, ESRT

O padrão de comunicação sem fio IEEE 802.15.4 [IEE11] (e sua extensão IEEE 802.15.4e [IEE12], que introduz uma nova camada de enlace) também é comumente utilizados em projetos de WSN por ter sido desenvolvido para redes sem fio de baixa taxa de transmissão e possuírem mecanismos de gerenciamento de energia. Trata-se de uma padronização que reúne especificações da camada física e da camada de enlace, fornecendo, portanto, a base para o desenvolvimento das camadas superiores, mais próximas da aplicação.

Segue um breve comentário sobre cada camada do modelo OSI e algumas particularidades dos protocolos desenvolvidos para Redes de Sensores Sem Fio. Excetuando-

se a camada física, todas as outras camadas se encontram embutidas no Sistema Operacional dos nós sensores, quando estes dispõem de um.

2.3.2.2.1 – Camada Física

A Camada Física de uma rede de sensores sem fio é a responsável pela transmissão dos dados de um ponto a outro no espaço tridimensional e é mais frequente a utilização de ondas de rádio frequência por serem omnidirecionais, não havendo, assim, a necessidade do emissor e do receptor estar alinhados.

A comunicação ótica é a que consome menor energia por quantidade de bits transmitidos [RUI04a], mas está sujeita às condições atmosféricas e é unidirecional. O nó sensor *Smart Dust* [MIC02b], por exemplo, possui uma comunicação ótica realizada através de um CCR (*Corner Cubic Reflector*), permitindo uma comunicação de até um quilômetro de distância.

O sistema de comunicação realizado por infravermelho é unidirecional e tem a desvantagem de ter curto alcance, geralmente poucos metros.

2.3.2.2.2 – Camada de Enlace

A Camada de Enlace é responsável pelo acesso ao meio e controle de erros. Segundo [TAN97], “a principal tarefa da camada de enlace de dados é transformar um canal de transmissão bruta de dados em uma linha que pareça livre de erros de transmissão”.

Os protocolos MAC (*Medium Access Protocol* – Protocolo de Acesso ao Meio), pertencentes à Camada de Enlace, têm como principal objetivo evitar a colisão de mensagens no meio, gerenciando as transmissões de forma a garantir que dois nós não acionem seus rádios para a transmissão de dados ao mesmo tempo.

Em redes sem fio tradicionais, um protocolo MAC adequado deve ser capaz de gerenciar os nós com justiça, reduzir o tempo de latência, garantir alto *throughput* (quantidade de dados transferidos de um lugar para outro em um determinado tempo) e maximizar a utilização da largura de banda disponível. Contudo, em WSN esses atributos são secundários, uma vez que o que se deseja é a garantia de uma cobertura de sinal que abranja todos os nós sensores, além do uso reduzido da energia disponível nos nós [YE_06].

O ponto crítico da camada de enlace em WSN é o acesso ao meio, pois sendo o acionamento do rádio a ação do nó sensor que utiliza a maior quantidade de energia, um meio mal gerenciado pode provocar diversas tentativas de transmissão e retransmissão de dados que consumirão uma quantidade de energia adicional aos nós.

Os três principais eventos associados à transmissão de dados que possuem grande impacto no gasto energético dos nós são: **colisões** (quando dois nós tentam enviar dados ao mesmo tempo; terminal escondido e estação exposta [TAN97]), ***overhearing*** (quando um nó recebe dados destinados para outros nós), e **escuta ociosa** (*idle listening* – quando o nó permanece escutando o canal que não possui tráfego).

A seguir é feito um breve resumo dos quatro principais protocolos desenvolvidos para a Camada de Enlace de WSN (Tabela 2.H) encontrados na literatura, indicando as principais estratégias utilizadas para minimizar esses eventos que reduzem o tempo de vida dos nós sensores.

O protocolo **S-MAC** (Sensor-MAC) [YE_06] foi desenvolvido especialmente para Redes de Sensores Sem Fio, utilizando a sequência de mensagens RTS-CTS-DATA-ACK para evitar as colisões durante a transmissão. Quando dois nós tentam transmitir ao mesmo tempo e há conflito de sinalizações RTS, cada nó aguarda um tempo aleatório para tentar uma nova transmissão. O problema de *overhearing* é minimizado também, pois os nós entram em estado de SLEEP quando recebem um RTS não destinados à eles.

A escuta ociosa também é considerada no S-MAC, pois utilizando *duty cicles* pequenos, os nós permanecem mais tempo no estado SLEEP que no estado LISTENING. Além disso, o esquema de cabeçalho do S-MAC é reduzido, diminuindo o tráfego de informações desnecessárias na rede.

Os grandes problemas gerados por esse protocolo é (1) a necessidade de sincronização entre os nós para efetuar a troca de mensagens RTS-CTS-DATA-ACK e (2) o aumento na latência, uma vez que muitos nós podem estar no estado de SLEEP devido ao pequeno *duty cicle*.

O protocolo **T-MAC** (*Time-out-MAC*) [DAM03] é uma variação do protocolo S-MAC. O T-MAC utiliza o mesmo esquema RTS-CTS-DATA-ACK do S-MAC, mas altera o ciclo de trabalho (*duty cicle*) dos nós de acordo com o tráfego da rede. Com isso, ele tenta reduzir a latência e aumentar a economia de energia nos nós.

Além do problema da sincronização, o T-MAC sofre do problema conhecido como “dormir cedo” [RUI04a], isto é, um nó pode dormir enquanto o outro ainda tenta transmitir dados para ele, uma vez que o ciclo de trabalho de cada nó é variável de acordo com a atividade da rede perceptível no nó.

O protocolo **ARC** (*Adaptive Rate Control*) [WOO01] realiza o ajuste na largura de banda com o objetivo de alocar as transmissões a serem realizadas pelos nós sensores. Ele se baseia no protocolo CSMA (*Carrier Sense Multiple Access*), mas adiciona um atraso aleatório

antes do estágio de escuta do meio, com o objetivo de evitar repetidas colisões. Este esquema permite que o nó entre em estado de SLEEP antes de “escutar” o meio.

O ARC é indicado para nós sensores que não dispõem de muito recurso computacional, visto que não há complexidade na sua execução e sinais de controle que precisem ser interpretados.

O protocolo **TRAMA** (*Traffic Adaptive Multiple Access*) [RAJ03b] é baseado em um algoritmo de eleição distribuído que leva em consideração o tráfego da rede para nomear o nós que tem direito de acesso ao meio. Ele utiliza o protocolo de alocação estática do canal TDMA (*Time Division Multiplexing Access*).

O TRAMA garante a inexistência de colisões na rede tanto para comunicações do tipo *unicast* (um para um), *broadcast* (um para todos) e *multicast* (um para muitos). Contudo, devido ao processo de eleição e ao próprio conceito embutido no protocolo TDMA, este tipo de abordagem gera uma alta latência na rede.

Considerações

Sem dúvida alguma o acesso ao meio é o principal responsável pelo gasto adicional, isto é, não necessário, de energia em uma Rede de Sensores Sem Fio. Devido à alta densidade de nós na rede, eventos de colisão podem ocorrer constantemente, havendo perda de dados e necessidade de retransmissão.

Assim sendo, os protocolos MAC são fundamentais para o gerenciamento do meio, evitando, assim, gastos excessivos de energia e atrasos na execução de tarefas na rede. Quanto menor o tamanho das mensagens e menos vezes os nós forem acionados para transmissão, maior é a economia de energia não só pelo fato do rádio ser acionado durante menos tempo, mas também pelo fato de haver redução no número de colisões que acarretariam a necessidade de retransmissão de mensagens.

Uma das táticas mais utilizadas para se reduzir o tamanho das mensagens e evitar o chamado *packet overhead* (tamanho excessivo do pacote) é reduzir o tamanho do cabeçalho da mensagem, o que geralmente envolve mudança no protocolo de comunicação. Uma segunda forma seria realizar a compactação os dados, que envolve mais tempo de processamento. E ainda uma terceira maneira seria utilização de pacotes de transmissão com tamanho variável, a depender do tamanho da mensagem. Alguns rádios utilizam protocolos cujos pacotes transmitidos são de tamanho fixo, havendo gasto energético excessivo quando a mensagem enviada é menor que o tamanho disponível para ela no pacote e no caso de uma mensagem ser maior que o tamanho padrão do pacote, há a necessidade de se dividir a mensagem e transmitir o cabeçalho do pacote mais de uma vez (uma vez para cada pacote).

Os protocolos MAC que envolvem tempos na execução de tarefas (*duty cicles*, por exemplo) devem levar em consideração o tempo necessário para "acordar" o rádio, seja para iniciar uma transmissão ou para estar pronto para uma recepção. Isso aumenta o tempo de latência e deve ser considerado no momento do acionamento do rádio. O rádio TR1000, por exemplo, necessita de 20 μs para sair do modo SLEEP e entrar no modo LISTENING [RFM99].

2.3.2.2.3 – Camada de Rede

O objetivo da camada de rede é rotear as mensagens que são transmitidas pelo canal de comunicação. Trata-se, portanto, de identificar o destinatário da mensagem e encontrar um caminho de tráfego que una o nó origem ao nó destino. Existem três tipos de roteamento [RUI04a]: **plano** (todos os nós da rede desempenham o mesmo papel na rede), **hierárquico** (existem nós fontes e nós líderes de grupos – clusters) e **geográfico** (inclui a localização geográfica/cartesiana dos nós como estratégia de roteamento). A classificação do roteamento com sendo geográfico pode estar associada aos tipos plano e hierárquico, não sendo uma classificação mutuamente exclusiva.

O endereçamento dos nós pode ser uma estratégia auxiliar na localização dos nós durante o processo de roteamento. Contudo, a identificação dos nós com endereços globais não é um recurso adequado para WSN, visto que redes dessa natureza possuem centenas ou milhares de nós, conferindo ao identificador de endereço um tamanho demasiadamente grande e encarecendo, assim, o custo de envio de mensagens identificadas. Outro fator que [RUI04a] aponta sobre a não necessidade de haver um endereçamento global é o objetivo das consultas realizadas em WSN, geralmente voltadas para a obtenção de informações de uma determinada região e não de um nó sensor específico. Esse apontamento leva em consideração, portanto, a aplicação em execução na rede.

Os protocolos da camada de rede ainda podem ser divididos em **pró-ativos** e **reativos** [RAB00][LI_04]. Nos protocolos pró-ativos a rede faz uma atualização das rotas de forma periódica; nos reativos, somente quando há a necessidade de transferência de dados é que as rotas são estabelecidas. Segundo [RAB00], as redes com poucos nós se beneficiam de protocolos reativos, enquanto que os protocolos pró-ativos são mais indicados para redes com grande número de nós e alta taxa de transmissão. Tal distinção (pró-ativo ou reativos) não é considerada quando os nós não precisam "conhecer" a topologia da rede para fazer o roteamento.

Os protocolos da camada de rede também são conhecidos como protocolos de roteamento. A Tabela 2.5 apresenta os principais protocolos dessa camada e a seguir são

descritos, de forma resumida, o funcionamento desses protocolos.

Tabela 2.5 – Principais Protocolos de Roteamento (Camada de Rede) para WSN

Protocolo	Proativo/ Reativo/ Híbrido	Adaptação aos recursos de energia	Hierárquico (*clsters*)/ Plano	Baseado na localização	QoS	Agregação de dados	Rota única/ Múltiplas rotas	Balanceamento de carga
Direct Diffusion [INT00]	Híbrido		Plano			✓	Múltiplas	
SAR [SOH00]	Híbrido	✓	Plano		✓		Múltiplas	✓
LEACH [HEI00]	Proativo		Hierárquico			✓	Única	✓
TEEN [MAN01]	Proativo		Hierárquico			✓	Única	
PEGASIS [LIN02]	Proativo		Hierárquico			✓	Única	✓
SPEED [HE_03]	Reativo		Plano	✓	✓		Única	✓
Flooding [ZHA09]	Reativo		Plano				Múltiplas	
Gossiping [HED88]	Reativo		Plano				Única	
Rumour Routing [BRA02]	Reativo		Plano			✓	Múltiplas	
SPIN [HEI99]	Reativo	✓	Plano			✓	Múltiplas	✓
GEAR [YU_01]	Reativo	✓	Plano	✓			Única	

O **Flooding** [ZHA99] é o mais simples dentre os protocolos de roteamento, sendo também conhecido como Flooding Clássico (*Classic Flooding*). Quando um nó deseja transmitir uma mensagem, ele a transmite para os seus vizinhos. Cada vizinho que recebeu a mensagem verifica se é uma mensagem que ainda não foi recebida e, neste caso, armazena e retransmite a mensagem para seus vizinhos, com exceção do nó que transmitiu a mensagem. O algoritmo

do protocolo converge quando todos os nós da rede recebem uma cópia da mensagem.

Uma variante do Flooding Clássico é o protocolo **Gossiping** [HED88] [HEI99]. O Gossiping busca economizar a energia reduzindo o número de nós que recebem as mensagens. Quando um nó recebe uma mensagem, ele a retransmite somente para alguns nós vizinhos, sendo essa escolha realizada de forma aleatória. Devido a este caráter aleatório de seleção de nós, o nó que enviou a mensagem também pode ser eleito como um nó que a receberá retransmitida, garantindo, assim, uma melhor probabilidade de espalhamento da mensagem.

O protocolo **Direct Diffusion** [INT00] é composto de quatro tipos de elementos: *interests*, *data messages*, *gradients* e *reinforce*. As *interests* são mensagens nas quais os usuários da rede fazem requisição de dados aos nós sensores enquanto que as *data messages* são mensagens que contém os dados dos nós (valores dos sensores). A trajetória das mensagens de *interests* criam os *gradients* que interligam os nós entre o nó de origem e o nó destino. A rede reforça (*reinforce*) uma ou algumas dessas trajetórias, determinando rotas mais favoráveis para o tráfego das mensagens.

A ideia central do protocolo **Rumor Routing** [BRA02] é a utilização de agentes que criam trajetórias para as mensagens que são enviadas pela rede. Mensagens posteriores se beneficiam dos roteamentos já traçados pelos agentes. Os agentes são enviados pela rede para executarem um “passeio aleatório” durante o qual é realizado o mapeamento da rede. Cada nó da rede mantém uma lista de seus vizinhos e participação em rotas de mensagens.

O protocolo **SPIN** (*Sensor Protocol for Information via Negotiation*) [HEI99] baseia-se em dois princípios: negociação e adaptação aos recursos disponíveis. Para isso, o SPIN utiliza três tipos de mensagens para se comunicar: ADV, REQ e DATA. Quando um nó sensor possui um dado a ser transmitido, ele envia um sinal de ADV para seus vizinhos; os nós vizinhos que ainda não possuem tal dado transmitem um sinal de REQ; por fim, o nó detentor do dado envia o pacote DATA, contendo o dado. O processo continua até a mensagem atingir a estação rádio base.

O roteamento **LEACH** (*Low-Energy Adaptive Clustering Hierarchy*) [HEI00] é um tipo de roteamento hierárquico no qual há a formação de líderes de grupos (*clusters*). Esses líderes são responsáveis por receber e agregar os dados dos demais nós do grupo (podendo haver compressão e/ou combinação de dados) e enviar através de uma única mensagem (*hop* único) os dados para a estação rádio base. De tempos em tempos (*rounds*) há a troca de líderes, fazendo com que o gasto energético seja distribuído entre os nós, uma vez que a energia do líder é mais utilizada que as dos outros nós. A escolha dos líderes é realizada de forma probabilística dentro dos próprios nós (auto-eleição), levando em consideração a

quantidade de energia disponível e a quantidade de *clusters* que a rede deve ter (determinado de acordo com a topologia da rede).

O **PEGASIS** (*Power-Efficient Gathering in Sensor Information Systems*) [LIN02] pode ser considerado um aprimoramento do LEACH no quesito de economia de energia [KUM12]. Cada nó sensor transmite seus dados para o vizinho mais próximo (distância geográfica), produzindo, assim, uma cadeia que conduz a mensagem até as estações rádio base. A cada nó sensor em que a mensagem passa é realizada a fusão dos dados. Quando um nó morre, a cadeia é reconstruída.

O protocolo **TEEN** (*Threshold Sensitive Energy Efficient Sensor Network Protocol*) [MAN01a] é hierárquico e reativo, havendo a agregação dos dados presentes nas mensagens em vários níveis de *clusters*, formados a partir da proximidade dos nós, até que a mensagem seja recebida por uma estação rádio base. O TEEN não é recomendado para aplicações que dependam de sensoriamento periódico, uma vez que existe um tempo máximo (*threshold*) para que um determinado nó leia os valores dos seus sensores e os enviem ao nó centralizador do *cluster*. Se esse tempo máximo não é atingido por um nó, seus dados ficam ausentes no *cluster* e, por consequência, os dados não chegam à estação rádio base. O APTEEN (*Adaptive Periodic Threshold Sensitive Energy Efficient Sensor Network Protocol*) [MAN01b] é um aprimoramento do TEEN que permite tanto a coleta periódica de dados como a propagação dos dados na ocorrência de eventos programados.

O **SPEED** (*Stateless Protocol for End-to-End Delay*) [HE-03] é um exemplo de protocolo desenvolvido com a preocupação na Qualidade de Serviço (QoS). Cada nó sensor retém os dados dos sensores de todos seus vizinhos (garantia de redundância), sendo a trajetória até a estação rádio base determinada pela posição geográfica dos nós. O protocolo SPEED tenta garantir a velocidade de recebimento de mensagens dos nós sensores até a estação rádio base e evita a formação de situações de congestionamento de mensagens através de uma regulação do tráfego.

Outro protocolo voltado para QoS é o **SAR** (*Sequencial Assignment Routing*) [SOH00]. A decisão de roteamento é determinada levando em consideração três fatores: a quantidade energia disponível, a qualidade de serviço em cada trajetória e o nível de prioridade estabelecido para a mensagem a ser transmitida. Além disso, o protocolo leva em consideração que uma única rota não garante a entrega do pacote a uma estação radio base devido a possíveis falhas, portanto, múltiplas rotas são determinadas.

O protocolo de rede **GEAR** (*Geographic and Energy-Aware Routing*) [YU_01] foi desenvolvido para aplicações nas quais o usuário busca obter informações de uma determinada área coberta pela rede de sensores. Os nós sensores necessitam possuir algum tipo de

hardware capaz de identificar a sua localização no espaço (GPS, por exemplo). Quando um nó sensor recebe uma mensagem, ele verifica se existe um nó vizinho que esteja mais próximo (distância geográfica) do nó destino da mensagem; se houver, ele envia a mensagem para este nó, caso contrário, a escolha do nó destino é realizada baseando-se em uma função de custo que envolve a disponibilidade energética de cada nó vizinho. Se o nó que receber a mensagem estiver na região de interesse, há uma propagação recursiva da mensagem (*recursive geographical flooding*) para os nós vizinhos também pertencentes à região.

Considerações

O protocolo Gossiping apresenta três problemas [HEI99]: implosão (como cada nó envia os dados para seus vizinhos, determinados nós recebem os dados mais de uma vez), *overlap* (a concatenação de dados pode resultar de múltiplas cópias de dados de uma mesma região) e não há economia de energia nos nós, visto que não há otimizações com relação a roteamento e redução de dados transmitidos, que ocorre de forma redundante. O Gossiping ainda pode gerar um atraso considerável na entrega da mensagem à estação rádio-base ou ainda não haver a entrega da mensagem.

O protocolo SPIN resolve o problema de implosão através das mensagens de sinalização, mas há dois problemas nesse tipo de protocolo. O primeiro refere-se ao tamanho do pacote a ser transmitido em relação ao tamanho dos sinais de ADV e REQ. Algumas redes enviam apenas alguns bits de sinalização quando ocorrem determinados eventos, mas a troca de mensagens ADV-REQ-DATA pode ser mais custosa em termos energéticos que o envio simples da mensagem DATA. O segundo problema é que um nó pode consumir energia em excesso se estiver interessado em vários ADV [DAI05].

A principal vantagem do LEACH é a economia de energia, pois os nós que não são líderes permanecem em estado de SLEEP quando não estão transmitindo dados para o líder. A necessidade de sincronismo entre os nós é uma desvantagem desse protocolo uma vez que a auto-eleição de líderes e o tráfego de dados são realizados de forma sincronizada. Outra desvantagem é a existência de apenas um *hop* no envio de dados entre os líderes dos grupos e a estação rádio base. Em redes de grande extensão geográfica, tal abordagem pode consumir muita energia dos nós líderes, fazendo com que os nós mais distantes da estação rádio-base possivelmente criem grupos que não se comuniquem mais a partir de um determinado momento, mesmo possuindo um pouco de energia (suficiente para fazer *multi-hops*).

Uma comparação realizada em [KUM12] conclui que o protocolo APTEEN resulta em um menor consumo de energia que o protocolo LEACH.

2.3.2.2.4 – Camada de Transporte

A Camada de Transporte tem como objetivo garantir a entrega de dados para o nó destinatário. Segundo [RUI04b], a maioria das aplicações de WSN admitem perda de dados e, por isso, a maioria delas não possui essa camada na sua pilha de protocolos.

Contudo, os protocolos PSQF (*Pump Slowly*, *Fetch Quickly*) [WAN02] e ESRT (*Event-to-Sink Reliable Transfer*) [SAN03] se propõem a evitar a perda de dados em WSN. Deve-se ressaltar que a transmissão redundante de algoritmos como o *Direct Difusion* garante de forma indireta a entrega dos dados, visto que eles trafegam por caminhos distintos na rede até a estação rádio base.

Para a maioria das Redes de Sensores Sem Fio, a Qualidade de Serviço (QoS – *Quality of Service*) está mais relacionada com a sobrevivência dos nós sensores (ampliando o tempo de vida da rede) do que a perda de dados.

2.3.2.2.5 – Considerações sobre as camadas e os protocolos

Apesar de todo o cuidado na elaboração de protocolos específicos para as Redes de Sensores Sem Fio, [HEI00] ressalta que o protocolo de rede mais eficiente em termos de consumo energético depende da topologia da rede. Portanto, a distribuição dos nós na área de interesse influencia significativamente no desempenho do protocolo escolhido. Uma vez que não ocorre uma adaptação dinâmica do protocolo em função da distribuição dos nós, pode ocorrer que o protocolo escolhido para determinada rede não seja adequado à topologia estabelecida em uma distribuição aleatória de nós.

A aplicação mais uma vez tem um papel fundamental na determinação do protocolo mais adequado a ser programado nos nós sensores, uma vez que uma rede reativa tem um comportamento distinto de uma rede baseada em consulta.

A maioria das WSN não necessita que todas as camadas estejam presentes para que haja uma comunicação eficiente, isto é, satisfatória à aplicação. Quanto maior a quantidade de camadas, maior o tamanho das mensagens a serem transmitidas, maior o gasto energético associado aos envios e recebimentos das mensagens e, consequentemente, menor será o tempo de vida da rede.

Há ainda a possibilidade de juntar funções específicas de duas ou mais camadas em uma única. A tarefa de gerenciamento da rede pode (ou deve) ocorrer em conjunto com roteamento, envio de mensagens, etc. Separar em camadas pode ser útil em redes tradicionais, mas significa gastar mais energia nas Redes de Sensores Sem Fio.

A análise de desempenho dos protocolos muitas vezes não desconsidera a influência dos protocolos utilizados nas camadas inferiores. Assim, um protocolo da camada de rede pode ser melhor que outro em termos de economia de energia se for utilizado um determinado protocolo da camada de enlace. A utilização de outro protocolo da camada de enlace pode modificar o resultado obtido na análise dos protocolos da camada de rede.

2.4 – Conhecimento da Rede e Comunicação

Grande parte dos artigos que apresentam novos protocolos muitas vezes não leva em consideração que os nós sensores não dispõem de todos os dados necessários para executar suas decisões. Muitos protocolos partem do pressuposto que informações globais da rede (vizinhança, melhor trajetória, mapa de energia da rede etc.) estão disponíveis nos nós, sem levar em conta o gasto energético associado à obtenção desses dados.

A falha no raciocínio encontra-se na dualidade de entendimento da rede como um objeto físico real ou como a entidade abstrata de um grafo. As Redes de Sensores Sem Fio podem ser estudadas e analisadas tanto do ponto de vista do nó, que possui um conhecimento limitado da rede, como a partir de um grafo, que oferece um conhecimento absoluto da estrutura da rede a partir de uma visão externa da mesma.

Sob a ótica de um grafo, os nós da rede (nó sensor ou estação rádio base) são representados como os vértices do grafo e a possibilidade de comunicação entre os nós são as arestas do grafo. Matematicamente, o grafo de uma Rede de Sensores Sem Fio pode ser escrito como $G(t) = (V(t), E(t))$, onde $G(t)$ é o grafo formado pela rede no instante t, $V(t)$ é o conjunto dos nós em operação (vértices do grafo) no instante t e $E(t)$ são as comunicações possíveis entre os nós (arestas do grafo) no instante t. Além disso, para cada aresta $E_i(t)$ existe um par de vértices $\{V_{i1}(t),V_{i2}(t)\}$ que identifica os vértices que estão ligados pela aresta, isto é, quais nós são passíveis de comunicação no instante t. A Figura 2.6 apresenta uma Rede de Sensores Sem Fio representada como um grafo.

O que determina a possibilidade de comunicação entre dois nós de uma rede sem fio é a relação existente entre potência de transmissão do rádio transmissor pertencente ao nó que envia a mensagem e sensibilidade do rádio receptor pertencente ao nó que recebe a mensagem. Se a potência do sinal que atinge o nó receptor for suficiente para sensibilizar o seu rádio, a comunicação é possível. Em outras palavras, se a potência recebida pelo rádio do receptor for maior ou igual à sensibilidade do seu rádio, ele é capaz de receber o sinal transmitido.

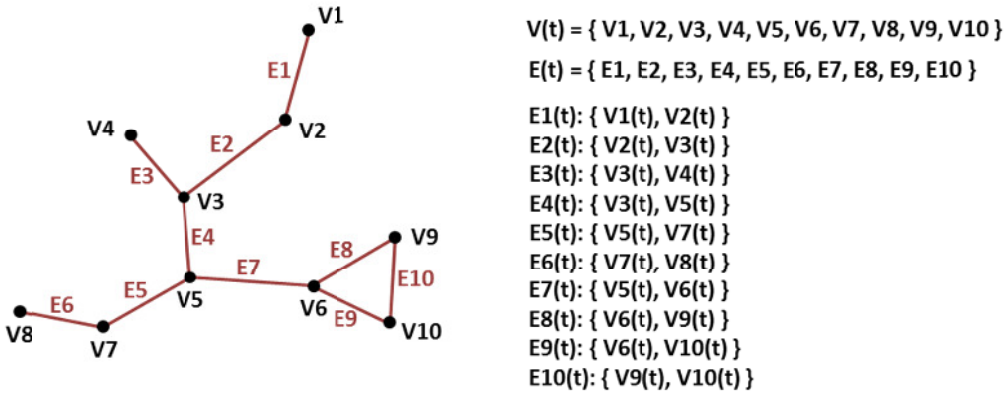


Figura 2.6 – Uma WSN representada como um grafo. Os vértices Vi são os nós sensores (ou estações rádio base) e as arestas Ei são ligações que identificam a possibilidade de comunicação entre dois nós.

Uma das formas mais utilizadas para o cálculo da potência recebida pelo receptor em função da potência transmitida pelo emissor é a Equação de Friis [HAY08] (Equação 2.1).

$$P_R = P_T G_R G_T \left(\frac{\lambda}{4\pi d}\right)^2 \tag{2.1}$$

onde:
P_R = potência recebida [mW]
P_T = potência transmitida [mW]
G_R = ganho da antena do receptor
G_T = ganho da antena do transmissor
λ = comprimento de onda [m]
d = distância entre o emissor e o receptor [m]

A Equação 2.1 pode ser escrita em termos de decibéis relativos ao miliWatt (dBm), medida comumente utilizada no cálculo da potência de antenas, tomando a forma da Equação 2.2.

$$P_R = P_T + G_R + G_T + 20log_{10}\left(\frac{\lambda}{4\pi d}\right) \tag{2.2}$$

onde:
PR = potência recebida [dBm]
PT = potência transmitida [dBm]
GR = ganho da antena do receptor [dB ou dBi]
GT = ganho da antena do transmissor [dB ou dBi]
λ = comprimento de onda [m]
d = distância entre o emissor e o receptor [m]

Se a potência recebida pelo receptor (P_R) for maior ou igual à sensibilidade do seu

rádio, a mensagem pode ser considerada recebida (podendo ou não conter erros). O gráfico da Figura 2.7 apresenta as curvas da Equação de Friis para o rádio CC2500 [CC_09], que opera na frequência de 2,4 Ghz (λ = 0,125 m). A configuração de potência de transmissão para esse rádio pode ser ajustada em -8 dBm, -10 dBm e -12 dBm e a sensibilidade de recepção em -82 dBm e -88 dBm. O ganho utilizado para as antenas é de 2 dBi [PUL07]. Assim, se a sensibilidade do receptor for ajustada para -88 dBm e o emissor transmitir a -10 dBm, a comunicação poderá ser realizada se os par emissor-receptor estiver a uma distância máxima de aproximadamente 125,22 m.

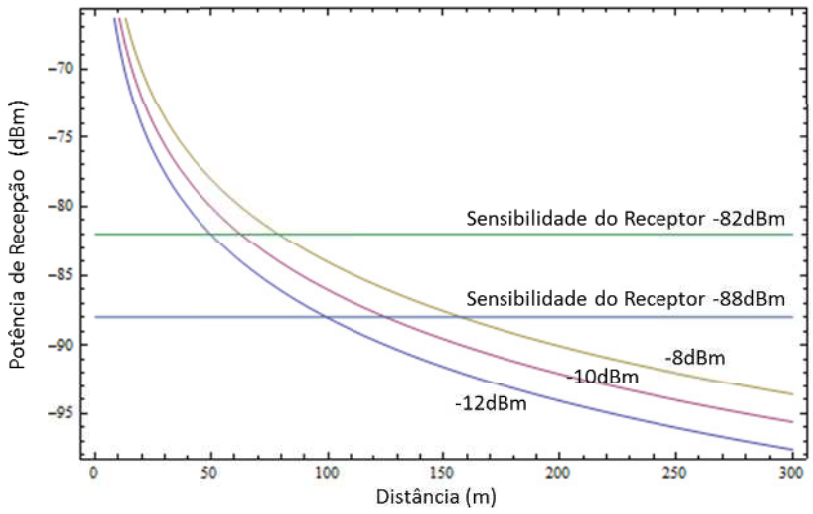


Figura 2.7 – Gráfico potência de recepção em função da distância do transmissor-receptor para o transceptor CC2500. Transmissor com potência de transmissão -8 dBM, -10 dBm e -12 dBm.

Deve-se considerar, além da potência de transmissão, a existência de barreiras (paredes, vegetação etc.) ou sinais de interferência entre nós, os quais podem comprometer a comunicação entre eles.

As barreiras físicas existentes entre os emissores e receptores são fatores atenuante dos sinais transmitidos. A diminuição do sinal depende tanto do tipo do material como da frequência de transmissão. A Tabela 2.6 apresenta alguns materiais e a perda relacionada para uma frequência de 2,4 GHz.

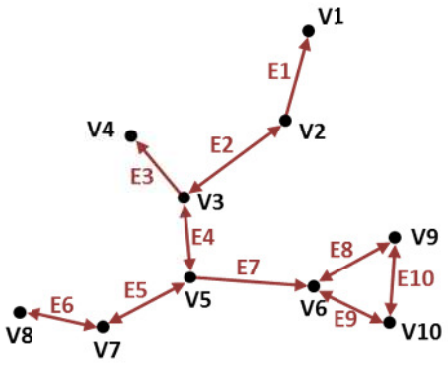
Se a potência de transmissão e a sensibilidade de recepção dos rádios são diferentes para cada nó, o grafo representante da rede torna-se dirigido, isto é, pode existir um nó *Vi(t)* que se comunica com *Vj(t)*, mas *Vj(t)* não se comunica com *Vi(t)*. Esta situação é difícil de ocorrer entre nós sensores em uma WSN, pois geralmente os nós são homogêneos. Contudo, pode ocorre entre as estações rádio base e os nós sensores, uma vez que, de maneira geral, as estações rádio base possuem antenas de maior ganho.

A Figura 2.8 ilustra um grafo dirigido. Os nós V1 e V4 não conseguem se comunicar com os nós V2 e V3, respectivamente, assim como o conjunto de nós V6, V9 e V10 não conseguem transmitir seus dados para o resto da rede uma vez que o nó V6 não consegue realizar transmissão para o nó V5.

Tabela 2.6 – Perda de sinal para diferentes materiais (frequência de 2,4GHz) [CAV07]

Material	Atenuação (dBm)
Banheiro	-25
Parede de tijolo	-6
Parede de gesso	-2
Parede gesso/vidro	-1
Laje	-8

Tanto a abordagem realizada a partir dos nós (entidades reais) como da rede observada como um grafo são úteis no desenvolvimento de protocolos e estudo do funcionamento das WSN. Contudo, a visão real da rede a partir do conhecimento presente em cada nó é que de fato permite que a rede execute suas atividades, sendo os grafos uma ferramenta válida apenas no campo da simulação computacional, visualização global do funcionamento da rede e estudo do desempenho de protocolos.



V(t) = { V1, V2, V3, V4, V5, V6, V7, V8, V9, V10 }

E(t) = { E1, E2, E3, E4, E5, E6, E7, E8, E9, E10 }

E1(t): { V2(t)→ V1(t) }
E2(t): { V2(t) → V3(t); V3(t) → V2(t) }
E3(t): { V3(t) → V4(t) }
E4(t): { V3(t) → V5(t) ; V5(t) → V3(t) }
E5(t): { V5(t) → V7(t); V7(t) → V5(t) }
E6(t): { V7(t) → V8(t); V8(t) → V7(t) }
E7(t): { V5(t) → V6(t) }
E8(t): { V6(t) → V9(t); V9(t) → V6(t) }
E9(t): { V6(t) → V10(t); V10(t) → V6(t) }
E10(t): { V9(t) → V10(t); V10(t) → V9(t) }

Figura 2.8 – Uma WSN representada como um grafo dirigido. Diferente do grafo da Figura 2.6, alguns nós não conseguem se comunicar com os demais devido a orientação das arestas do grafo. O nó V1, por exemplo, não consegue se comunicar com o nó V2.

Uma vantagem de se analisar uma WSN como um grafo em ambiente computacional é a possibilidade de se estabelecer relações de vizinhança de forma mais clara e determinar rotas para as mensagens de maneira otimizada, utilizando peso nas arestas, por exemplo. A relação de vizinhança entre os nós pode ser estabelecida por diferentes critérios, sendo o geométrico e o alcance dos rádios os mais usuais (Figura 2.9).

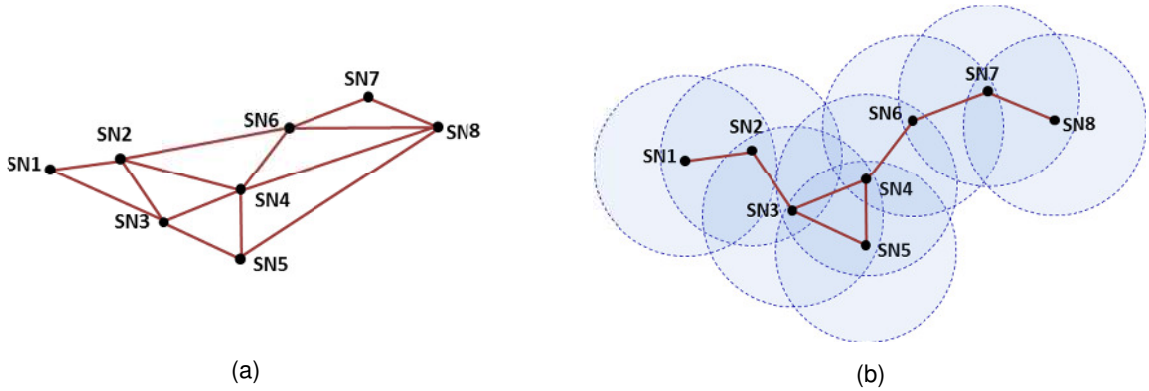


Figura 2.9 – Relação de vizinhança entre os nós de uma WSN estabelecida segundo critério (a) geométrico e (b) alcance dos rádios

Do ponto de vista da rede real, a relação de vizinhança estabelecida pelo critério geométrico não faz sentido, pois, por exemplo, na Figura 2.9 o nó SN4 não consegue se comunicar com o nó SN8. Mas do ponto de vista de análise de grafo, isto é, realizando uma interpretação de relação na posição dos nós, os nós SN4, SN5, SN6, SN7 e SN8 podem estar geograficamente relacionados a uma mesma área de interesse.

A presença de barreiras atenuadoras de sinal faz com que algumas possibilidades de conexão entre os nós sejam desfeitas, alterando as relações de vizinhança entre os nós.

2.5 – Ciclo de vida de uma WSN[8]

Toda Rede de Sensores Sem Fio possui um Ciclo de Vida, não importando a qual aplicação se destina. Este ciclo é composto de três fases: Estabelecimento da Rede (Distribuição dos Nós e Formação da Rede), Operação da Rede (Sensoriamento, Processamento e Comunicação) e Manutenção da Rede (Reação, Prevenção, Correção e Adaptação).

O ciclo de vida de uma WSN tem início pela distribuição dos nós sensores e das estações rádio base de forma planejada ou aleatória na área de interesse. A partir do momento em que os nós encontram-se posicionados no ambiente, inicia-se o processo de formação da rede, no qual são executados algoritmos de roteamento, localização de nós, formação de *clusters*, determinação do número de *hops* necessários para cada nó acessar uma estação rádio base, entre outros. Quais e quantos algoritmos são executados no procedimento de formação da rede depende da aplicação da rede e como seus nós foram distribuídos na área. Uma WSN cujos nós sensores foram inseridos no ambiente de forma planejada, por exemplo,

[8] Vida e morte são palavras utilizadas como catacrese em relação à Rede de Sensores Sem Fio. Vida está relacionado ao período de operação da rede, enquanto morte está relacionado à situação em que a rede para de funcionar.

não necessita que um algoritmo de localização dos nós seja executado.

A Figura 2.10 ilustra as fases de Estabelecimento e Operação de uma Rede de Sensores Sem Fio, sendo que neste exemplo os nós são distribuídos no ambiente de forma aleatória.

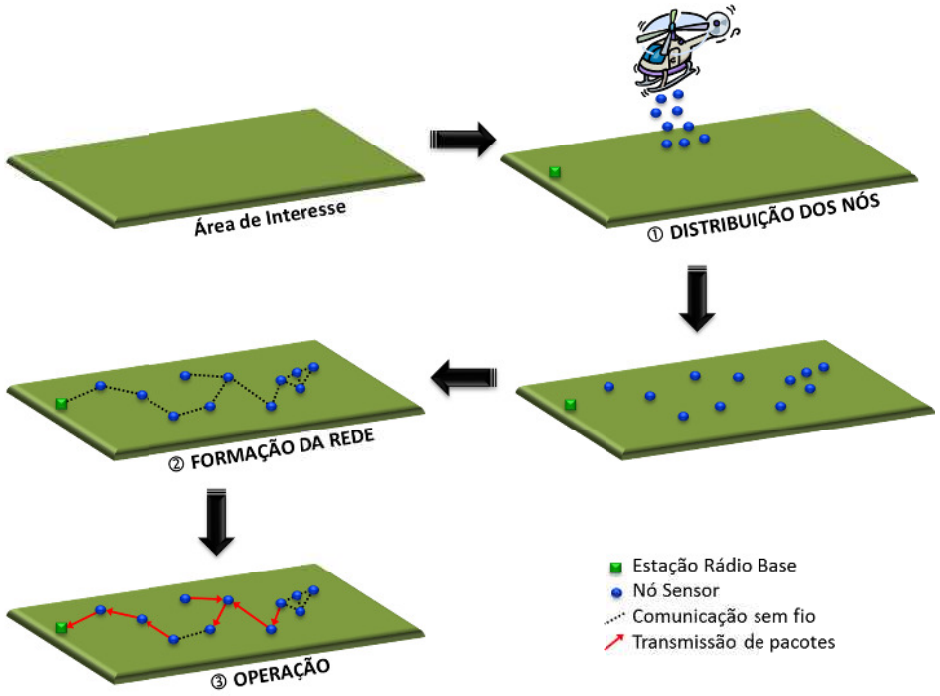


Figura 2.10 – Estabelecimento e Operação de uma Rede de Sensores Sem Fio. A ① DISTRIBUIÇÃO DOS NÓS é realizada na Área de Interesse. Estando os nós posicionados no ambiente, inicia-se a ② FORMAÇÃO DA REDE (estabelecimento de rotas e outros procedimentos que dependem da aplicação da rede). Após o Estabelecimento da WSN, ela entra em ③ OPERAÇÃO. Adaptação de [MAT04].

Após a rede estar estabelecida (distribuição dos nós e formação da rede), a fase de operação tem início. A Rede de Sensores Sem Fio permanece nessa fase a maior parte do tempo, executando tarefas de sensoriamento, processamento e comunicação. A aplicação da rede, os protocolos de comunicação e esquemas de gerenciamento de energia irão reger a operação da rede.

Durante a operação de uma Rede de Sensores Sem Fio há a necessidade de realizar a manutenção da sua estrutura devido a eventos internos (falhas nos módulos de comunicação, perda de energia nos nós sensores etc.) e externos (destruição de nós ou modificação das suas localizações, interferências persistentes em determinadas áreas, inserção de novos nós etc.). As falhas ou modificações resultantes desses eventos podem ser detectadas através de consultas realizadas pelas estações rádio base aos nós sensores ou por tarefas de gerenciamento que são executas de forma distribuída na rede. Contudo, devido à própria natureza distribuída das Redes de Sensores Sem Fio e pelo fato da comunicação entre

os nós da rede ser realizada exclusivamente pelo meio sem fio, falhas em determinados nós da rede ou alterações topológicas severas podem impedir o reestabelecimento da rede de forma simples e rápida.

A manutenção da rede pode ocorrer das seguintes formas: por **adaptação**, adicionando nós ao roteamento quando novos nós são inseridos na rede ou recriando o roteamento com menos nós participantes na medida em que nós sensores saem da rede por falha ou falta de energia; por **prevenção**, alterando as rotas dos pacotes com o objetivo de poupar energia de nós que estão sendo utilizados em demasia nas rotas vigentes, aumentando, assim, o tempo de vida desses nós e permitindo o funcionamento da rede por mais tempo; por **correção**, alterando as estratégias na execução de tarefas, quando detectado que as mesmas estão fornecendo resultados diferentes do esperado; ou por **reação**, a partir do momento que os nós da rede não respondem mais a comandos emitidos pelas estações rádio base, significando que pode ser preciso realizar alguma operação de gerenciamento da rede (envio de comandos para a rede em operação), demandar uma nova formação da rede (recriação de rotas) ou que a rede parou de funcionar.

A Figura 2.11 apresenta o fluxograma do Ciclo de Vida de uma Rede de Sensores Sem Fio.

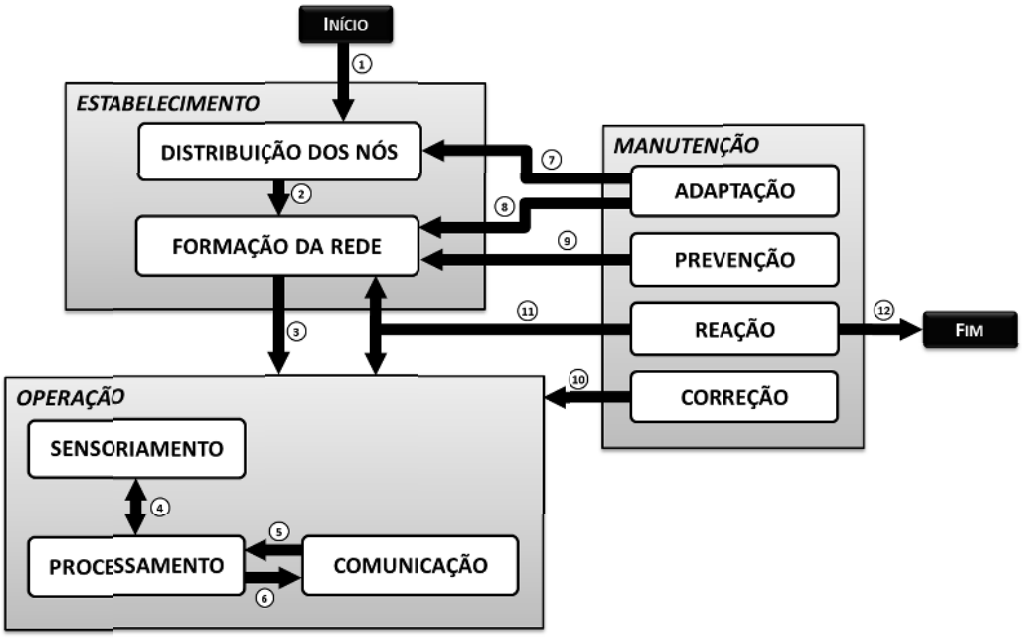


Figura 2.11 – Ciclo de vida de uma Rede de Sensores Sem Fio.

Quando a comunicação entre a sub-rede composta pelos nós sensores cessa definitivamente em relação às estações rádio base, a WSN pode ser considerada como não mais em funcionamento. Alguns nós sensores ainda podem permanecer ativos por um tempo (ainda restando energia em suas baterias que permita o funcionamento), mas certamente existe um ou mais nós sensores que deixaram de funcionar e são essenciais às rotas de pacotes que estabelecem conexão da sub-rede de nós sensores com as estações rádio base.

O tempo de vida de uma Rede de Sensores Sem Fio, isto é, seu tempo operacional, pode ser definido, então, pelo intervalo entre o instante em que a rede foi estabelecida pela primeira vez até o momento em que o último pacote de dados é recebido por uma estação rádio base. A redundância dos dados propagados pela rede permite que ela continue funcionando mesmo diante a falha de alguns nós sensores [KUL99].

Contudo, essa definição de tempo de vida não é consenso na literatura e pode ser falsa para determinadas aplicações. Há casos em que após a distribuição dos nós sensor em uma área é detectado que o fenômeno de interesse concentra-se em uma subárea específica. Supondo que os nós dessa subárea “morram” primeiro, mesmo que os demais nós continuem a enviar dados, a utilidade da rede é questionável. O tempo de vida de uma WSN pode, portanto, ser definida de diversas maneiras, a depender fundamentalmente do contexto da aplicação.

Apesar dessa dependência da aplicação, [DIE09] e [MAK09] apresentam estudos independentes sobre as definições de tempo de vida para WSN que aparecem na literatura e que levam em consideração outros parâmetros. Segue uma coletânea de definições extraída de ambos os trabalhos e de outros artigos consultados.

O tempo de vida de uma WSN pode ser definida a partir do seu estabelecimento até o instante que:

(1) o primeiro nó sensor falha [KAN05][SHI06];

(2) certa quantidade de nós sensores falha [RAI05][VER05][SOL08];

(3) todos os nós sensores fiquem sem energia [HEI00][TIA02];

(4) a rede fica desconectada em algum ponto crucial à comunicação [MAK09];

(5) ainda existe uma quantidade de nós sensores conectados às estações rádio base [BLO02];

(6) a taxa de recebimento de pacotes nas estações rádio base fica abaixo de um limite [DON05];

(7) o número de erros de transmissão ultrapassa um limite [VER05];

(8) o número de pacotes recebidos com erro ultrapassa um limite [CHE05];

(9) é detectada a primeira falha na coleta de dados [CHE05];

(10) o tempo de ocorrência de um evento e a recepção da sua informação ultrapassa um limite [KUM05];

(11) não existem mais rotas centralizadoras de comunicação (*backbones*) [DON05].

(12) até que o 1º *cluster head* deixe de funcionar em redes com formação de clusters [CHI02][SOR05][SOL08];

(13) certa percentagem da área de interesse não se encontra mais coberta com pelo menos um nó sensor [CAR05b][LIU05];

(14) a área de interesse não se encontra mais coberta com certa quantidade de nós sensores [BHA01][BHA02];

(15) uma percentagem de nós não mais possui um caminho de comunicação até uma estação rádio base [CAR06];

(16) a rede não mais provê informações com certa taxa na detecção de eventos [TIA02];

(17) a rede não mais satisfaz os requisitos da aplicação [KUM05].

As definições utilizadas pelos autores empregam os termos “falha”, “morte” e “ficar sem energia” de forma não bem definida com relação aos nós sensores. Para fim de entendimento e no contexto das definições, esses termos são interpretados como “deixar de funcionar”. O termo “falha” (itens (1) e (2)) é o mais crítico para essa generalização, pois poderia ser entendido como um momento no qual o resultado de uma operação não ocorreu de forma correta, mas em havendo uma recuperação do sistema, a operação poderia voltar a ser realizada corretamente. Essa interpretação só faz sentido no contexto do item (9), isto é, uma falha na operação e não no nó sensor.

[DIE09] apresenta, ainda, definições para o tempo de vida de uma WSN formulada a partir de combinações das definições apresentadas, como, por exemplo, área de cobertura da rede em conjunto com conectividade. Tais definições não serão consideradas como particulares, mas sim como composições das demais.

Pelo fato de serem encontradas muitas definições na literatura, [DIE09] e [MAK09] tentam agrupá-las em classes. A Tabela 2.7 e a Tabela 2.8 apresentam essas classificações, relacionando-as com as definições apresentadas. Nem todas essas definições se enquadram nas classificações dos autores.

Apesar das classificações apresentadas na Tabela 2.7 e a na Tabela 2.8 serem coerentes do ponto de vista teórico, a Tabela 2.9 apresenta uma classificação formulada tendo em vista o conhecimento da rede. Algumas definições só fazem sentido a partir do momento em

que existe um conhecimento total da rede (como um grafo), como ocorre nos processos de simulação. A definição (3), por exemplo, que especifica que o tempo de vida de uma WSN se encerra quando todos os nós da rede estejam sem energia, só faz sentido quando se tem o conhecimento global da rede. Do ponto de vista do conhecimento operacional (dados que chegam às estações rádio base), não se pode afirmar que todos os nós estejam sem energia, uma vez que alguns nós podem estar operando normalmente, mas não exista uma rota que permita a entrega de pacotes para uma estação rádio base.

Tabela 2.7 – Classificação das definições do tempo de vida das WSN [DIE09]

Classificação	Definições
Quantidade de nós sensores que falham	(1) (2) (3)
Ocorrência da primeira partição na rede	(4) (5) (11)
Taxa de recebimento de pacotes	(6) (8) (16) (17)

Tabela 2.8 – Classificação das definições do tempo de vida das WSN [MAK09]

Classificação	Definições
Quantidade de nós sensores operantes	(1) (2) (3)
Área de cobertura da rede	(13) (14)
Conectividade da rede	(4) (5) (11) (15)
Qualidade de Serviço (QoS)	(6) (7) (8) (9) (10) (16) (17)

Tabela 2.9 – Classificação das definições do tempo de vida segundo o conhecimento da rede

Classificação	Definições
Conhecimento Operacional	(5) (6) (8) (9) (17)
Conhecimento Total	(1) (2) (3) (4) (7) (10) (11) (13) (14) (15) (16)

Considerações

Apesar de todas as definições de tempo de vida de uma WSN ter o seu mérito nos estudos em que foi utilizada, a 17ª definição (a rede não mais satisfaz os requisitos da aplicação) é a mais abrangente e a que teria uma melhor coerência aos estudos das Redes de Sensores Sem Fio, uma vez que a natureza e a programação dessas redes também são dependentes da aplicação a que destinam.

A definição adotada pelos pesquisadores tem impacto direto no estudo da eficiência

dos protocolos de comunicação em WSN, mas muitas vezes essa importância é negligenciada nas publicações. Tanto para fins individuais como para estudos comparativos, o critério tempo de vida adotado muitas vezes mascara os resultados, podendo levar a falsas interpretações nos valores obtidos para o tempo de vida das WSN.

Uma questão ainda pouco abordada nos artigos sobre WSN é o que acontece quando uma rede deixa de funcionar. Poder-se-ia simplesmente abandonar os nós sensores caso a rede não seja mais requisitada pela sua aplicação. Novos nós sensores podem ser inseridos na área para que uma nova rede seja formada, utilizando ou não os nós sensores ainda em funcionamento da rede antiga. Ou novos nós sensores poderiam ser inseridos apenas nas áreas onde houve desconexões de rede, se estas áreas forem conhecidas à priori. Não existe uma resposta única para todos os casos, pois, mais uma vez, a aplicação da WSN é que determinará o que será feito com a rede que para de funcionar.

2.6 – Consumo e gerenciamento de energia em WSN

Em uma Rede de Sensores Sem Fio, os nós sensores possuem quantidades limitadas de energia em suas baterias e, na maior parte das aplicações, essa energia é insubstituível. A utilização da energia de forma adequada é fator primordial para garantir um maior tempo operacional à rede.

As estações rádio base, por possuírem maior quantidade de energia que os nós sensores e estarem localizadas estrategicamente (permitindo a substituição das baterias), não possuem influência crítica no tempo de vida das WSN. Apesar disso, por não realizarem a tarefa de sensoriamento, mesmo que elas continuem a operar por mais tempo que os nós sensores, as estações rádio base, sozinhas, não são capazes de gerar nenhum dado útil do ponto de vista da aplicação.

O cômputo do consumo de energia em uma Rede de Sensores Sem Fio está associado às operações realizadas nos nós sensores que, de maneira geral, podem ser divididas em três estágios: **sensoriamento**, **processamento** e **comunicação**.

O estágio de sensoriamento consiste na conversão de um sinal físico/químico qualquer para um sinal elétrico (transdutor), condicionamento desse sinal e conversão analógico-digital. Tal estágio varia de rede para rede, conforme a natureza do sinal e a tecnologia utilizada no sensor presente nos nós, não podendo ser quantizada sem levar em consideração o hardware empregado.

Os sensores podem medir propriedades físicas (pressão, temperatura, umidade, fluxo), propriedades relacionadas ao movimento (posição, velocidade, aceleração),

propriedades de contato (força, tensão, torque, vibração), presença (toque, proximidade, distância, movimento), propriedades químicas (presença de gases e elementos dissolvidos em líquidos), propriedades bioquímicas (agentes biológicos) ou identificação (imagens). Cada tipo de sensor possui uma determinada tecnologia, característica de funcionamento e um gasto energético específico associado.

O estágio de processamento (ou de computação) representa a “inteligência” do nó sensor, envolvendo o microprocessador e as unidades de armazenamento (memórias voláteis e não voláteis). Nesse estágio o gasto energético está associado tanto ao componente de hardware empregado como ao software de gerenciamento e processamento do nó sensor.

O hardware do processador deve fornecer estados de baixa atividade (SLEEP, por exemplo), diminuindo o consumo de energia. Quanto maior a frequência do processador, maior o consumo de energia [SIL04]. O software, por sua vez, deve ser desenvolvido para realizar o mínimo de operações necessárias e acionar os outros dispositivos a menor quantidade de vezes possível e por tempo reduzido.

O estágio de comunicação é o que utiliza a maior quantidade de energia do nó [BHA06][SOH00]. O rádio (hardware) que estabelece a comunicação entre os nós deve possuir tecnologia de baixo consumo e o seu acionamento deve ser ponderado pelo software do sistema. Por isso, estratégias de comunicação eficientes devem ser estabelecidas entre os nós e entre os nós e a estação rádio base para que o gasto energético seja minimizado. A redução no número de transmissões e recepções realizadas, portanto, aumenta o tempo de vida da rede.

Uma das alternativas para reduzir o consumo energético dos nós sensores é alterar periodicamente o estado dos nós, deixando-os a maior parte do tempo em estados de baixo consumo energético. A Tabela 2.10 apresenta, como exemplo, alguns estados globais para nós sensores levando em consideração os estados dos componentes: microcontrolador, memória, sensores e conversores A/D e do rádio.

O estado TRANSMITING (TRANSMITINDO) e RECEIVING (RECEBENDO) são os que utilizam maior quantidade de energia, pois todos os componentes do sistema estão ativos para transmitir ou receber mensagens, respectivamente. O estado READY (PRONTO) é o estado do nó que permite a leitura dos dados dos sensores (e/ou conversores A/D), além de ser um estado que antecede a transmissão ou recepção de pacotes. O estado OBSERVING (OBSERVANDO) é o estado de escuta do meio, isto é, o rádio e os sensores são os únicos elementos ligados, estando o primeiro no estado de recepção de mensagens (RX). O estado STANDBY (PRONTIDÃO) é aquele reservado exclusivamente para sensoriamento, no qual todos os outros componentes estão dormindo ou desligados. O estado SLEEP (DORMINDO) é

o de baixo consumo, no qual o rádio e os sensores estão desligados e os componentes de controle (microcontrolador e memória) permanecem apenas com alguns módulos internos em execução e, geralmente, se reduz a frequência de execução de suas instruções. O último estado é o OFF (DESLIGADO), no qual nenhum dos componentes está em funcionamento.

Alguns rádios, como o CC2500 [CC_09], por exemplo, possuem os modos de operação IDLE e SLEEP, permitindo a formação de outros estados globais para o nó, mais pertinentes que o estado OFF em algumas situações em que se deseja reduzir o consumo de energia, mas sem prejudicar a escuta do meio.

Tabela 2.10 – Estados dos nós sensores e dos seus componentes [WAN04] [9]

Estado do Nó	Microcontrolador	Memória	Sensores e Conversores A/D	Rádio
TRANSMITING	ACTIVE	ACTIVE	ON	TX
RECEIVING	ACTIVE	ACTIVE	ON	RX
READY	IDLE	SLEEP	ON	RX
OBSERVING	SLEEP	SLEEP	ON	RX
STANDBY	SLEEP	SLEEP	ON	OFF
SLEEP	SLEEP	SLEEP	OFF	OFF
OFF	OFF	OFF	OFF	OFF

A Figura 2.12 apresenta um exemplo de transição de estados dos nós sensores de acordo com os estados da Tabela 2.10. Ressaltar-se que a transição entre os estados também possui um gasto de energia e um tempo de latência associados, que devem ser considerados para não haver gastos excessivos pela simples troca de estado. O tempo em que o nó fica em determinado estado, além do seu consumo em cada estado, é, portanto, uma variável importante para se determinar se haverá de fato uma economia energética na mudança de estado do nó.

A periodicidade do sensoriamento realizado na rede também tem influência direta no consumo de energia da mesma. Redes com coleta de dados contínua utilizam mais energia

[9] Os estados ACTIVE, IDLE, SLEEP e OFF, além de outros, podem ser definidos pela quantidade de módulos eletrônicos que estão ativos no componente, refletindo, assim, no consumo de energia total do componente. A definição de cada estado varia de acordo com o componente, mas de maneira geral, pode-se entender cada estado como se segue:

ACTIVE (ATIVO): Maior gasto de energia com todos os módulos do componente ligados, utilizando toda a potência requerida pelo componente;
IDLE (INATIVO): Estado de baixo consumo energético, permanecendo alguns módulos de processamento e memória ativos;
SLEEP (DORMINDO): Estado de menor consumo energético, permanecendo apenas os módulos de controle do componente ligados;
OFF (DESLIGADO): Componente completamente desligado.

que redes cuja política de sensoriamento é periódica ou reativa.

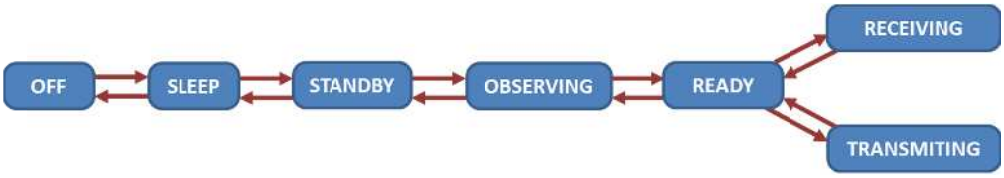


Figura 2.12 – Transição entre estados dos nós sensores. Adaptado de [WAN04].

Alguns Sistemas Operacionais para Redes de Sensores Sem Fio como o TinyOS [TIN06], SOS [HAN05] e MANTIS [BHA05] já incluem módulos de gerenciamento de energia, alterando o estado dos seus componentes de acordo com a necessidade de suas utilizações.

Devido ao procedimento de roteamento de pacotes partindo dos nós sensores em direção às estações rádio base, a carga da bateria de alguns nós sensores terminam antes que a de outros. A Figura 2.13 ilustra uma rede com doze nós sensores e uma estação rádio base, indicando a carga de roteamento de cada nó, isto é, a quantidade de nós sensores que dependem daquele nó para que seus pacotes sejam encaminhados para uma estação rádio base.

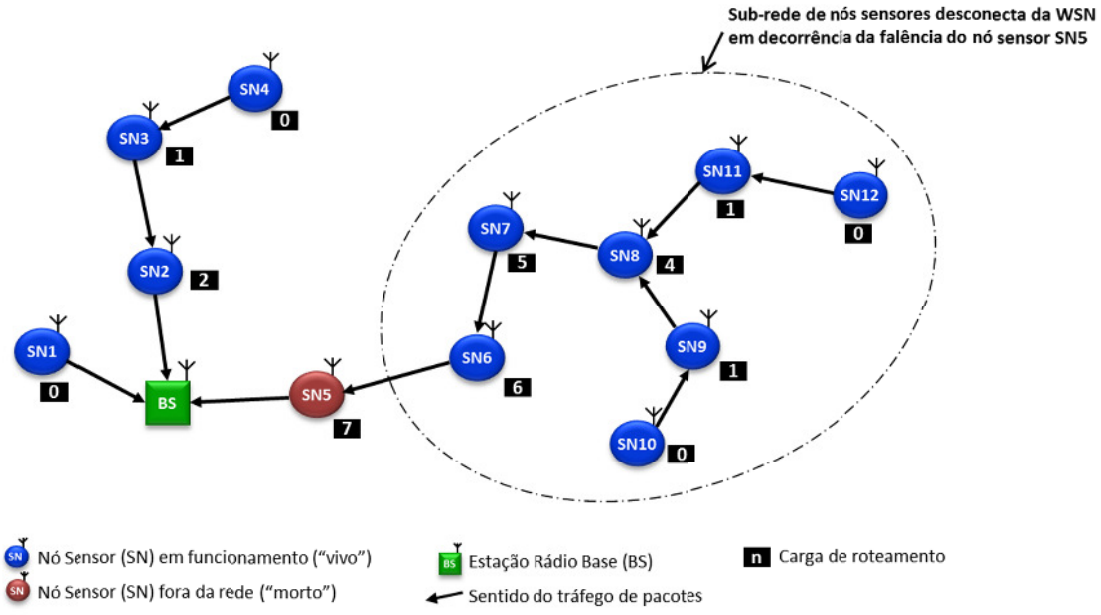


Figura 2.13 – Carga de roteamento nos nós e desconexão de sub-rede em decorrência da falência do nó sensor SN5.

Como o nó sensor SN5 possui a maior carga de roteamento da rede, certamente ele será o primeiro a "morrer", pois toda vez que um dos nós da sub-rede assinalada na Figura 2.13 transmitir um pacote, ele certamente passará pelo nó SN5. Assim, a carga da bateria dos nós

mais próximos às estações rádio base tendem a terminar primeiro e de forma progressiva pelos nós até aqueles que se encontram em posições mais distantes das estações rádio base.

A forma proposta pelos autores para se contornar esse problema, conhecido como Buraco de Energia (E*nergy Hole*), é a utilização de nós sensores com cargas de bateria heterogêneas, posicionando nós com maior quantidade de energia mais próximos às estações rádio base. Os trabalhos [SIC05] e [WU_06], com proposta de melhoramento no trabalho de mestrado [MIR11], apresentam alguns arranjos de nós com cargas de baterias diferentes.

Comumente Mapas de Energia, como o da Figura 2.14, são utilizados para a análise da eficiência da rede e muitas vezes considerados para auxiliar no prolongamento do seu tempo de vida. A permanência de certa homogeneidade de distribuição energética no Mapa de Energia de uma rede não garante que ela tenha um tempo de vida longo, mas indica que a distribuição energética ou uso energético está equilibrado. Tal homogeneidade no Mapa de Energia ao longo da "vida" de uma WSN permite dizer que, ao seu término operacional, pouca energia foi perdida nos nós que restaram desconectados da rede.

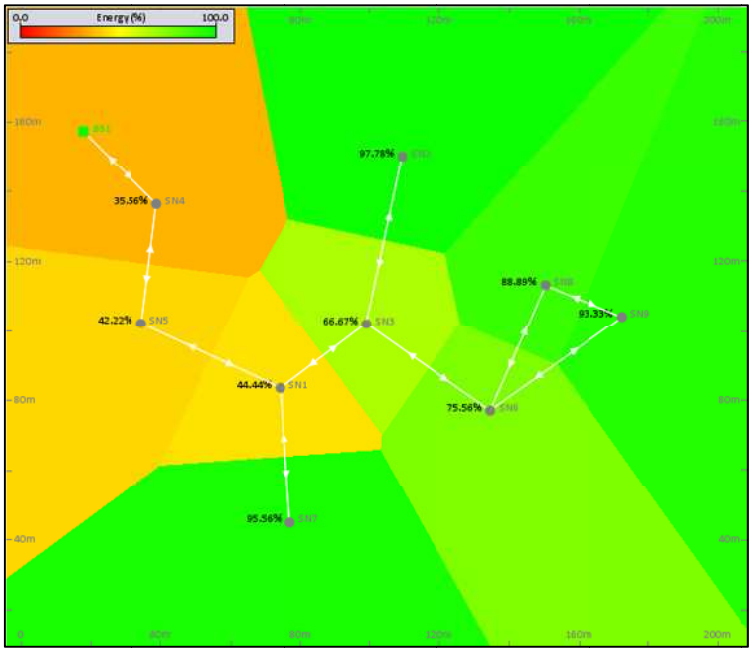


Figura 2.14 – Exemplo do mapa de energia de uma Rede de Sensores Sem Fio. O valor máximo (100%) corresponde a energia máxima (inicial) de cada nó.

Do ponto de vista de conhecimento da rede, o Mapa de Energia é uma outra abstração da rede, como no caso do grafo. O conhecimento desse mapa é pertinente apenas em simulações computacionais e, possivelmente, poderia ser de conhecimento das estações rádio base.

Considerações

A partir da análise dos elementos envolvidos no consumo energético das Redes de Sensores Sem Fio, pode-se constatar que a busca pelo aumento do tempo de vida da rede passa tanto por melhorias nos elementos de hardware como nos elementos de software. Com relação ao hardware, tecnologias micro e nano-eletrônicas de menor consumo devem emergir; com relação ao software, protocolos devem ser elaborados de forma a minimizar a quantidade de acionamento dos rádios.

Do ponto de vista de funcionamento global da rede, existe uma dualidade entre economia de energia e disseminação de pacotes. Se por um lado se deseja evitar o acionamento dos rádios devido ao gasto de energia associado, por outro, e como objetivo principal da existência da rede, deseja-se que os pacotes contendo as informações dos sensores sejam recebidos pelas estações rádio base. Um equilíbrio no número de vezes que um nó aciona seu rádio e a escolha adequada de quais nós estão envolvidos em uma comunicação pode minimizar o consumo de energia da rede, maximizando a sua vida útil.

Segundo [SIL04], a comunicação *multihop* é uma das principais formas de economia de energia em uma WSN, visto que a energia requerida para a comunicação entre dois nós arbitrários é dependente da distância entre os nós (Equação 2.1 e Equação 2.2). O ajuste da potência de transmissão e sensibilidade dos rádios dos nós sensores de forma a permitir o estabelecimento de uma conexão *multihop* eficiente entre os nós pode diminuir o *overhearing*, aumentando o tempo de vida da rede. A densidade da rede está diretamente relacionada ao critério utilizado nesse possível ajuste, visto que quanto maior a densidade da rede, maior a probabilidade da ocorrência de *overhearing*, além de gerar uma quantidade maior de colisões que resulta em necessidade de retransmissão.

Apesar dessa avaliação benéfica da política *multihop*, ela faz com que a maior parte dos pacotes que um nó sensor receba não seja destinado para ele, necessitando um gasto extra para processar o pacote e retransmiti-lo para outros nós [BHA06]. Deve-se levar em conta, que os gastos energéticos dos rádios nas operações de recebimento e envio de pacotes geralmente não são iguais [COL03]. O rádio TR1000 [RFM99], por exemplo, consome 12 mA para transmitir e 4,5 mA para receber pacotes (valores médios).

Com relação ao gasto extra de processamento necessário nos nós para identificar a necessidade de repassar o pacote, um sistema inteligente que auxilie o rádio a identificar e redirecionar os pacotes sem a necessidade de processá-los nos nós sensores intermediários diminuiria a quantidade de energia e o tempo gasto no roteamento do pacote. [LI_01] sugere que a comunicação entre os nós deve ocorrer sem a intervenção do sistema operacional, não

havendo, portanto, necessidade do microcontrolador se envolver nesse processo, permanecendo em estado de menor consumo energético. O rádio deve ser dotado de certa capacidade de processamento para tomar as decisões pertinentes para esse tipo de retransmissão.

Por outro lado, o processamento de dados localmente nos nós sensores é uma forma de reduzir o trafego de informações e economizar energia [ASA98]. A compactação e a fusão de dados pode reduzir o tempo com que o rádio do nó sensor permanece enviando e recebendo os pacotes. [CIA06] ressalta que reduzindo a quantidade de informações que necessitam ser transmitida pela rede de sensores sem fio pode-se conseguir uma grande redução no gasto energético. Para isso, sugere a utilização de algoritmos de compressão de dados nos nós sensores e escolha cuidadosa na forma de representação dos dados.

Outra análise realizada por [CIA06] é que na busca de um melhor aproveitamento energético, nem sempre a escolha do menor caminho (geométrico ou que envolva menor quantidade de nós) para rotear um pacote é a que leva à maior economia: há dependências tanto topológicas como funcionais da rede que devem ser levadas em consideração. Nós sensores que disponham de pouca energia devem ser evitados como nós intermediários no processo de roteamento.

A menor rota, portanto, nem sempre é a melhor para garantir o tempo de vida da rede. O problema é identificar qual a melhor rota (energética) a cada instante, pois o seu estabelecimento está associado a uma informação global: maximizar o tempo operacional da rede e não apenas minimizar localmente o gasto energético de cada nó.

Em relação ao modelo de execução dos sistemas operacionais, conforme indica [LI_01], um sistema dirigido por eventos chega a economizar até 12 vezes mais energia que um sistema tradicional e de propósito geral. Nessa mesma linha de raciocínio, [DUL02] salienta que sistemas operacionais que se baseiam em eventos podem ser estruturados em módulos, que são executados em respostas a eventos externos e, quando não mais existem eventos para serem tratados, o sistema pode entrar em modo de economia de energia, mas, para isso, é preciso que o código tenha sido escrito de forma não bloqueante. Não se deve utilizar, portanto, laços (*loops*) para ler sinais de entrada na espera do valor; deve-se utilizar interrupções que “acordarão” o sistema do modo de economia de energia (SLEEP).

Outra questão importante, abordada em [CHI01], refere-se ao comportamento das baterias durante a sua utilização. Em muitos casos, as baterias podem ter um aproveitamento melhor alternando-se períodos de uso e inatividade. Isto se deve porque durante os períodos de inatividade, a bateria recupera parte da energia. Portanto, não se pode considerar o uso de baterias como uma simples soma de eventos. Isso tem um impacto direto na análise de

eficiência energética em protocolos de comunicação que utilizam ciclos de trabalho (*duty cicle*) como estratégia de economia de energia.

Com relação ao desempenho dos protocolos de comunicação, apesar de todos os esforços para desenvolvê-los de forma a maximizar a vida operacional das Redes de Sensores Sem Fio, existe um fator que influencia o desempenho de qualquer protocolo e que apresenta um elevado grau de complexidade para ser contornado: a topologia da rede.

A distribuição dos nós de grande parte das WSN é realizada de maneira aleatória, não se sabendo de antemão a posição dos nós no ambiente a ser monitorado e a topologia de rede que poderá ser estabelecida diante a posição relativa desses nós sensores. A eficiência energética de um protocolo de comunicação depende diretamente da topologia estabelecida pela rede e dos parâmetros de comunicação dos rádios [HEI00]. Portanto, um protocolo só poderá ser considerado mais eficiente que outro se forem realizados testes em diferentes topologias, o que geralmente não é constatado na maioria dos artigos publicados.

2.7 – Aplicações das Redes de Sensores Sem Fio

As aplicações das Redes de Sensores Sem Fio são diversas e a cada dia surgem novas potencialidades para esse tipo de rede. Desde aplicações voltadas para o dia-a-dia das pessoas nas cidades (controle de tráfego de veículos nas ruas, monitoramento da qualidade do ar, segurança de patrimônios, prevenção de rupturas em estruturas da construção civil, etc.) até o monitoramento e a pesquisa de áreas naturais de difícil acesso e observação (temperatura dos oceanos, migração de animais, predição de avalanches e abalos sísmicos, etc.).

Apesar dos exemplos de aplicação das Redes de Sensores Sem Fio apresentados na literatura muitas vezes não corresponderem à configuração de WSN como é definida neste trabalho, os autores que ilustram seus trabalhos ou utilizam as redes sem fio em aplicações específicas as classificam como WSN.

Há diversas formas de agrupar os exemplos de aplicações das WSN, não havendo um consenso na literatura. Optou-se pelos seguintes agrupamentos, levando em conta suas similaridades: Agropecuária e Monitoramento Ambiental, Engenharia Civil e Ambiente Urbano, Indústria, Medicina e Monitoramento de Saúde e Aplicações Militares e Aeronáutica.

2.7.1 – Agropecuária e monitoramento ambiental

A agricultura de precisão, caracterizada pela utilização da tecnologia da informação associada à agricultura industrial moderna [DAV98], envolve o planejamento, cultivo, gerenciamento e a colheita de gêneros alimentícios, tendo como principais objetivos a redução

nos custos e o aumento na produtividade agrícola. As WSN podem ser utilizadas como fornecedoras de dados para a tomada de decisões que dependam das variáveis ambientais envolvidas no processo produtivo agrícola, tais como: temperatura, umidade e alterações químicas do solo [KAL11]. Os nós sensores são espalhados pela área de cultivo e periodicamente enviam informações sobre as variáveis de monitoramento. Assim, pode-se economizar água na irrigação e reduzir a quantidade de produtos químicos utilizados no controle das culturas.

O monitoramento de animais também é uma área de grande utilidade das WSN. O controle do peso dos animais pode ser realizado à distância pela utilização de nós sensores acoplados às patas dos animais, evitando assim o estresse a que são submetidos durante a pesagem convencional [NAD07].

A aplicação das Redes de Sensores Sem Fio no monitoramento ambiental vem se ampliando e diversificando no decorrer dos anos, principalmente pela intensificação da atividade antrópica em áreas de preservação ambiental e a criação de leis ambientais mais rígidas. Em casos particulares de pesquisas em ambientes naturais de acesso restrito ou que a presença humana gera interferência no objeto pesquisado, ocorre a impossibilidade de se obterem dados periódicos, sendo uma alternativa factível a aplicação das WSN nesses ambientes [MAI02].

O monitoramento de áreas de risco, principalmente movimentação de massa no caso do Brasil (rastejo, escorregamentos, corridas de lama, desprendimento de rocha, etc.), pode se beneficiar de uma rede de nós sensores instaladas nos locais de risco para se realizar o monitoramento periódico de pequenas alterações em blocos de rocha, saprolito e solo que anunciam os eventos de movimentação de massa. Um sistema de monitoramento remoto de fluxo de detritos (*debris flow*) na China utilizando Redes de Sensores Sem Fio pode ser analisado em [CHI12].

As Redes de Sensores Sem Fio também podem ser utilizadas no monitoramento de áreas contaminadas com produtos químicos ou passíveis de contaminação, cujo acesso geralmente apresenta risco à presença humana. Em particular, áreas de minas descomissionadas são um grande problema ambiental pela falta de monitoramento dos estéreis gerados e acumulados no local pela atividade mineradora.

A geofísica é outra área que pode se beneficiar do uso das Redes de Sensores Sem Fio [ODE08]. Desde o monitoramento constante de áreas com atividade geológica proeminente, sendo exemplos o registro das tensões existentes nas falhas tectônicas (que podem provocar abalos sísmicos de grandes proporções) e a composição dos voláteis exalados em vulcões (caracterizando a possibilidade de erupções), até o registro do impacto sísmico no corpo

rochoso pela mineração realizada através do desmonte por explosivos.

2.7.2 – Engenharia civil e ambiente urbano

A conservação de estruturas de engenharia civil requer periódicas avaliações do estado de conservação dos materiais que as compõem. O procedimento de vistoria geralmente é custoso, envolvendo equipes especializadas e muitas vezes requer que os trabalhadores fiquem sujeitos a situações de risco. As Redes de Sensores Sem Fio podem ser utilizadas como ferramenta de monitoramento e diagnóstico para a execução de reparos em obras da construção civil. Nós sensores podem ser instalados em posições críticas de pontes, por exemplo, para o monitoramento da vibração e tensão de suas estruturas.

Com o crescimento das cidades, variáveis que registram dados do microclima e a qualidade do ar nas diferentes regiões urbanas permitem a criação de um banco de dados capaz de auxiliar no planejamento urbano. A concentração de CO_2 em determinada região, por exemplo, pode acarretar propostas de mudanças na configuração das vias de trânsito, deslocando o fluxo de automóveis para regiões menos críticas.

O setor de automação e segurança residencial também é visto como de grande potencial para o desenvolvimento de aplicações que utilizem as WSN. A interação entre as diferentes redes de comunicação é de fundamental importância nesse tipo de aplicação das WSN, pois, na maior parte dos casos, o usuário é o consumidor final, já habituado com as interfaces de comunicação disponíveis em dispositivos que utilizam as redes WLAN e de telefonia celular, por exemplo. Já em ambientes de grande fluxo de pessoas, como prédios e estações de metrô, o controle de calor, ventilação e condicionamento de ar pode ser mais eficiente se utilizar as WSN [CAL03], garantindo uma melhor qualidade de controle climático e redução no consumo de energia. Tanto em aplicações residenciais como prediais, o fato dos nós sensores serem sem fio permite uma rápida instalação do sistema e a baixo custo.

Em [DAV12] as Redes de Sensores Sem Fio são aplicadas no auxílio à segurança de aeroportos. O principal objetivo desse projeto é a detecção de invasores no perímetro dos aeroportos utilizando nós sensores dotados de acelerômetros.

O tráfego de veículos em avenidas e rodovias pode se beneficiar das Redes de Sensores Sem Fio, tanto no que diz respeito ao planejamento de engenharia de tráfego, como tarifação por deslocamento. A disponibilidade de vagas em estacionamentos de prédios públicos e shopping centers também pode ser monitorada por WSN, auxiliando no controle de disponibilidade de vagas e até mesmo conduzindo o motorista para uma vaga disponível [RAJ03a].

Ainda em ambiente urbano, as Redes de Sensores Sem Fio podem ser utilizadas na ocorrência de desastres (nevascas, tsunamis, enchentes, etc.) como rede colaborativa de dados que indiquem a localização dos desastres [SAH07]. No combate a incêndios urbanos, uma WSN pode ser estabelecida com a distribuição por helicóptero de nós sensores resistentes à temperatura na área de ocorrência tanto para auxiliar no combate ao fogo como na localização de vítimas.

2.7.3 – Indústria

A utilização de Redes de Sensores Sem Fio no ambiente industrial tem sido uma alternativa viável para a redução no custo de instalação de dispositivos de sensoriamento. O custo da implantação de redes cabeadas e a necessidade de mudanças constantes na estrutura do chão de fábrica fazem com que as redes sem fio seja a alternativa preferencial no ambiente industrial.

Geralmente nas WSN com aplicações industriais o fluxo de dados é baixo, mas a confiança na rede deve ser alta. Uma rede com muitos nós que permita o estabelecimento de diversas rotas para as mensagens pode ser útil nesse tipo de aplicação [CAL03].

Outro setor industrial que se beneficia das Redes de Sensores Sem Fio são as cadeias de suprimentos (*supply chain*). As informações dos produtos necessárias ao bom funcionamento dessas cadeias podem ser obtidas com nós sensores acoplados aos produtos. Assim, a utilização de um alimento na indústria, por exemplo, pode ser determinada simplesmente pela sua data de validade, mas a utilização de sensores que registram condições específicas dos produtos (temperatura, umidade, ocorrência de descongelamento, etc.) e transmitem essa informação para a cadeia de suprimentos pode garantir um produto final com um padrão de qualidade melhor.

2.7.4 – Medicina e monitoramento de saúde

A aplicação das WSN na área médica difere da telemetria médica, uma vez que esta está voltada para aplicações críticas. Pelo fato das Redes de Sensores Sem Fio serem um campo de pesquisa ainda recente e as plataforma desenvolvidas pouco testadas, a sua aplicação para a manutenção da vida ou o monitoramento crítico de pacientes não é recomendada.

Apesar disso, o monitoramento não crítico de saúde é uma das áreas mais promissoras de aplicação das Redes de Sensores Sem Fio. Ela está relacionada ao registro de informações de saúde não vitais, como o acompanhamento do desempenho de atletas durante

o treinamento e o controle do peso, quantidade de açúcar no sangue e outras substâncias do corpo humano que se apresentam em desordem em pacientes e precisam ser monitoradas.

Em [GAN13] o protótipo de um sistema para o monitoramento cardiovascular e de atividade física é descrito, sendo constituído por cinco sensores fisiológicos: electrocardiograma, pressão sanguínea, oxímetro, temperatura, acelerômetro e giroscópio, estes últimos responsáveis pela registro do movimento corporal.

2.7.5 – Aplicações militares e aeronáutica

As Redes de Sensores Sem Fio tem sido utilizadas em aplicações militares desde a década de 50 com a implantação de uma rede de sensores acústicos na base oceânica do Atlântico e Pacífico [YU_06]. Apesar dessas redes durarem poucas semanas, sua aplicação foi extremamente importante no mapeamento do fundo oceânico, essencial para a navegação submarina.

O monitoramento de fronteiras, a vigilância de campos de batalhas, a localização de invasores, a detecção de armas químicas e biológicas são alguns exemplos de aplicações militares que se beneficiam da utilização das Redes de Sensores Sem Fio [WAN04].

No setor aeronáutico, as WSN têm sido testadas como instrumento auxiliar para manobras e a preservação da estrutura de aeronaves. [WIL08] explora a utilização das Redes de Sensores Sem Fio em aeronaves como forma de eliminação de fios utilizados na comunicação de sensores acoplados nas estruturas das aeronaves, reduzindo, assim, a massa total do sistema. Contudo, a necessidade da troca de baterias nos sensores ainda é algo questionável para esse tipo de aplicação, uma vez que os nós sensores, na maior parte das vezes, são colocados em locais de difícil acesso.

2.8 – Rede de Sensores Sem Fio e a Internet das Coisas

A Internet das Coisas (IoT – *Internet of Things*) pode ser definida como um conjuntos de “coisas” ou “objetos” que possuem um endereço único e estão de alguma forma conectados à Internet. Essas “coisas” podem ser telefones móveis, sensores, atuadores, máquinas, entre outros [GIU09].

Este conceito expande a ideia de conectividade na Internet, na qual não somente as pessoas e ideias estão recebendo e fornecendo informações, mas os objetos do dia a dia também entram nesse cenário compartilhado.

As Redes de Sensores Sem Fio apresentam um papel importante nesse novo paradigma da Internet das Coisas. O benefício da interligação entre as WSN e outros elementos

da IoT vai além da conexão remota, uma vez que ambos os sistemas podem trabalhar de forma colaborativa e prover serviços [ALC10].

Um problema relacionado ao endereçamento específico de cada nó de uma WSN provém da própria natureza da rede. O tipo de protocolo utilizado para endereçamento e empacotamento dos dados, comumente o IPv6 (*Internet Protocol* versão 6), pode ser muito grande para ser utilizado pelos nós sensores, utilizando uma quantidade de energia adicional para processar os pacotes e transmiti-los. Uma alternativa seria utilizar o endereçamento 6LoWPAN (IPv6 aplicado a Redes de Área Pessoal de Baixo Consumo – *Low Power Wireless Personal Area Networks*) que permite uma segmentação da rede, fazendo com que os endereços atribuído aos nós possuam menos bytes que o IPv6 e o tráfego da rede seja limitado à determinadas regiões da rede.

Contudo, cada nó sensor não necessariamente teria um identificador único nesse cenário de conexão global, pois o conhecimento de uma única informação gerada por toda a Rede de Sensores Sem Fio, ou por parte dela, pode ser suficiente para a aplicação.

2.9 – Desafios em Redes de Sensores Sem Fio

Conforme foi explorado nos itens anteriores, o grande desafio no desenvolvimento das Redes de Sensores Sem Fio é garantir a permanência da sua atividade pelo tempo necessário à sua aplicação diante a restrição energética dos nós sensores.

Como o custo da transmissão e recepção de pacotes são os eventos que consomem maior quantidade de energia dos nós sensores, protocolos que consigam ponderar de forma eficiente o número de vezes que o rádio é acionado, sem haver perda de mensagens, evitando ao máximo a ocorrência de colisões e o *overhearing*, é um desafio que pode prolongar o tempo operacional da rede.

A auto-adaptação da rede em decorrência das mudanças de tráfego de pacotes, estado dos nós ou perdas de conectividade também é um desafio, uma vez que a troca de dados necessárias para ser realizar mudanças no roteamento pode consumir uma quantidade de energia dos nós sensores maior que a própria permanência da rede em um estado não adaptado às mudanças.

Com relação ao desenvolvimento de protocolos, o aumento nas suas complexidades pode permitir a utilização das redes em aplicações mais variadas sem haver perdas de eficiência [WAN04]. Contudo, o gerenciamento integrado dessas redes não é tarefa trivial [RUI04b]. O aumento da complexidade não deve, portanto, ser um fator que dificulte o gerenciamento da rede.

A dualidade existente entre nós sensores de propósito geral e economia de energia é outro grande desafio em Redes de Sensores Sem Fio. Se por um lado a utilização de nós sensores específicos para uma determinada aplicação encarece a produção da rede, apesar de seus nós consumirem uma menor quantidade de energia, por outro, nós sensores produzidos sem planejamento para aplicações específicas tendem a consumir maior quantidade de energia, mas o custo de produção pode ser menor pela diversidade de aplicações. Há, portanto, a necessidade de se ponderar essas duas formas de "economia": baixo custo de produção e baixo consumo de energia. O desafio estaria em desenvolver nós sensores baseados em grupos funcionais (tipo de sensoriamento, por exemplo) e não a partir dos requisitos específicos de uma aplicação.

A escalabilidade é outro parâmetro ainda não muito bem explorado nos trabalhos científicos. A tarefa de desenvolver e implantar rede que contenham dezenas de milhares de nós com uma organização e característica específica não é uma tarefa simples. Mesmo em ambiente computacional, a utilização de dezenas de milhares de nós muitas vezes não é factível em alguns simuladores e a análise dos dados gerados não é de fácil realização.

A integração tempo real, segurança e qualidade de serviço é outro desafio de extrema complexidade nas Redes de Sensores Sem Fio. Tais características são específicas para determinadas aplicações e certamente não são requisito para grande parte delas. Apesar disso, quando necessário, a combinação desses três requisitos é um grande desafio para os desenvolvedores de protocolos.

Com relação à aleatoriedade na distribuição dos nós no ambiente de sensoriamento, há a necessidade de se desenvolver um mecanismo de verificação da cobertura estabelecida pelos nós. Não é uma tarefa (energeticamente) simples identificar se todos os nós distribuídos no meio estão conectados e, ao mesmo tempo, se a densidade de nós está adequada para o sensoriamento da área. Outro ponto a ser considerado é o desenvolvimento de políticas para inserção de novos nós no ambiente em monitoramento, tanto para fins de completar a cobertura de sensoriamento como para ampliar o tempo de vida da rede com o auxílio de novos nós.

A união de funcionalidades das camadas do modelo de rede pode ser encarada como uma forma de minimizar os gastos energéticos relacionados à comunicação, uma vez que tanto o tamanho das mensagens como a quantidade de mensagens enviadas e recebidas pode ser reduzida. Tal estratégia é um desafio, na medida em que muitos módulos de rádios utilizados em nós sensores são desenvolvidos com uma pilha de protocolos já pré-estabelecida.

A fragmentação presente nas pesquisas de WSN é uma característica que dificulta a integração dos dados de simulação e de ambientes experimentais [HAN03]. Não existe uma

metodologia comum que permita aos pesquisadores avaliarem de forma sistemática seus trabalhos e compará-los com os demais. A heterogeneidade de aplicações e as diferentes topologias das Redes de Sensores Sem Fio são os principais fatores que dificultam o estabelecimento de um conjunto dos elementos que caracteriza essas redes de forma a permitir uma avaliação comparativa dos seus desempenhos.

Há, ainda, uma diferença entre projetos e protocolos desenvolvidos por grupos mais focados no desenvolvimento de hardware e grupos mais focados em desenvolvimento de software. Enquanto os primeiros possuem uma visão mais voltada para o desempenho de dispositivos e descrição do funcionamento dos mesmos, os segundos tendem a abstrair em demasia a noção de rede, vendo-a praticamente como um grafo. O desenvolvimento de Redes de Sensores Sem Fio pode ser encarado como uma oportunidade e ao mesmo tempo um desafio que permite unir esses dois enfoques, reduzindo a lacuna existente entre o mundo físico e o mundo computacional.

Resumo

As Redes de Sensores Sem Fio possuem aplicações em diversos campos: engenharia, indústria, agropecuária, geologia, climatologia etc. Este capítulo buscou conceituar as WSN descrevendo suas principais características, classificações e arquiteturas (hardware e software). De forma resumida, foram descritos os principais protocolos de comunicação desenvolvidos para as WSN, seguindo a divisão estabelecida pelo modelo OSI.

O ciclo de vida de uma WSN foi descrito, assim como foram comentadas as principais maneiras de se determinar o tempo de vida da rede. Apesar de não haver um consenso, de maneira geral, os elementos formadores das redes e a sua operação está diretamente relacionada à aplicação da rede.

Um panorama do consumo e gerenciamento de energia das Redes de Sensores Sem Fio foi traçado, além de destacados os principais desafio existentes no desenvolvimento dessas redes.

Este capítulo apresentou alguns projetos de nós sensores utilizados em *testbeds* de WSN e aplicações específicas. No seguinte, Capítulo 3, os simuladores existentes para o estudo do comportamento das WSN serão abordados.

3
Simuladores de Rede de Sensores Sem Fio

"É indispensável estudar a natureza dos outros
antes de darmos livre curso à nossa".
August Strindberg

Os projetos de Redes de Sensores Sem Fio podem se tornar demasiadamente complexos sem o auxílio de ferramentas computacionais dedicadas que permitam escalabilidade, facilidade na distribuição topológica dos nós, reprogramação rápida do conteúdo de software dos nós e uma análise adequada do desempenho e tempo de vida da rede.

Os simuladores são largamente utilizados como meio de antecipar o comportamento das WSN antes que elas de fato sejam materializadas, permitindo um maior entendimento do seu funcionamento, identificação de problemas lógicos e estruturais e limitações da aplicação. Isso se traduz em minimização dos custos e no tempo de projeto, principalmente no que diz respeito à escalabilidade da rede e dificuldade na implantação e recuperação dos nós nos ambientes.

Muitos simuladores dedicados para WSN foram propostos e vêm sendo utilizados tanto em meio acadêmico como em aplicações comerciais [QUA01], enquanto outros simuladores desenvolvidos para redes de qualquer natureza têm sido adaptados para simulações de WSN [NS_97][SOB06][PAR00][OPN12]. Contudo, a facilidade na descrição dos elementos de hardware, permissão para programação de protocolos de comunicação sem

imposição rígida de camadas de redes, manipulação de variáveis do ambiente no qual os nós são testados e inclusão de analisadores específicos de desempenho das redes muitas vezes não são verificadas em conjunto nos simuladores ou estão disponíveis de forma limitada.

Apesar disso, [KUR05] apresenta uma análise estatística dos artigos referentes às MANETS publicados na MobiHoc[1] do ano de 2000, resultado que em 75,5% deles os autores utilizaram simuladores para validar seus experimentos. É relevante, portanto, a importância dos simuladores no estudo das Redes de Sensores Sem Fio.

Este capítulo apresenta um breve comentário sobre as vantagens e desvantagens de se utilizar simuladores no desenvolvimento de WSN e relaciona os simuladores comumente utilizados no desenvolvimento de WSN, ressaltando suas principais características e limitações.

Na medida em que as deficiências e restrições dos simuladores existentes para o estudo das WSN são evidenciadas, os requisitos estruturais do WSN-BaSS (*Wireless Sensor Network – Barrier Synchronization Simulator*), simulador desenvolvido como parte desta tese (Capítulo 5), são determinados.

3.1 – Modelo Computacional e Simulação

Um modelo é definido como uma abstração física ou matemática de um processo, dispositivo ou conceito do mundo real [KNE93]. Um modelo computacional (ou modelo de simulação) pode ser definido como uma representação matemática e/ou algorítmica do comportamento de um sistema real, sendo implementado em uma linguagem capaz de ser interpretada por computadores.

Modelos computacionais são desenvolvidos com um propósito ou aplicação específicos [SAR10]. O resultado da implementação de um modelo computacional é um software denominado simulador[2], capaz que reproduzir em ambiente computacional o comportamento de um sistema específico.

Contudo, os resultados produzidos por simuladores nunca são idênticos ao encontrados nos sistemas do mundo real. Isto ocorre devido às reduções descritivas inerentes da compreensão do funcionamento do sistema real e pelas limitações encontradas no ambiente computacional que reproduzirá o modelo.

Apesar dessa inexatidão representativa, uma ferramenta de simulação possui pelo

[1] MobiHoc (*Mobile Ad Hoc Networking and Computing*) é um Simpósio Internacional em Redes Ad Hoc e Computação organizado anualmente pela ACM (*Association for Computing Machinery*).
[2] Um **simulador** difere de um **emulador** na medida em que o primeiro é um software que permite a representação e o estudo do funcionamento de um sistema qualquer e o segundo é um software executado em um computador que reproduz as funções de outro dispositivo computacional.

menos quatro vantagens que a torna fundamental no estudo de determinados sistemas reais. A primeira é a praticidade e velocidade na execução de experimentos em comparação com a sua realização no ambiente real. A segunda é a possibilidade de estudo do comportamento do ambiente real sem a necessidade da intervenção humana ou inserção de máquinas no ambiente, o que muitas vezes pode alterar as características que se pretende observar ou ainda provocar a depredação do ambiente. A terceira é a capacidade de predição dos simuladores, o que permite o estabelecimento de medidas mitigadoras para a preservação de sistemas reais antes que eles sejam degradados. E, finalmente, a quarta vantagem está relacionada ao custo relativamente mais baixo no estudo de ambientes pelo uso de simuladores se comparado a investigações realizadas *in situ*.

Tradicionalmente existem dois tipos de simuladores: os analíticos (*analytic simulators*) e os ambientes virtuais (*virtual environments*) [FUJ00]. Enquanto os simuladores analíticos se preocupam em executar simulações o mais rápido possível, realizando análise de sistemas complexos e reproduzindo de forma precisa as relações temporais, os ambientes virtuais podem ser considerados como de tempo-real, criando um ambiente realístico e dependente integralmente da ação humana no controle das entidades pertencentes ao modelo.

O desenvolvimento de um simulador para Redes de Sensores Sem Fio deve levar em consideração que o usuário/programador almeja uma execução rápida das suas simulações e, ao mesmo tempo, que elas descrevam a realidade da forma mais exata possível. Os simuladores analíticos são, portanto, os que melhor se adéquam às necessidades de desenvolvimento das WSN.

Os ambientes virtuais geralmente se limitam a aplicações como jogos interativos, treinamento de profissionais (pilotagem de aeronaves, por exemplo) e representação de ambientes remotos controláveis (cirurgias médicas realizadas à distância, por exemplo), possuindo pouca ou nenhuma utilidade no desenvolvimento de WSN.

3.1.1 – Simulação *versus* Realidade

Conforme exposto, a utilização de um simulador para a representação de um sistema possui diversas vantagens relacionadas à flexibilidade na elaboração e execução de modelos, rapidez na obtenção dos resultados e baixo custo de desenvolvimento. Contudo, um simulador só pode ser considerado adequado se for elaborado segundo alguns critérios que o validem (Figura 3.1).

Antes da obtenção de um modelo computacional, o sistema real deve ser estudado e um modelo conceitual deve ser produzido a partir de dados experimentais ou conhecimento

matemático/algorítmico do funcionamento do sistema. Este modelo conceitual deve ser validado através de testes realizados no sistema real, confrontando os resultados obtidos no modelo e no sistema real, realizando os ajustes necessários no modelo.

Uma vez que o modelo conceitual apresenta-se calibrado, um modelo computacional pode ser implementado a partir das regras de funcionamento do sistema estabelecidas no modelo conceitual. Testes devem ser realizados para a verificação se o modelo computacional é fiel ao modelo conceitual.

Em uma primeira fase, o modelo computacional deve ser utilizado, assim como o modelo conceitual o foi, para comparar seus resultados com o comportamento do sistema real, servindo como base para um melhor entendimento do funcionamento da entidade-problema. A constatação de erros de implementação e inconsistências com o modelo conceitual (ou sistema real) são frequentes, servindo para o aperfeiçoamento do modelo. Uma vez que o modelo computacional é validado pela sua utilização, ele é elegível para entrar em uma segunda fase de utilização, na qual a realização de experimentos permite prever o comportamento do sistema real de forma mais confiável.

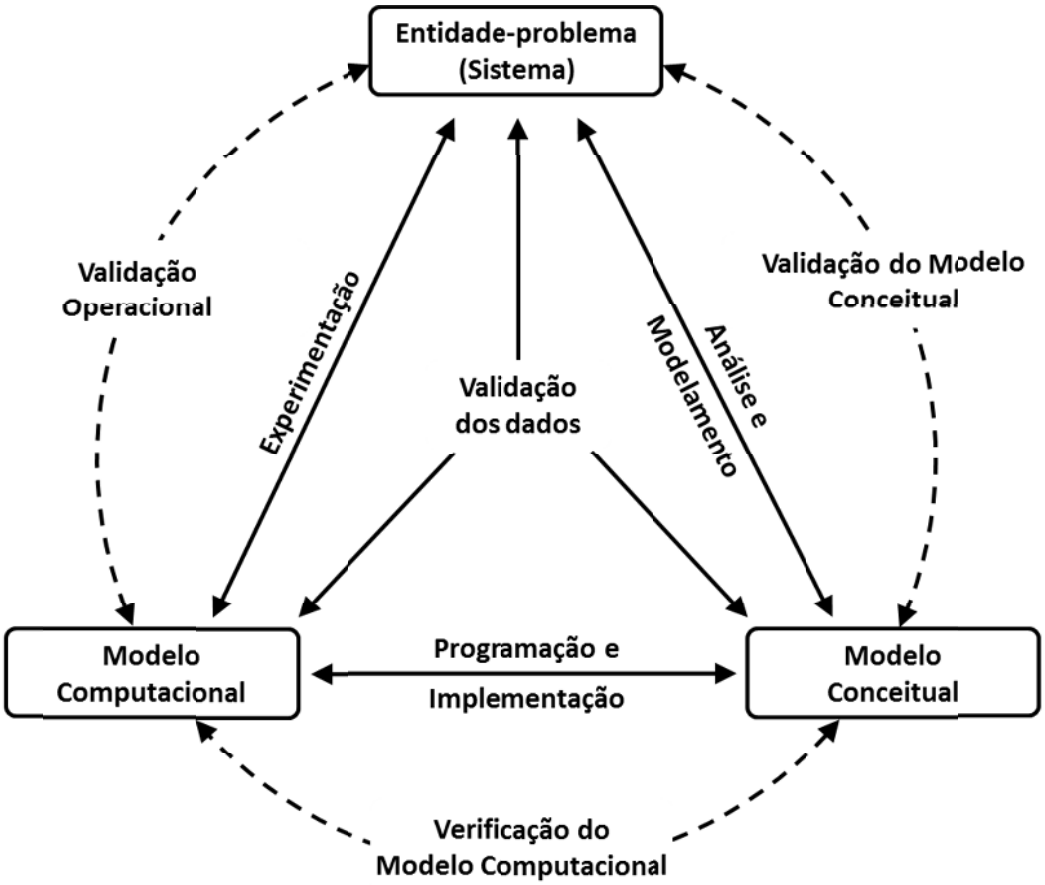


Figura 3.1 – Diagrama do Processo de Modelamento. Adaptado de [SAR10].

Portanto, um simulador nada mais é do que um modelo computacional que prevê o comportamento de um sistema real. No caso do desenvolvimento de um simulador para Redes de Sensores Sem Fio, o que se busca é criar um ambiente de simulação capaz de incorporar três tipos de modelos conceituais: (1) modelos de funcionamento dos nós de rede; (2) modelos de comunicação entre os nós; (3) modelos das variáveis ambientais nas quais a rede está inserida e que, na maior parte dos casos, monitora. A composição formada por esses três modelos permite prever o comportamento de uma WSN no mundo real.

3.1.2 – Simulação *versus* Protótipos

Assim como na maioria dos Sistemas Embarcados, as Redes de Sensores Sem Fio podem ser desenvolvidas tanto com o auxílio de simuladores como pela utilização de protótipos (*testbeds*). Essas duas abordagens podem - e em alguns casos devem - ser utilizadas em conjunto no desenvolvimento dos projetos.

Enquanto os simuladores são desenvolvidos baseados na descrição do comportamento dos objetos e do funcionamento do sistema, podendo haver a reprodução dos seus elementos em larga escala sem muita dificuldade, os protótipos são desenvolvidos seguindo as especificações de seus projetos e materializados de forma individual. Questões relacionadas ao processo de fabricação, disponibilidade de tecnologia e custo de componentes muitas vezes dificultam a produção de diversas unidades de protótipos para fins de teste.

A principal vantagem dos protótipos está no fato de que ao serem submetidos aos testes produzem resultados reais e permitem que problemas inerentes ao mundo físico sejam descobertos. Tais problemas muitas vezes não são revelados em ambientes de simulação, devido à natureza deveras restrita da representação das entidades em ambiente computacional.

Comparando as técnicas de desenvolvimento utilizando simuladores e protótipos, fica evidente a maior exatidão dos testes realizados em protótipos. Contudo, no desenvolvimento de Redes de Sensores Sem Fio (e outros sistemas que possuam vários dispositivos autônomos que estabelecem comunicação entre si) pode ser mais vantajosa a utilização de simuladores em algumas fases do projeto, principalmente na avaliação da funcionalidade e desempenho de protocolos de comunicação.

A Tabela 3.1 apresenta as principais características que se busca no projeto e análise de Redes de Sensores Sem Fio, comparando o método de desenvolvimento por simulação e por protótipo. No que diz respeito à praticidade de manuseio da rede tanto no sentido físico (*hardware*) como lógico (*software*), rapidez na obtenção dos resultados e controle das variáveis ambientais, a utilização de simuladores é mais adequada; se a exatidão e a

avaliação das interferências reais externas são requisitos importantes, é mais apropriada a utilização de protótipos dos nós sensores.

A união das duas técnicas de projeto garante uma melhor avaliação do desempenho da rede. A técnica de simulação permite alterações rápidas nos *software* dos nós durante a validação de protocolos de comunicação e permite o fácil teste da rede com dezenas/milhares de nós e diversificadas topologias. Protótipos de nós, por sua vez, seriam utilizados antes da fase de simulação como meio para obtenção de parâmetros reais da rede (consumo dos nós em cada estado, alcance real dos rádios de acordo com a potência de transmissão, por exemplo) e após a fase de simulação (quando o protocolo de comunicação já está definido e testado), em menor quantidade (algumas unidades), mas garantindo plenamente que as funcionalidades da rede estejam de acordo com as especificações do projeto.

Tabela 3.1 – Características do desenvolvimento de WSN utilizando Simulação e Protótipo

Característica	Simulação	Protótipo
Custo na produção dos nós	Custo zero	Custo alto
Manuseio dos nós (escalabilidade)	Nenhuma dificuldade	Dificuldade alta
Reprogramação ou reconfiguração dos nós	Fácil execução	Operação manual ou gasto de energia dos nós para atualização via rádio
Medição do Consumo Energético	Necessidade de um modelo de bateria adequado	Pode ser realizada por instrumentação apropriada, medindo o consumo real
Avaliação das Funcionalidades da Rede	Satisfatória, passível de comparação entre modelos	Resultado excelente (real)
Tempo necessário para avaliação do protocolo e tempo de vida da rede	Rápido, depende do potencial computacional e possivelmente do número de nós da rede e quantidade de mensagens enviadas	Tempo real. Pode ser inviável para análise do tempo de vida da rede
Repetibilidade nos testes	Extremamente preciso. Números pseudo-aleatórios garantem a repetibilidade dos testes e até mesmo na repetição de interferências na comunicação	Pode haver problemas de repetibilidade quando o ambiente está sujeito a interferências na comunicação
Averiguação se os erros das mensagens foram causados por colisão ou interferência	Origem do erro facilmente identificável. O simulador tem controle de tudo que acontece com os nós e as mensagens.	Não identificável, a não ser que o ambiente seja extremamente controlado (o que o torna de custo elevado)

Os simuladores são, portanto, parte importante no processo de desenvolvimento de WSN. Contudo, devido às restrições do ambiente computacional com relação à limitação das linguagens de programação em descrever objetos e fenômenos do mundo real (limitação de *software*) e principalmente à frequência máxima na execução de instruções (limitação de *hardware*), faz-se necessário ponderar o detalhamento na descrição do modelo (funcionamento dos componentes e protocolos de comunicação) para se conseguir um tempo de simulação compatível com o projeto. Isto é visto em detalhes no Capítulo 4.

Outra característica importante dos simuladores é que eles permitem um maior controle na repetição de experimentos (cenários específicos). Muitas vezes se almeja repetir uma sequência de comunicação entre os nós de uma WSN utilizando protocolos de comunicação diferentes. Em *testbeds* pode ocorrer problemas de interferências não controláveis que mascararam os resultados, perturbando a comunicação e gerando perdas de mensagens. Há, ainda, a dificuldade no estudo de colisão de mensagens em *testbeds*, pois na utilização de protótipos não existe nenhum elemento do sistema capaz de identificar se uma mensagem falha (com erro de CRC) foi originada por colisão (rádios de dois ou mais nós transmitindo ao mesmo tempo) ou por interferência do ambiente. Isso pode prejudicar a análise de desempenho de protocolos de comunicação, sendo que apenas em simulação esse tipo de análise pode ser realizado com fidelidade.

3.2 – Simuladores de Redes de Sensores Sem Fio

A disponibilidade de simuladores voltados para o estudo das Redes de Sensores Sem Fio vem crescendo desde o final da década de 90, a partir da inclusão de módulos específicos para WSN no simulador ns-2 [NS_97]. A escolha de um simulador adequado para uma determina análise em WSN não é tarefa simples, uma vez que cada grupo de pesquisa possui um foco diferente e a divergência de recursos encontrados nos simuladores muitas vezes não contempla todas as necessidades envolvidas no projeto.

A Tabela 3.2 apresenta em ordem alfabética as 17 ferramentas de simulação e emulação mais utilizadas no estudo das Redes de Sensores Sem Fio. Foram selecionadas as que apareceram mais vezes na literatura pesquisada, sendo que a maioria já foi alvo de investigação com relação à sua eficiência e suas deficiências.

Nem todos os simuladores presentes na Tabela 3.2 são dedicados às Redes de Sensores Sem Fio, mas são utilizados em estudos focados em determinadas particularidades das WSN. Muitas vezes o conhecimento e a familiaridade de um grupo de pesquisa na utilização de determinada ferramenta de simulação de redes (ou redes de sensores) pode gerar

resultados tão bons ou até superiores aos conseguidos com a utilização de ferramentas específicas para simulação de Redes de Sensores Sem Fio ainda em desenvolvimento e não totalmente validadas.

Tabela 3.2 – Ferramentas de Simulação e Emulação utilizadas para WSN

Ferramenta	Tipo	Linguagem	Repositório na Internet
ATEMU [POL04]	Emulador (Mica2)	C	http://www.hynet.umd.edu/research/atemu/
Avrora [TIT05]	Simulador para WSN	Java	http://compilers.cs.ucla.edu/avrora/
Castalia [BOU07]	Simulador para WSN	C++	http://castalia.research.nicta.com.au/index.php/en/
Cooja [SEH13]	Simulador do Contiki	C	http://www.contiki-os.org/start.html
Emstar [GIR07]	Emulador (Mica2)	C	http://static.usenix.org/event/usenix04/tech/general/full_papers/girod/girod_html/eu.html
GloMoSim [ZEN98]	Simulador de redes	C	http://pcl.cs.ucla.edu/projects/glomosim/
GTSNetS [OUL05]	Simulador para WSN	C++	http://www.ece.gatech.edu/research/labs/MANIACS/GTNetS/
J-Sim [SOB06]	Simulador de redes	Java	https://sites.google.com/site/jsimofficial/
MiXiM [KÖP08]	Simulador de redes	C++	http://mixim.sourceforge.net/
ns-2 / **ns-3** [NS_97] / [NS_05]	Simulador de redes	C++	http://www.isi.edu/nsnam/ns/ http://www.nsnam.org/
OMNeT++ [WAN05]	Simulador de redes	C++	http://www.omnetpp.org/
OPNET [OPN12]	Simulador de redes	C / C++ / Java	http://www.opnet.com/solutions/network_rd/modeler.html
QualNet [QUA01]	Simulador de redes	C	http://web.scalable-networks.com/content/qualnet
SENS [SAM04]	Simulador para WSN	C++	http://osl.cs.uiuc.edu/sens/
SENSE [CHE04]	Simulador para WSN	C++	http://www.ita.cs.rpi.edu/
SensorSim [PAR00]	Simulador de redes	C	http://nesl.ee.ucla.edu/projects/sensorsim/
TOSSIM [LEV03]	Emulador (Mica2)	C++ / Python	http://www.cs.berkeley.edu/~pal/research/tossim.html

Além dos simuladores e emuladores listados, outros merecem ser citados, mas com menor destaque devido a pouca utilização em pesquisas. O JSensor [RIB12], MATSNL [JUN07], Shawn [KRO05], Sidh [CAR05], VisualSense [BAL05] e WSNet [WSN06] são exemplos.

Os itens seguintes apresentam uma breve descrição das principais características das ferramentas de simulação presentes na Tabela 3.2. Não se busca um detalhamento rigoroso de cada ferramenta, tampouco é apresentada uma análise quantitativa de desempenho dos simuladores, uma vez que essa análise ultrapassa o propósito dessa tese. A motivação para o estudo apresentado é identificar as principais características desses simuladores e emuladores de Redes de Sensores Sem Fio, relacionando seus atributos e suas limitações de uso. Quando possível, são identificadas as vantagens e desvantagens na utilização das ferramentas.

A maioria das descrições foi realizada pelo estudo dos artigos de divulgação científica das ferramentas de simulação/emulação, manuais de usuário e sítios da Internet nos quais há a disponibilização das ferramentas e fóruns de discussão. A exceção está nos simuladores ns-2, J-Sim e QualNet (versão demonstrativa), nos quais foram realizadas suas instalações e executadas simulações.

Os estudos comparativos entre as ferramentas de simulação e emulação de Redes de Sensores Sem Fio apresentados em [IMR10], [JAI11], [KHA11], [CAV02] e [LHA12] foram de fundamental importância na elaboração dessa súmula apresentada a seguir.

3.2.1 – The Network Simulator (ns-2/ns-3) e SensorSim

O simulador ns-2 [NS_97] foi desenvolvido em 1997 a partir da sua versão preliminar (ns-1) na Universidade de Berkeley (EUA). Possuindo um núcleo de simulação do tipo *event driven* (vide Capítulo 4) e sendo desenvolvido na linguagem C++, é um simulador de redes de propósito geral, voltado tanto para o estudo de redes com fio como para redes sem fio. É o simulador de redes mais utilizado em pesquisas acadêmicas por disponibilizar uma grande quantidade de protocolos em todas as camadas de rede.

Segundo [JAI11], a dificuldade para se iniciar a utilização do ns-2 é a necessidade de aprendizado da linguagem TCL (*Tool Command Language*), linguagem de difícil entendimento e escrita. A dupla necessidade do conhecimento de TCL e C++ pode desencorajar o seu aprendizado e utilização [CAV02]. Além disso, a interface gráfica do ns-2 é limitada e a visualização da simulação só pode ser feita após a execução total da mesma.

Com relação à simulação de Redes de Sensores Sem Fio, o ns-2 disponibiliza

modelos de energia e de bateria, além de permitir a mobilidade dos nós. Contudo, o grande problema é a escalabilidade, pois redes com mais de 100 nós podem encontrar dificuldade de serem simuladas [JAI11][KHA11][KOR09]. Possivelmente esta limitação está relacionada à elevada utilização de memória pelo ns-2 [CAV02].

O fato das WSN apresentarem uma aplicação diretamente associada ao processo de comunicação também representa uma dificuldade para sua inclusão no modelo de simulação do ns-2, uma vez que o foco do simulador está nas camadas de rede e não no comportamento da rede diante a sua aplicação.

A contabilização da energia utilizada nos nós no ns-2 também é limitada aos procedimentos de envio e recebimento de mensagens, desprezando o gasto energético relacionado ao processamento e sensoriamento, fundamentais para determinação do tempo de vida das WSN [LHA12]. O ns-3 [NS_05] corrige esse problema, criando duas entidades relacionadas à energia no nó: uma relacionada à comunicação e a outra ao consumo do dispositivo eletrônico do nó.

Apesar do ns-2 e ns-3 apresentarem uma boa extensibilidade com relação à adição de novos protocolos e módulos, o fato da sua programação ser aberta permite a incorporação de código com problemas de travamento e muitas vezes não otimizados.

O simulador SensorSim [PAR00] foi desenvolvido tendo como base o ns-2, mas direcionando seu enfoque para redes de sensores, sejam elas com ou sem fio. Um modelo de consumo de energia mais adequado ao estudo das WSN foi incluído no SensorSim, sendo uma entidade central responsável por contabilizar separadamente a energia utilizada pelo rádio, CPU e pelos sensores presentes nos nós.

O SensorSim é um simulador considerado híbrido, isto é, permite o estabelecimento de comunicação dos nós simulados com nós de uma rede real. A viabilidade desse tipo de simulação é questionável, uma vez que o mecanismo de sincronização da rede virtual com a rede real pode provocar, em muitos casos, problemas na interpretação dos resultados.

3.2.2 – SENSE

O SENSE (*Sensor Network Simulator and Emulator* – Simulador e Emulador de Redes de Sensores) [CHE04] foi desenvolvido tendo como foco a extensibilidade, a reusabilidade e a escalabilidade [IMR10][KHA11].

O simulador SENSE foi desenvolvido em C++ tendo como base o simulador *event driven* de propósito geral COST [GIL02]. A arquitetura interna do simulador não está baseada no conceito de orientação à objetos, mas em porta-componente (*componente-port*) que permite

uma maior modularidade e independência entre os componentes[3] [KHA11].

A execução da simulação no SENSE pode ser realizada de forma serial ou paralela, sendo essa uma opção determinada pelos usuários do simulador. O resultado da simulação não dispõe de uma interface gráfica dedicada para visualização, fazendo com que o acompanhamento da simulação seja prejudicado e muitas vezes impedindo a detecção de erros.

3.2.3 – J-Sim

O J-Sim [SOB06] é um simulador desenvolvido na linguagem de programação Java. Apesar de a sua utilização estar comumente relacionada a simulações na área médica, o J-Sim também vem sendo empregado em simulações de Redes de Sensores Sem Fio.

A organização da programação orientada à objetos associada à arquitetura do J-Sim permite uma grande reusabilidade do código desenvolvido para este simulador. A biblioteca de classes do J-Sim já possui uma grande quantidade de protocolos para Redes de Sensores Sem Fio e mecanismos para a análise do consumo de energia nos nós sensores. Além disso, possui uma biblioteca de classes responsáveis por gerar uma interface gráfica para o acompanhamento das simulações.

Com relação à escalabilidade, o J-Sim permite a simulação de cerca de 500 nós [JAI11], possibilitada devido à otimização da JVM no uso da memória.

Apesar de todos os benefícios apresentados, [JAI11] salienta que pelo fato do J-Sim não ter sido desenvolvido no contexto das WSN, a inclusão de novos protocolos e componentes ao simulador não é realizada de forma simples.

3.2.4 – TOSSIM e ATEMU

O TOSSIM [LEV03] é um emulador para o módulos Mica2 [MOT02] e o sistema operacional TinyOS [TIN06]. Ele foi desenvolvido nas linguagens de programação C++ e Python, permitindo a execução de experimentos com milhares de nós sensores.

Por ser um emulador, o resultado da simulação, no que diz respeito à aplicação em execução nos nós sensores, é extremamente fiel à realidade. Os modelos de rádio também foram bem programados [JAI11], descritos no nível de bit. Contudo, há a ausência de um módulo que realize a contabilização da energia utilizada pelos nós das WSN. O PowerTOSSIM

[3]A diferença entre um objeto e um componente está no fato que toda ação executada entre componentes é registrada por uma interface de controle. No caso da arquitetura orientada à objetos, um objeto pode comunicar com outro objeto de forma direta, desde que possua uma referência (apontador) para o outro objeto. Essa possibilidade não existe na arquitetura de componentes.

[SHN04][PER08] é uma extensão pode ser adicionada ao TOSSIM para se realizar a contabilidade do gasto energético nos nós, essencial para o estudo das WSN.

Uma restrição encontrada no TOSSIM é ele ser limitado à emulação dos *motes* Mica2, não permitindo a simulação de redes de sensores com nós heterogêneos. A programação em emulação, portanto, é a mesma para todos os nós sensores.

O TOSSIM não permite a representação do ambiente no qual os nós estão inseridos, sendo uma limitação que restringe o acionamento dos *motes* na ocorrência de um determinado evento determinado pelo ambiente.

O TinyVIZ pode ser utilizado como uma ferramenta auxiliar do TOSSIM, permitindo a visualização da rede através de uma interface gráfica.

O ATEMU é um emulador gratuito específico para Redes de Sensores Sem Fio baseado no simulador TOSSIM. Ele emula o processamento de microcontroladores AVR [ATM13] comumente utilizados nos *motes* Mica2. Não se limitando à emulação das instruções internas de cada nó sensor, o ATEMU emula a comunicação entre os nós e registra o consume de energia nos nós sensores.

A grande vantagem do ATEMU é que ele permite que cada nó sensor possua um programa diferente. Sendo assim, dentro da arquitetura para WSN proposta pela AVR e implementada nos *motes* Mica, o ATEMU permite a emulação de redes homogêneas ou heterogêneas.

Como todo emulador, a grande desvantagem do ATEMU está relacionada à sua restrição de plataforma. Além disso, o tempo de simulação é mais lento que os demais simuladores [JAI11].

3.2.5 – GloMoSim e QualNet

O GloMoSim (*Global Mobile System Simulator* – Simulador de Sistema Móvel Global) [ZEN98] é um simulador desenvolvido na UCLA (Universidade da Califórnia) para redes com e sem fio. Este simulador foi construído sobre um núcleo de simulação com processamento paralelo do tipo *event driven* denominado PARSEC.

Utilizando o conceito de camadas, os atributos de simulação no GloMoSim são determinados através de arquivos de configuração (arquivos texto). Existe uma dificuldade para se descrever aplicações que não utilizam algumas camadas do modelo OSI [CAV02], sendo, portanto, esta estrutura demasiadamente rígida para o estudo de protocolos para WSN. De fato, [KHA11] considera o GloMoSim inadequado para o estudo de WSN devido a esta característica.

O GloMoSim é um simulador aberto e gratuito, mas foi atualizado somente até o ano

de 2000. A partir de então, passou a ser vendido como um produto pago denominado QualNet [QUA01].

O QualNet possui uma interface gráfica na qual é possível o desenvolvimento de protocolos, criação de cenários de redes com movimento e realizar a análise de desempenho da rede. Ele é composto por cinco módulos gráficos básicos: *QualNet Architect* (desenvolvimento de cenários e visualização da rede), *QualNet Analyzer* (ferramenta estatística para análise do comportamento da rede), *QualNet Packet Tracer* (para acompanhamento da trajetória de pacotes de dados pela rede), *QualNet File Editor* (para edição dos arquivos descritivos da simulação) e *QualNet Command Line Interface* (linha de comando para executar ações de controle da simulação).

3.2.6 – OMNeT++, Castália e MiXiM

O OMNeT++ [WAN05] é um simulador escrito em C++, baseado em um núcleo de simulação *event driven*. É um simulador não específico para WSN, podendo ser utilizado tanto para redes com e sem fio, mas possui módulos específicos para simulação de WSN, como o de controle do consumo de energia dos nós.

O simulador OMNeT++ possui interface gráfica tanto para a descrição da simulação como para acompanhar os resultados da sua execução em tempo real. O módulo EYE disponibiliza uma forma de definir o mapa de simulação estabelecendo probabilidades de erro e queda no sinal da transmissão.

Como diferentes grupos desenvolveram as partes do OMNeT++, há um problema de integração entre os módulos, não raro havendo o reporte de erros no decorrer das simulações [JAI11].

O Castalia [BOU07] é um ambiente de simulação para redes de sensores sem fio construído sobre o OMNeT++. Sua principal característica é possuir um modelo preciso de canal sem fio, além de possuir protocolos MAC e de roteamento já programados.

O MiXiM [KÖP08] é outro simulador baseado na arquitetura OMNeT++, adicionando à este diversas funcionalidades relacionadas à simulação de redes sem fio.

Tanto os nós de rede como os obstáculos podem possuir mobilidade no MiXiM. Essa mobilidade pode ser fundamental no estudo de algumas WSN, uma vez que a movimentação dos nós pode gerar a interrupção da comunicação entre os nós, forçando o estabelecimento de novas rotas para as mensagens. Na representação do MiXiM, os nós da rede possuem apenas uma localização, enquanto que os obstáculos possuem tanto localização como dimensões físicas, sendo esta importante para a determinação da atenuação na

propagação das ondas.

A identificação de colisão de mensagens é outro atributo importante no MiXiM, pois o estudo de protocolos de comunicação em WSN depende desse tipo de análise na determinação da sua viabilidade.

3.2.7 – OPNET

O OPNET [OPN12] é um simulador comercial originalmente destinado ao estudo de redes cabeadas. Com a sua evolução, foram incluídos módulos e protocolos que possibilitam a simulação de redes sem fio. A maior parte da simulação pode ser descrita e configurada com o uso de interface gráfica, sendo poucos itens programados em C, C++ ou Java.

O grande diferencial do OPNET é o seu modelo de transmissão e representação do meio físico, havendo a possibilidade de modelamento em 3D de diferentes tipos de terrenos, incluindo obstáculos.

Uma simulação no OPNET é constituída de três fases. Na primeira fase a topologia da rede deve ser descrita; na segunda, os nós sensores e o fluxo de mensagens são determinados; e na terceira, o protocolo de comunicação, os recursos e a aplicação são definidos por uma máquina de estados finita [KOR09]. Cada nível da pilha de protocolos deve ser descrito como uma máquina de estados, o que muitas vezes se torna penosa a tarefa de conversão a partir do algoritmo do protocolo [CAV02].

O OPNET possui um modelo que simula falha nos nós, sendo essa característica particularmente útil no estudo das WSN.

3.2.8 – Avrora

O simulador Avrora [TIT05] foi desenvolvido na linguagem Java especialmente para o estudo de Redes de Sensores Sem Fio.

Apesar de não ser um emular *stricto sensu*, o Avrora, assim como o ATEMU, simula os *motes* Mica2 baseados nos microcontroladores AVR [ATM13]. Mas diferentemente do ATEMU, o Avrora, por não ser um emulador vinculado à plataforma Mica, permite a simulação de outros códigos diferentes dos suportados pelo TinyOS.

Uma vez que apresenta uma execução instrução-por-instrução e não há sincronização entre os nós, o Avrora permite a simulação de até 10.000 nós sensores [IMR10]. Cada nó sensor é representado por uma *thread* em Java.

Apesar de essa escalabilidade ser uma possibilidade pertinente ao estudo das WSN, a falta de sincronização entre os nós pode ser um limitante para se atingir uma exatidão

adequada ao estudo das WSN.

O Avrora não disponibiliza interface gráfica para visualização da simulação.

3.2.9 – SENS

O simulador SENS (*Sensor, Environment and Network Simulator* – Simulador de Sensor, Ambiente e Rede) [SAM04] foi desenvolvido na linguagem C++ com o propósito específico de simulação de Redes de Sensores Sem Fio.

A arquitetura do SENS é modular e em camadas, permitindo a descrição da aplicação, do modelo de comunicação da rede e do meio físico. A aplicação consiste no elemento de software que deverá ser executado no nó sensor. O modelo de comunicação está relacionado às funções de envio e recebimento de pacotes, estes com formato fixo (intensidade do sinal, tempo de transmissão, identificação do emissor e o conteúdo da mensagem).

O ponto forte do SENS é a descrição do meio físico [IMR10], realizada pela descrição de uma matriz composta de elementos com características de propagação de sinal próprias, como grama, concreto e paredes. Outros materiais podem ser adicionados de acordo com a necessidade do usuário. O comportamento do meio pode ser computado à priori, aumentando, assim, a velocidade de execução da simulação, uma vez que não há a necessidade de reprocessar as condições ambientais a cada rodada de simulação.

Quanto à análise de desempenho da rede, o SENS possui a verificação de colisão de pacotes (importante para o estudo da eficiência dos protocolos de comunicação) e provê ferramentas de análise de consumo de energia.

Apesar de possuir uma boa estrutura, o simulador SENS não disponibiliza uma interface gráfica de visualização e sua documentação não está disponível publicamente [IMR10].

3.2.10 – Emstar

O Emstar [GIR07] é um emulador para Rede de Sensores Sem Fio desenvolvido para Linux. Os *motes* Mica2 [MIC02], MicaZ [MIC02] e Telos [TEL04] são alguns dos nós sensores que podem ser emulados no Emstar.

Bibliotecas, ferramentas e serviços compõem a estrutura interna do Emstar [IMR10]: as bibliotecas possuem as funções de envio e recebimento de mensagens; as ferramentas dão suporte à simulação, emulação e visualização da rede; e os serviços, por sua vez, estão relacionados aos eventos de sensoriamento e comunicação de rede.

A estrutura modular do Emstar suporta a emulação do funcionamento de cada módulo separadamente, além de permitir a introdução de falhas nos módulos para efeito de testes e avaliação das consequências no funcionamento da rede [JAI11].

O Emstar apresenta uma interface gráfica para o posicionamento e acompanhamento da simulação, mas esta sempre é realizada em tempo real. O estudo determinístico do tempo de vida da rede, portanto, torna-se inviável no Emstar.

3.2.11 - GTSNetS

O GTSNetS (*Georgia Tech Sensor Network Simulator*) [OUL05] é uma ferramenta de simulação específica para o estudo de Redes de Sensores Sem Fio. A sua arquitetura interna está baseada no GTNetS (*Georgia Tech Network Simulator*) [RIL03].

Desenvolvido na linguagem C++, o GTSNetS tem como principal objetivo a simulação de redes em larga escala (até 200.000 nós). Sua estrutura não busca simular um tipo de rede específica, mas dá ao pesquisador a possibilidade de testar diferentes protocolos, modelos de bateria e aplicações [KHA11].

A arquitetura interna do GTSNetS é composta pelas unidades de processamento, unidades de sensoriamento, unidades de comunicação e unidades de energia (baterias).

A contabilidade do consumo de energia dos nós sensores, o cálculo da expectativa de vida da rede, a mobilidade dos nós e o acompanhamento da rota dos pacotes são algumas características do GTSNetS que o fazem adequado para o estudo das Redes de Sensores Sem Fio.

O principal problema do GTSNetS, segundo [KHA11], está relacionado a falta de continuidade do projeto, que desde 2008 não tem sido atualizado.

3.3 – Considerações sobre os Simuladores de WSN

As pesquisas realizadas com Redes de Sensores Sem Fio revelam a importância e até mesmo a dependência na utilização de simuladores. Mesmo diante esse cenário, não existe um consenso sobre o simulador mais adequado ou ainda o que possua melhores características para determinado estudo em WSN. Muitas vezes grupos de pesquisa desenvolvem seus próprios simuladores, não apresentando detalhes das suas implementações e publicando apenas os resultados referentes ao estudo das Redes de Sensores Sem Fio. Há ainda trabalhos que não especificam qual simulador foi utilizado, como foi constatado em 57,9% dos artigos publicados sobre MANET no MobHoc de 2000 [KUR05].

A credibilidade de um estudo em WSN que utilize simulação está diretamente

relacionada à validação da ferramenta utilizada. Pelo fato dos simuladores serem ferramentas de alta complexidade computacional, principalmente no que se refere ao núcleo de simulação e aos modelos implementados, a validação realizada por testes de software, apesar de necessária, não é suficiente. O uso por diferentes grupos de pesquisa e a constante atualização com correção de erros amplia a validação do simulador, aumentando, assim, a sua credibilidade. Considerando a quantidade de simuladores utilizados, há um prejuízo nesse processo de validação, além de tornar o resultado dos estudos em WSN questionáveis diante possíveis erros introduzidos nos experimentos pela utilização de ferramentas de simulação não satisfatoriamente validadas.

A comparação de desempenho e exatidão dos resultados obtidos entre simuladores não é tarefa fácil, conforme é salientado em [LUC03]. Os critérios de análise quantitativa não são bem estabelecidos e muitas vezes não são aplicáveis diante as diferenças de implementação dos simuladores. O fato de alguns simuladores possuírem seu código e estrutura de processamento não disponível para estudo também é outro impasse para se realizar comparações adequadas e imparciais.

Apesar disso, [CAV02] compara o resultados de simulações realizadas no OPNET, ns-2 e GloMoSim, chegando a conclusão que a diferença de resultados obtidos nesses simuladores está relacionada às implementações das camadas física e MAC. O ns-2 e o OPNET são comparados em [LUC03] em termos de tempo de processamento, chegando a um resultado muito similar. Já a comparação entre o OMNET++ e o ns-2 foi realizada por [XIA08], chegando a conclusão que o OMNET++ possui um melhor desempenho que o ns-2 nos quesitos tempo de processamento e uso de memória.

A comparação entre os resultados obtidos nos simuladores ns-2, QualNet e OPNET com ensaios experimentais (*testbeds*), realizada em [RAC10], conclui que eles são muito próximos, mas possuem uma dependência muito forte com relação aos parâmetros utilizados na caracterização da camada física e no modelo de propagação utilizado.

Outros estudos indicam certa diferença nos resultados obtidos em experimentos realizados em simuladores e em *testbeds*, considerando ambientes abertos ou fechados. [LOH11] compara o ns-2, OPNET e QualNet com um *testbed* de uma rede 802.15.4 (ZigBee), concluindo que o modelo de propagação do ns-2 é mais preciso em relação ao *testbed* em ambientes fechados (*indoor*), enquanto o OPNET atinge resultados mais similares ao *testbed* em ambientes abertos (*outdoor*).

Com relação ao modelo de transmissão de dados utilizados nos simuladores, [KOT04] faz algumas considerações relevantes que os simuladores apresentados nesse capítulo não contemplam ou implementam de forma parcial: a área de alcance dos rádios não

deve ser definida estritamente como um círculo perfeito; o alcance dos rádios não deve ser fixo e igual para todos os nós; a simetria na comunicação não deve ser considerada uma verdade (um nó pode “escutar” outro, mas o contrário nem sempre é verdade); o sinal recebido por um rádio nunca é perfeito (isento de erros); e a potência de recepção não deve ser uma função exclusiva da distância.

Estes quesitos no modelo de transmissão são fundamentais para se criar um cenário mais realista nos simuladores, mas ampliam a necessidade de uma interação maior e mais dinâmica entre os elementos do simulador. Assim, rádio e antena do emissor, rádio e antena do receptor, localização dos nós e condições ambientais que interferem na transmissão devem ser consideradas em conjunto para cada par de nós da simulação.

Os emuladores descritos apresentam como principal limitação a restrição de sua utilização para apenas uma plataforma de *motes*. Para estudos que visam o estudo do comportamento de diferentes *hardwares*, eles são inadequados.

A arquitetura do Emstart revela outro problema, relacionado ao tempo de execução dos experimentos, pois sendo em tempo real, impossibilita a análise adequada do tempo de vida da rede, possivelmente de meses ou anos.

Com relação aos simuladores que não foram desenvolvidos especialmente para o estudo de Redes de Sensores Sem Fio, como é o caso do J-Sim, o principal problema está relacionado à dificuldade de inclusão de novos protocolos de comunicação. O GloMoSiM apresenta problema semelhante, pois sua estrutura engessada às camadas do modelo OSI dificulta o estudo de protocolos de comunicação baseados em outros modelos.

A necessidade de uma interface gráfica também é algo importante e que, preferencialmente, exiba os eventos de rede durante a simulação e não somente no final da mesma. Muitas simulações utilizam muito tempo de processamento, principalmente as que buscam contabilizar o tempo de vida da rede, sendo pertinente a observação dos eventos enquanto estão sendo simulados para que erros possam ser descobertos antes do final da simulação, diminuindo, assim o tempo necessário de pesquisa.

A escalabilidade é outro ponto que os desenvolvedores de simuladores para WSN levam em consideração. Em [SIC05], os autores apontam de forma clara que os simuladores ns-2, OPNET, Glomosim, OMNET++, Qualnet, entre outros, são inadequados para a execução de simulações com milhares de nós por um tempo de simulação de alguns meses ou até anos. Isso se deve pelo fato desses simuladores apresentarem um nível de descrição demasiadamente detalhado das camadas física e MAC. A solução encontrada por esses autores foi o desenvolvimento de um simulador próprio que permitisse a abstração das camadas mais baixas, focando o estudo no problema de consumo de energia.

Apesar da escalabilidade nos simuladores de WSN ser algo importante, ela pode ser questionável na sua amplitude. A Tabela 2.1 apresenta que as Redes de Sensores Sem Fio são formadas por centenas a milhares de nós sensores, sendo essa quantidade razoável para o estudo dessas redes em ambiente computacional. Além disso, de fato existem aplicações com essa demanda, como monitoramento ambiental de grandes áreas, por exemplo. Por outro lado, ferramentas que possibilitam alta escalabilidade geralmente penalizam outras características importantes na simulação, como a exatidão nos processos simulados. O sincronismo entre os nós é de suma importância na exatidão de uma simulação, mas o simulador Avrora, por exemplo, abre mão dessa característica em detrimento da escalabilidade.

A busca pelo desenvolvimento de uma ferramenta de simulação que seja referência para o estudo das Redes de Sensores Sem Fio é ilusória. Cada grupo de pesquisa tem sua linguagem de programação habitual e tendência a utilizar as ferramentas que já estão familiarizadas, uma vez que o custo de aprendizado de novas ferramentas é um item de grande peso no cronograma dos projetos.

Por outro lado, as ferramentas de simulação apresentadas neste capítulo revelam uma multiplicidade de implementações para os simuladores de WSN, sendo as principais diferenças relacionadas à exatidão (sincronização e descrição de elementos de software e hardware), modularidade e estrutura interna, velocidade de processamento, facilidade de uso e disponibilidade de documentação. O estudo dessas diversas formas de abordagens, seus problemas e as soluções adotados são de suma importância para o entendimento e a reflexão das necessidades envolvidas na elaboração de um simulador para Redes de Sensores Sem Fio.

3.4 – Características desejáveis em um simulador de WSN

Apesar de não haver consenso em todos os estudos voltados para as Redes de Sensores Sem Fio, os itens relacionados a seguir podem ser considerados importantes em uma ferramenta de simulação que busque atender às variadas necessidades dos pesquisadores de WSN. Os itens foram extraídos tanto do estudo das WSN apresentados nesse capítulo, das próprias definições de WSN (Capítulo 2) e outros estudos apresentadas nos artigos pesquisados.

Um simulador destinado ao estudo das WSN deve possuir as seguintes características:

- estrutura interna de sincronização de eventos precisa e suficientemente exata para gerar resultados confiáveis;

- elementos de software capazes de representar de forma adequada o funcionamento do hardware, sem ser específico para uma determinada plataforma;
- flexibilidade para a descrição dos elementos de hardware e de software, permitindo ao usuário-programador que ele possa balancear o detalhamento descritivo com a velocidade de execução da simulação, tornando o estudo factível;
- estrutura adequada para o desenvolvimento e estudo de protocolos de comunicação sem haver a necessidade de adaptações a modelos pré-estruturados no simulador;
- possibilidade de descrever o ambiente e as variáveis ambientais que influenciam no funcionamento dos nós de forma adequada aos estudos propostos;
- possibilidade de inclusão de diferentes modelos de transmissão que utilizem como parâmetros as características do rádio, antena e meio no qual estão inseridos os nós sensores;
- escalabilidade suficiente para o estudo de WSN de grande porte (até milhares de nós sensores), mas sem que para isso haja perda de qualidade no resultado obtido nas simulações;
- reusabilidade, isto é, possibilidade de reutilização de módulos já programados nas mais variadas simulações. A modularidade na estrutura do simulador é fundamental, portanto;
- disponibilidade de uma interface gráfica para a criação da rede (definição da sua geometria), além de acompanhamento da simulação através de atributos personalizados para cada estudo, sem haver a necessidade de finalização de todo o tempo de simulação para realizar a visualização dos eventos;
- possibilidade de execução da simulação sem interface gráfica, permitindo que o tempo de simulação seja minimizado em estudos nos quais o acompanhamento visual não se faz necessário.

A extensibilidade deve fazer parte de todos os itens do simulador, isto é, diferentes implementações dos seus componentes devem ser facilmente incluídas, sem haver restrições por parte da estrutura do simulador.

Os dez itens apresentados são contemplados no simulador apresentado nessa tese (Capítulo 5). As ferramentas estudadas e apresentadas nesse capítulo, apesar de serem muito bem elaboradas, não contemplam todos os quesitos levantados, ou são implementados de

forma parcial.

Além dessas características desejáveis em um simulador para WSN, o núcleo de simulação apresentado no próximo capítulo (Capítulo 4) é o grande diferencial do simulador proposto e implementado.

Resumo

Este capítulo apresentou uma descrição sucinta das principais ferramentas de simulação/emulação utilizados no estudo das Redes de Sensores Sem Fio, assim como evidenciou a vasta e necessária utilização de simuladores nesta área.

A importância do procedimento de modelagem para a representação computacional dos elementos a serem simulados foi vista, assim como o papel dos protótipos no estudo das Redes de Sensores Sem Fio.

As características presentes nas ferramentas de simulação levantadas nesse capítulo, assim como os elementos presentes na definição das WSN apresentadas no Capítulo 2, formam as bases do projeto do simulador para o estudo de Redes de Sensores Sem Fio que é apresentado nesta tese. Foram levantadas dez principais características que podem ser consideradas essenciais na composição de uma ferramenta de simulação destinada ao estudo das WSN. Todas elas estão presentes no simulador proposto.

O capítulo seguinte, Capítulo 4, apresenta de forma detalhada a concepção, implementação e forma de utilização de um núcleo de simulação (BaSS) desenvolvido com propósito geral, mas que será o alicerce para o simulador de Redes de Sensores Sem Fio (WSN-BaSS) proposto, foco do Capítulo 5.

4

BaSS
Um *framework* para o desenvolvimento de simuladores

"Na verdade, a imaginação não passa de um modo da memória,
emancipado da ordem do tempo e do espaço".
Samuel Taylor Coleridge

O desenvolvimento de qualquer tipo de simulador passa pela escolha ou pela elaboração de um componente de *software* que seja capaz de coordenar a simulação e permitir a descrição adequada do comportamento dos objetos e do sistema que será simulado. Trata-se do núcleo de simulação, que dá suporte ao desenvolvimento de simuladores e comumente apresenta-se na forma de um *framework*.

Este capítulo apresenta os principais conceitos envolvidos no desenvolvimento de um núcleo de simulação e, a partir deles, a modelagem e implementação do *framework* BaSS (*Barrier Synchronization Simulator*), cerne do capítulo e base para o desenvolvimento de um simulador para Rede de Sensores Sem Fio.

Desenvolvido na linguagem de programação Java, o BaSS é composto de um conjunto de catorze classes que implementa o modelo teórico de sincronização por barreiras aperfeiçoado. A inclusão de novos parâmetros de controle no modelo de sincronização por barreiras original tem como principal objetivo permitir uma sincronização mais refinada entre as tarefas executadas pelos objetos simulados.

O BaSS foi desenvolvido como um núcleo de simulação discreto do tipo *event-*

driven baseado em agentes, sincronizando os eventos de maneira a permitir o controle de precedência causal e que os objetos simulados atendam à chamadas geradas por outros objetos (interrupções externas) e geradas internamente aos objetos (temporizadores ou interrupções internas).

O *framework* apresentado é resultado da evolução de diversas abordagens concebidas, implementadas e testadas ao longo do desenvolvimento do projeto de construção de um simulador voltado para Redes de Sensores Sem Fio. Inicialmente foi concebido como um bloco interno a um simulador de WSN [SAL08], passando por um pacote de programação de simuladores que permite o processamento síncrono [SAL09a], sendo finalmente remodelado como um *framework* de simulação de uso geral e abrangente. Apenas a modelagem e a implementação deste último é apresentada neste capítulo, resultado final de todo o processo de desenvolvimento do núcleo de simulação.

4.1 – Definição e classificação dos núcleos de simulação

Um núcleo de simulação pode ser definido como um programa que controla a evolução de um conjunto de variáveis que mudam de estado ao longo do tempo da simulação [FUJ00]. A mudança no estado das variáveis dos objetos simulados pode ocorrer em resposta a uma mudança nas condições internas dos objetos ou induzidas por fenômenos externos perceptíveis aos objetos.

A manipulação dos eventos promotores das mudanças nos estados dos objetos, o tratamento dos eventos-resposta dos objetos frente a essas mudanças e a coordenação temporal de todos esses eventos é responsabilidade do núcleo de simulação.

De acordo com a forma com que os núcleos de simulação, ou simplesmente simuladores neste contexto, realizam a coordenação temporal da simulação, isto é, como o tempo é avançado ao longo da simulação, eles podem ser classificados como **contínuos** (*continuous simulators*) ou **discretos** (*discrete simulators*) [FUJ00][KOK00][TYA02].

Nos simuladores contínuos, o estado do sistema (conjunto dos estados de todas as variáveis do sistema) muda continuamente. Para que isso possa ocorrer, o sistema é representado através de um conjunto de equações que descrevem como as variáveis do sistema mudam ao longo do tempo. Os sistemas baseados em leis físicas bem conhecidas ou determinadas para um sistema específico podem ser simulados valendo-se dessa abordagem. O estudo do fluxo de ar em asas de aeronaves, o estudo de modelos climáticos e meteorológicos, o comportamento de circuitos elétricos e eletrônicos, entre outros, são exemplos de sistemas cuja descrição matemática permite a elaboração de simuladores

contínuos.

Os simuladores discretos, por sua vez, avançam o tempo de acordo com a própria mudança do estado do sistema. Esta mudança ocorre de forma discreta, isto é, há uma mudança abrupta ("salto") nos valores das variáveis presentes no sistema, indicando que houve uma progressão temporal. Quando a variável que determina o avanço do tempo é o próprio relógio da simulação, subclassifica-se este tipo de simulador discreto como **disparado por tempo** (*time-driven* ou *time-stepped*). Se o avanço no tempo de simulação é coordenado pela mudança de estado dos elementos simulados, diz-se que o simulador discreto é **disparado por evento** (*event-driven* ou *event-stepped*).

Assim, em um simulador *time-driven* o tempo é dividido em intervalos de mesmo tamanho Δt, denominado quadro de tempo, sendo o controle do avanço do tempo da simulação realizado quadro-a-quadro. Assim, como o tempo na simulação é incrementado de forma discreta, estando o simulador no tempo t_k, o tempo Δt será adicionado ao relógio de simulação após a computação dos eventos que ocorrem entre t_k e t_{k+1}, isto é, entre t_k e $t_k+\Delta t$.

O que determina o tamanho do quadro de tempo em um simulador *time-driven* é a natureza dos elementos simulados, isto é, o tempo mínimo com que os elementos que compõem o sistema simulado mudam seus estados no mundo real. Se o valor de Δt estabelecido for muito grande em comparação com o tempo necessário para a execução da tarefa que consome menos tempo no sistema, a simulação não poderá ser realizada de forma adequada, uma vez que existem eventos com tempo de execução menor que o quadro de tempo estabelecido para a simulação. No caso contrário, isto é, se o valor determinado para Δt for muito pequeno em comparação com o tempo necessário para a execução da tarefa que consome menos tempo no sistema, o simulador consumirá processamento desnecessário, pois em alguns quadros de tempo não haverá mudança de estado em nenhum elemento da simulação.

Nos simuladores *event-driven* a progressão temporal é controlada por uma sequência de eventos que ocorrem em determinados momentos ao longo do tempo. Um **evento** pode ser considerado um conjunto de mudanças nas variáveis de um elemento representado no simulador que possua um significado importante para o sistema, isto é, tal mudança no estado do elemento implicará mudanças futuras nos estados de outros elementos do sistema. Cada evento possui uma identificação temporal (*time stamp*) associada a ele, a qual indica o momento em que o evento ocorreu, podendo, assim, situar o evento na linha de tempo da simulação. Além do momento de ocorrência, cada evento também possui um tempo de duração da sua execução, isto é, o intervalo de tempo necessário para que o evento se complete.

Ao contrário do avanço temporal quadro-a-quadro presente nos simuladores *time-driven*, o que promove o avanço do tempo nos simuladores *event-driven* é o intervalo de tempo entre um determinado evento e o próximo evento. Neste contexto, o avanço temporal das simulações cujos elementos geram eventos esparsos de curta duração é realizado de forma mais rápida em simuladores *event-driven*, se comparados ao mesmo sistema descrito em um simulador *time-driven*. Essa diferença na velocidade de progressão temporal ocorre, pois no modelo *time-driven* o simulador fica ocioso (no sentido de não estar executando comandos relacionados diretamente às ações que estão sendo simuladas), utilizando processamento apenas para o incremento do relógio interno da simulação (passo-a-passo) até a ocorrência de uma ação de fato relevante aos elementos simulados.

Apesar do modelo teórico de simulação *event-driven* ser útil enquanto conceito para o desenvolvimento de simuladores, em [TYA02] é ressaltada a dificuldade para se implementar em ambiente computacional a interação entre dois ou mais eventos que ocorrem de forma simultânea. Um mecanismo de sincronização mais refinado seria necessário para coordenar a execução de eventos concorrentes.

Há ainda a denominação ***fully-driven****,* introduzida em [KOK00], para descrever sistemas que possuam características tanto de *event-driven* como de *time-driven*, uma vez que estes termos não são complementares. De fato, como é explicado em um exemplo apresentado em [KOK00], a simulação de uma arquitetura de computador pode requisitar tanto uma simulação *time-driven* para a execução das instruções no processador (com quadro de tempo determinado pela frequência de operação do processador) como uma simulação *event-driven* para atender aos eventos de interrupção.

O termo **simulação baseada em agentes** (*agent-based simulators*) também é pertinente à classificação de simuladores. Introduzido em [WOO97], especifica simuladores cujos elementos estão inseridos em algum tipo de ambiente e são capazes de executar ações flexíveis e autônomas no ambiente para atingir seus objetivos. A partir dessa definição, subdivide-se a classe de simuladores discretos em simuladores discretos tradicionais e simuladores discretos baseados em agentes. Os simuladores discretos tradicionais apresentam elementos que possuem apenas atributos, que nada mais são que propriedades que permitem a interação do elemento com o ambiente e com o controle da simulação. Já nos simuladores baseados em agentes, os elementos da simulação, isto é, os agentes, possuem tanto atributos como métodos, sendo estes regras que permitem a interação com o ambiente e entre os agentes.

Dessa forma, os simuladores baseados em agentes são mais completos e flexíveis, permitindo uma melhor modelagem de sistemas complexos que requerem interação entre os

elementos simulados. Além disso, a própria definição envolvendo os conceitos de atributos e métodos se enquadra perfeitamente ao paradigma de programação orientada a objetos, facilitando a sua implementação em linguagens desta natureza.

4.2 – Os tempos na simulação

Em um simulador existem duas progressões temporais que devem ser consideradas a partir do instante em que a simulação inicia até um determinado momento ou o final da simulação.

A primeira progressão temporal é chamada de **tempo de execução** (*wallclock time*[1]), que é o tempo em que o programa de simulação executa seus comandos de controle da simulação e os comandos internos dos objetos simulados [FUJ00]. Esse tempo depende da quantidade de comandos que precisam ser executados pelo simulador e da frequência do processador em que a simulação está sendo executada. Quanto maior a frequência de processamento do computador e menor a quantidade de comandos a serem executados, menor é o tempo de execução. Por outro lado, uma maior quantidade de objetos presentes no sistema simulado e um maior detalhamento na descrição comportamental destes objetos resultam em uma maior quantidade de comandos a serem executados pelo simulador, implicando em um maior tempo de execução.

A segunda progressão temporal é o **tempo de simulação** (*simulation time*), que corresponde à medição do tempo que o sistema de objetos simulado de fato teria utilizado no mundo real para executar as tarefas que estão sendo simuladas. Trata-se de uma abstração do **tempo físico** (ou **tempo real**), ou seja, é uma representação do tempo do sistema físico real [FUJ00].

O tempo de simulação não está diretamente relacionado com a capacidade de processamento do computador, mas está ligado à forma com que o sistema de objetos está representado no ambiente computacional. Quanto mais refinada a descrição do comportamento dos objetos e de seus gastos temporais associados à execução de tarefas específicas, mais próximo o tempo de simulação medido pelo simulador estará em relação ao tempo real que o sistema físico teria utilizado no mundo real.

A relação entre o tempo de execução e o tempo de simulação se dá, portanto, a

[1] O termo *wall clock time* pertence à língua inglesa e possui o significado de “tempo do relógio de parede”. Na linguagem técnica da computação o termo ainda é encontrado nas grafias *wall-clock time*, *wallclock time* ou *wall time*, referindo-se à percepção humana de passagem do tempo do início à finalização de uma tarefa que está sendo executada por um computador. Comumente este tempo é medido a partir do relógio interno dos computadores.

partir (1) da capacidade de processamento do computador utilizado para a simulação, (2) do nível de detalhamento com que os objetos simulados, seus comandos e a comunicação entre objetos são descritos e (3) da forma que o simulador organiza e dá andamento ao processo de simulação.

A viabilidade na utilização de um simulador para avaliação do comportamento de um determinado sistema olhando sob a ótica dessa relação temporal (tempo de execução *versus* tempo de simulação) depende, portanto, da natureza e complexidade do sistema a ser simulado.

A simulação de um circuito eletrônico complexo e descrito de forma bem detalhada, por exemplo, pode levar mais de 48 horas de tempo de execução para simular apenas 5 nano segundos de tempo de simulação. Mas, apesar da relação tempo de execução por tempo de simulação para esse exemplo ser muito grande, se a simulação dos 5 nano segundos de tempo de simulação for suficiente para caracterizar o comportamento do circuito, então a utilização do simulador adotado é factível.

Considerando o exemplo da análise do tempo operacional (“tempo de vida”) de uma rede de sensores sem fio, se o nível de detalhamento descritivo associado ao processo de organização e andamento da simulação no núcleo do simulador resultar em uma alta razão de tempo de execução por tempo de simulação, a simulação se torna inviável. Não se poderá contabilizar o tempo operacional da rede se, por exemplo, 1 minuto de tempo de simulação consumir 2 horas de tempo de execução. Uma rede cujo tempo operacional estimado fosse de 12 meses necessitaria de 120 anos de tempo de execução para alcançar esse resultado. Para esse tipo de problema, o procedimento de simulação deve ser o mais rápido possível e o detalhamento na descrição dos objetos necessitará de uma redução (aumento no nível de abstração descritiva), atingindo, assim, um valor baixo para a razão tempo de execução por tempo de simulação, tornando a simulação factível para o sistema.

Em simuladores do tipo tempo real (*real-time*) a relação entre o tempo de simulação e o tempo de execução é de 1:1. Trata-se de um caso particular em que a progressão de ambos os tempos deve permanecer sincronizada ao longo da simulação. Caso o conjunto *software-hardware* no qual a simulação está sendo realizada não garanta que o tempo de simulação permaneça continuamente em sincronia com o tempo de execução, não se pode caracterizar o simulador como de tempo real.

Os simuladores de tempo real geralmente são utilizados para treinamento de usuários na utilização ou controle de um determinado dispositivo físico, representado pelo simulador, antes que o usuário de fato entre em contato com o dispositivo no mundo real. Neste caso, a utilização do simulador não está diretamente relacionada à avaliação de desempenho

ou funcionamento de um sistema, mas serve como ferramenta para desenvolver e/ou avaliar a habilidade do usuário em utilizar o sistema.

4.3 – Modelos de sincronização

4.3.1 – Sincronização de Processos

A sincronização de processos que estão em execução paralela dentro de um sistema é realizada utilizando mecanismos que suspendem a execução destes processos até que uma determinada condição seja satisfeita ou um determinado estado seja atingido. Esses mecanismos de sincronização podem ser classificados como barreiras [AXE86][ARE89][FUJ00] ou travas (*locks*) [ARE89][FAN05], de acordo com o seu objetivo no sistema.

As barreiras são utilizadas como um mecanismo que sincroniza a progressão na execução dos comandos de um grupo de processos que estão sendo executados em paralelo. Os comandos dos processos são executados em paralelo até que a próxima instrução de barreira (pertencente ao conjunto de comandos de cada processo) seja atingida em cada um dos processos pertencentes ao grupo. A partir deste instante, todos os processos são desbloqueados e continuam sendo executados em paralelo até que uma nova instrução de barreira seja atingida. Esse procedimento de bloqueio e desbloqueio continua até o último conjunto de barreiras existente nos processos.

As travas, por sua vez, são utilizadas para garantir que um processo tenha acesso exclusivo a um determinado recurso do sistema (operação atômica), impedindo que os demais processos executem ações simultaneamente sobre o mesmo recurso. Existem três tipos de travas [ARE89][FAN05]: as exclusões mútuas (*mutex*), os semáforos (*semaphores*) e os *spinlocks*.

As exclusões mútuas, a mais simples das travas, são bloqueios realizados nos processos de forma a garantir que somente um processo tenha direito de acesso a um determinado recurso. Por exemplo, quando dois ou mais processos desejam enviar dados por uma porta serial, um *mutex* permite que apenas um processo o faça por vez, fazendo com que os demais processos fiquem bloqueados até que o processo que detém o acesso à porta serial a libere.

Os semáforos também garantem que apenas um processo acesse um determinado recurso por vez, mas este mecanismo é utilizado quando o sistema possui mais de um recurso do mesmo tipo e há uma quantidade maior de processos (se comparado à quantidade de recursos do mesmo tipo) tentando utilizá-los. Supondo que um sistema possua N recursos do

tipo X e esses recursos sejam compartilhados por M processos (M>N). Se uma quantidade K de processos (K>N) deseja acessar um recurso do tipo X, mas apenas N recursos estão disponíveis no sistema, então há de se implementar uma política que distribua esses N recursos para os N processos e bloqueie os outros K-N processos até que pelo menos um recurso do tipo X seja liberado. Essa política de distribuição de recursos de um mesmo tipo entre processos com bloqueio dos processos excedentes é implementada pelo mecanismo de semáforo.

Spinlocks são travas semelhantes aos *mutex*, mas são mais eficientes, pois não envolvem espera. Os processos que estão aguardando a liberação do recurso compartilhado não ficam "esperando" uma notificação do sistema para poderem acessar o recurso, mas ficam observando a liberação do recurso e em seguida garantem o seu acesso. Por isso são denominadas travas do tipo "*spin*", pois ficam "virando" ao redor do recurso até que o mesmo seja liberado.

4.3.2 – Sincronização em simuladores

Os mecanismos de sincronização apresentados no item 4.3.1 estão relacionados com o tempo, por isso são utilizados para a sincronização de processos. Mas no contexto de simuladores, o que se busca é um tipo de sincronização como o mecanismo de barreiras, pois este permite o controle do avanço sincronizado dos processos.

O mecanismo de barreira é, portanto, o mais adequado para ser utilizado como um marcador de progressão temporal entre processos que executam de forma paralela em um sistema. Em simuladores do tipo *time-driven*, o tempo associado a cada barreira é determinado por um relógio global, isto é, as barreiras criadas em intervalos de tempo de simulação regulares determinam a progressão temporal no sistema simulador. Em simuladores do tipo *event-driven*, as barreiras são determinadas de acordo com os eventos que ocorrem no sistema, ou seja, os eventos são os responsáveis pela progressão temporal.

Com relação à sincronização de eventos em uma simulação, ela pode ser realizada de forma conservativa ou otimista. No modelo conservativo [CHA79][BRY77], busca-se garantir que não haja violação de causalidade, isto é, um evento e_1 com início estabelecido no tempo t não deve ser processado na existência de um evento e_2 com tempo de início menor que t e que tenha influência sobre o evento e_1. Já o modelo otimista [FUJ01] permite que a execução de eventos interdependentes ocorra de forma parcialmente desordenada do ponto de vista temporal, mas mecanismos de detecção e recuperação devem ser implementados para corrigir as inconsistências geradas.

O modelo otimista possui como vantagem a possibilidade de maximizar o

paralelismo no processamento dos eventos. Contudo, o mecanismo de recuperação (*rollback*) pode ser custoso em termos computacionais. [FUJ90] propõe um algoritmo de *rollback* baseado no mecanismo de *Time Warp* [THO98]. Esse algoritmo prevê o envio de uma anti-mensagem relacionada à mensagem que foi processada prematuramente com o objetivo de desfazer o evento processado. Pode acontecer que outras mensagens tenham sido processadas em decorrência do evento prematuramente executado, causando a geração de outras anti-mensagens. A operação de *rollback*, portanto, pode ser um limitante para o desempenho da simulação, uma vez que a recuperação envolve processamentos adicionais que muitas vezes se propagam além de uma única recuperação.

Os algoritmos de sincronização conservativa, por sua vez, não usufruem de forma plena do processamento paralelo dos computadores, uma vez que há a avaliação do próximo evento a ser processado e apenas esse evento é executado. Não existem dois ou mais processos com comandos sendo executados simultaneamente.

Apesar do mecanismo de barreiras ter sido desenvolvido no contexto de processamento paralelo, ele pode ser adaptado para realizar a execução serial de processos que simulam a execução paralela, garantindo, assim, uma sincronização conservativa.

Contudo, o principal problema da utilização de barreiras é a possibilidade da simulação entrar em situação de bloqueio permanente (*deadlock*), uma vez que o controle do bloqueio e desbloqueio dos processos deve ser realizado de forma minuciosa e depende fortemente da linguagem de programação adotada na sua implementação.

Apesar do possível problema de *deadlock*, o modelo de sincronização conservativa por barreiras foi o eleito para compor o arcabouço do núcleo de simulação desenvolvido pela sua simplicidade de implementação e controle temporal, além de não exigir um processamento paralelo real.

Segue um detalhamento do mecanismo de sincronização por barreiras na sua concepção original, que será adaptado às exigências do núcleo de simulação proposto no item 4.4.

4.3.3 – Modelo sincronização por barreiras

A sincronização controlada por barreiras (*Barrier Synchronization*) é uma técnica utilizada em computação distribuída destinada ao controle centralizado de processos que executam de forma paralela [NIC95]. A implementação do modelo de sincronização por barreiras em sistemas geralmente é realizada por software, mas há soluções em hardware que utilizam barreiras como mecanismo de sincronização [HIL98].

Quando aplicada ao contexto de simulação, a sincronização por barreiras pode ser definida como um método de sincronização conservativa que possibilita o controle síncrono dos comandos de processos em execução simultânea pelo estabelecimento de barreiras temporais [FUJ00].

No modelo teórico de sincronização por barreiras, todos os processos em execução simultânea (execução pseudo-paralela em ambiente computacional monoprocessado) executam seus comandos individuais de forma sequencial até que um comando de barreira temporal seja atingido em cada processo. Na barreira, a execução do processo é bloqueada. Um controlador de sincronismo global é responsável pela coordenação de todos os processos, sendo que o avanço na execução dos processos só ocorre quando todos os processos estejam bloqueados na barreira de controle temporal com a mesma marca temporal (*timestamp*). Neste ponto, os processos são desbloqueados e continuam a executar os comandos imediatamente após o comando da barreira [TZE05].

Todos os comandos presentes entre duas barreiras dentro de um processo são executados de forma pseudo-paralela com os comandos de outros processos que também se encontram no mesmo intervalo entre barreiras (entre *timestamps* iguais), isto é, na mesma janela temporal.

A Figura 4.1 apresenta em pseudocódigo a sequência de comandos de cinco processos, representados na figura como *threads* (processos simples): *Thread A, Thread B, Thread C, Thread D* e *Thread E*.

Thread A

```
ThreadA {
  run() {
    A1;
    A2;
    A3;
    A4;
    barrier;
    A5;
    A6;
    barrier;
    A7;
    A8;
    A9;
    barrier;
    A10;
    barrier;
    A11;
    ...
  }
}
```

Thread B

```
ThreadB {
  run() {
    B1;
    B2;
    B3;
    B4;
    B5;
    barrier;
    B6;
    B7;
    B8;
    barrier;
    B9;
    B10;
    barrier;
    B11;
    B12;
    B13;
    barrier;
    B14;
    ...
  }
}
```

Thread C

```
ThreadC {
  run() {
    C1;
    C2;
    barrier;
    C3;
    C4;
    C5;
    C6;
    barrier;
    C7;
    C8;
    C9;
    C10;
    barrier;
    C11;
    C12;
    barrier;
    C13;
    ...
  }
}
```

Thread D

```
ThreadD {
  run() {
    D1;
    D2;
    D3;
    barrier;
    D4;
    barrier;
    D5;
    D6;
    D7;
    D8;
    D9;
    barrier;
    D10;
    D11;
    barrier;
    D12;
    ...
  }
}
```

Thread E

```
ThreadE {
  run() {
    E1;
    E2;
    E3;
    E4;
    E5;
    barrier;
    E6;
    E7;
    E8;
    barrier;
    E9;
    barrier;
    E10;
    E11;
    barrier;
    E12;
    ...
  }
}
```

Figura 4.1 – Pseudocódigo de cinco *threads* (*Thread A*, *Thread B*, *Thread C* e *Thread D*) sincronizadas por barreira. O comando *barrier* determina as barreiras, delimitando janelas temporais.

A execução dessas *threads* é controlada pelo mecanismo de sincronização por barreiras, sendo as barreiras definidas pelo comando *barrier*. A Figura 4.2 ilustra a execução dos comandos dessas cinco *threads:* as *threads* que terminam a execução de seus comandos antes que todas as outras *threads* atinjam a próxima barreira são bloqueadas até que todas as *threads* atinjam a barreira, sendo, somente assim, desbloqueadas.

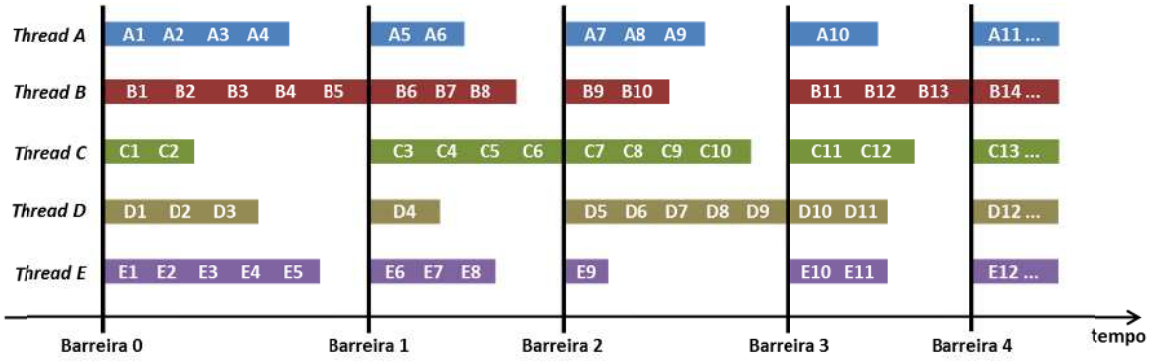


Figura 4.2 – Execução das cinco *threads* da Figura 4.1 controladas por sincronização de barreira. Ao atingir uma instrução de barreira a *thread* é bloqueada e deve aguardar até que as outras *threads* cheguem à mesma barreira temporal. Um sincronizador global controla o desbloqueio das *threads*. Assim, na janela temporal entre as barreiras 0 e 1 todas as *threads* devem aguardar a *Thread B* terminar a execução do comando *B5* para continuarem a executar. Entre as barreiras 2 e 3 todas as *threads* devem aguardam o fim da execução do comando *D9* da *Thread D* para serem desbloqueadas e continuarem a execução de seus comandos.

4.3.3.1 – Tipos de barreiras

Os tipos de barreiras apresentados na literatura de computação distribuída diferem essencialmente quanto ao modo de verificação se todas as *threads* estão bloqueadas para que o avanço no processamento, isto é, o desbloqueio das *threads*, possa ser realizado de forma consistente. O objetivo maior é reduzir o número de troca de mensagens e verificações de sinalizações entre as *threads*.

O modelo de barreiras descrito no item 4.3.1 e ilustrado na Figura 4.2 é o modelo clássico de barreira, sendo também é conhecido como **Barreira Centralizada** [FUJ00] ou **Barreira Central** [BAL03]. Uma maneira para se ter controle de quando todas as *threads* atingiram a próxima barreira é a utilização de uma variável inteira compartilhada por todas as *threads*, a qual é atribuída como valor inicial a quantidade de *threads* que estão sendo sincronizadas. Conforme as *threads* vão finalizando as suas atividades e atingindo a próxima barreira, elas decrementam o contador e, assim, o controlador de sincronismo global sabe que deve desbloquear as *threads* quando o valor do contador chegar a zero, reinserindo o valor inicial (quantidade de *threads*). Apesar de essa abordagem requerer a utilização de semáforo para acesso à variável compartilhada (caso contrário podem ocorrer inconsistências na leitura/escrita do seu conteúdo), ela evita que o controlador de sincronismo necessite consultar

o estado de todas as *threads* para decidir quando desbloqueá-las.

Existe a possibilidade de agrupar *threads* que compartilham os mesmos recursos e podem ser executadas isoladamente de outros grupos de *threads*. Cada grupo de *threads* teria um contador particular, mas que estaria relacionado a um contador global (que de fato controla o avanço nas barreiras) através de uma estrutura hierárquica de árvore. Devido a esta estrutura, este tipo de arranjo de barreiras ficou conhecido como **Barreira de Árvore** (*Tree Barrier*) [YEW87]. Tal agrupamento de *threads* permite que cada grupo seja executado em um determinado núcleo de processamento, diminuindo, assim, o tempo de processamento do conjunto total de *threads.*

Outro tipo de barreira é conhecido como **Barreira Borboleta** (*Butterfly Barrier*) ou Barreira de Brooks (*Brooks' Barrier*) [BRO86]. Trata-se de uma maneira de organizar as *threads* em uma estrutura de árvore binária (altura $log_2 N$, onde N é o número de *threads* a serem sincronizadas) na qual em cada nível da árvore (passo), partindo das folhas, ocorre a execução das *threads* em processadores diferentes. Devido à organização binária, a cada passo um conjunto de *threads* múltiplo de dois permanece em sincronismo entre si até atingir a raiz, onde todas as *threads* envolvidas atingem a barreira de sincronismo. A vantagem desse procedimento é que a informação de sincronismo não precisa ser disseminada de forma generalizada [FUJ00], uma vez que é garantida a sincronia das *threads* pertencentes ao mesmo passo.

A **Barreira de Disseminação** (*Dissemination Barrier*) [HEN88] é um aprimoramento da Barreira Borboleta, pois no caso da quantidade de *threads* não ser uma potência de dois, a Barreira Borboleta pode se tornar ineficiente [BAL03]. A Barreira de Disseminação deixa de lado a ideia do sincronismo de conjuntos de *threads* múltiplos de dois para utilizar um mecanismo de sinalização (execução ou em espera). Nesta abordagem formam-se pares de *threads* que trocam suas sinalizações apenas entre si. A cada etapa (*round*), novos pares são formados de maneira que ao final de $log_2 N$ etapas (onde N é o número de *threads*), todas as *threads* encontram-se bloqueadas e podem ser desbloqueadas com segurança.

A **Barreira de Torneio** (*Tournament Barrier*) [HEN88] utiliza o princípio de seleção de pares de *threads* como se fossem disputar uma partida. Apesar disso, os "jogos" não são competitivos, sendo a escolha do vencedor realizada de forma aleatória ou determinada à priori. Sempre haverá um ganhador e um perdedor para cada pseudo-partida, então a cada conjunto de partidas as *threads* "ganhadoras" da etapa anterior "jogam" entre si, assim como as *threads* "perdedoras". Ao final do torneio todas as *threads* atingirão a barreira. Cada torneio é composto de $log_2 N$ etapas, como nos casos anteriores, mas o algoritmo de sincronização utilizando Barreira de Torneio é mais simples e menor que o de Barreira Butterfly e o Barreira de

Disseminação, sendo, portanto mais eficiente [BAL03].

Os cinco tipos de barreiras descritos de forma sucinta não apresentam todos os elementos de sincronização necessários ao núcleo de simulação proposto. As diferentes abordagens dessas barreiras foram concebidas no contexto de processamento paralelo, não tendo como foco principal o desenvolvimento de simuladores. Contudo, o conceito de barreira é adotado como mecanismo central de sincronização do núcleo de simulação desenvolvido, acrescentando alguns elementos que transformam o conceito simples em um mecanismo de sincronização aprimorado e flexível.

4.4 – O núcleo de simulação proposto

No mundo real os objetos animados executam de forma independente algumas tarefas que dependem apenas de seu estado intrínseco e possivelmente de variáveis ambientais a que estão submetidos. Essas tarefas podem ser repetitivas (cíclicas) ou possuírem uma sequencia de etapas com início, meio e fim. Mas esses objetos também podem executar tarefas que dependam da interação com outros objetos. Essa interação necessita que um objeto seja capaz de se comunicar com outro objeto (ou outros objetos) e esse, por sua vez, seja capaz de atender à requisição do objeto externo. O atendimento à requisições externas possivelmente requer que o objeto que foi requisitado interrompa a execução de suas tarefas internas para poder responder à requisição.

Os sistemas orgânicos e mecânicos reais, aqui e em sistemas computacionais representados como objetos, são capazes, portanto, de executar suas tarefas e atender às requisições de outros objetos, ou ainda atender a requisições de si mesmo (requisições internas), interrompendo sua tarefa principal. Esse comportamento ocorre de forma natural no mundo real, no qual os objetos executam ações simultâneas, isto é, em paralelo, e pausam suas tarefas (desviam sua atenção) quando são requisitados por outros objetos.

O *framework* de simulação proposto, cuja fundamentação teórica e implementação são apresentadas neste capítulo, é composto por classes desenvolvidas na linguagem de programação Java que implementam o controle síncrono de execução dos comandos internos dos objetos e coordenam o atendimento à requisições (interrupções) originadas de outros objetos e/ou originadas de dentro do próprio objeto. A sincronização temporal dos objetos é realizada tendo como base o modelo de sincronização por barreiras e para permitir que os objetos simulados realizem avanços temporais não padronizados, o núcleo de simulação é do tipo *event-driven*.

Segue uma descrição detalhada do *framework* desenvolvido.

4.4.1 – Requisitos do comportamento dos objetos na simulação

Como foi visto, para atender aos requisitos de simulação propostos os objetos representados devem ser capazes de executar suas ações de forma simultânea, manter o sincronismo temporal entre si e responder a requisições externas e internas. O mecanismo de sincronização a que os objetos serão submetidos deve ser capaz de:

(1) Manter a execução simultânea (pseudo-paralela) dos comandos internos pertencente a cada objeto;

(2) Sincronizar a execução dos comandos dos objetos de forma conservativa utilizando um controle de progressão temporal *event-driven* baseado em agentes;

(3) Em um determinado instante de simulação, garantir a precedência causal entre comandos, pertencentes a objetos distintos, que deveriam ser executados de forma simultânea no mundo real;

(4) Tratar solicitações externas (interrupções geradas por outros objetos) e internas (interrupções e sequências de comandos temporizadas do próprio objeto), permitindo o desvio na execução da sequencia principal de comandos dentro do objeto em favor da execução de outras sequencias de comandos solicitadas (tratamento da interrupção), voltando à sequencia principal ao término.

Os quatro requisitos enumerados podem ser alcançados tendo como base a estrutura de *threads* nativa da linguagem de programação Java e o modelo de sincronização por barreiras. Ambos precisam de adaptações para se atingir com rigor os requisitos apresentados.

4.4.2 – Pseudo-paralelismo na linguagem Java

Em ambientes computacionais monoprocessados, o paralelismo na execução de tarefas não pode ser criado de forma semelhante ao mundo real, mas algumas estratégias de pseudo-paralelismo podem ser empregadas com o intuito de se conseguir o efeito aparente de execução simultânea dos comandos executados pelos objetos simulados.

A abordagem pseudo-paralela em computação nada mais é do que o processamento intercalado de trechos de comandos pertencentes a objetos distintos, resultando em um aparente paralelismo na execução dos comandos desses objetos. Devido à alta frequência de execução das instruções no computador e a limitada percepção humana

diante essa frequência, o pseudo-paralelismo pode ser empregado em diversas aplicações computacionais, sem perda na qualidade dos resultados obtidos.

A intercalação dos trechos de comandos pertencentes a objetos distintos pode ser realizada por diferentes estratégias para se conseguir o pseudo-parelelismo. Uma primeira possibilidade é executar um comando de cada objeto por vez, dando a mesma oportunidade de execução para cada objeto e passando a permissão de execução de objeto para objeto, de forma cíclica. Uma segunda possibilidade é agrupar nos objetos os comandos que possuam tempos de execução semelhantes e executá-los em blocos, passando de objeto a objeto a permissão de execução. Pode-se, ainda, criar prioridades de execução para os objetos, de forma que os objetos que possuam tarefas consideradas mais "importantes" executem seus comandos com maior frequência que os comandos de outros objetos.

Na linguagem de programação Java, a execução de trechos de programas de forma pseudo-paralela pode ser conseguida com a utilização de *threads*, nativas nessa linguagem de programação. Uma *thread* é uma subdivisão de um processo que executa seus comandos de forma autônoma e concorrente com outras *threads* existentes no mesmo processo. A utilização de múltiplas *threads* executando de forma pseudo-paralela pode ser realizada de forma a atender ao **requisito (1)**.

Contudo, um problema verificado na utilização dessa solução é a sincronização das sequências de comandos executados no interior das *threads* em relação às outras *threads*.

Blocos de comandos sequenciais pertencentes às *threads* de Java são executados de forma pseudo-paralela pela JVM (*Java Virtual Machine*), mas a decisão de parada após a execução de um comando pertencente a um método em execução do objeto O_1, mudança para a execução de outro bloco de comandos de um objeto O_2 e retorno à execução do bloco de comandos seguinte do objeto O_1 é feita exclusivamente pela JVM. A determinação da quantidade de comandos (bloco de comandos) executados por vez em cada *thread* também é feita de acordo com a arquitetura interna da JVM.

A Figura 4.3 apresenta a declaração de duas *threads* em pseudo-código e uma execução para dois objetos instanciados, um da classe *Thread F* e outro da classe *Thread G*. A execução dos comandos ocorre a partir dos métodos definidos com o nome *run()* em cada uma das *threads*. Considerando de forma hipotética que cada comando (*F1, F2, F3, F4, G1, G2* e *G3*) consuma uma mesma unidade de tempo para executar, dever-se-ia ter como sequência de execução *F1* e *G1* no tempo *t*, *F2* e *G2* no tempo *t1*, *F3* e *G3* no tempo *t2* e *F4* no tempo *t3*, resultando na sequência: $[F1//G1]_t \Rightarrow [F2//G2]_{t1} \Rightarrow [F3//G3]_{t2} \Rightarrow [F4]_{t3}$. Como em ambiente computacional monoprocessado não é possível executar os comandos da *Thread F* ao mesmo tempo que os comandos da *Thread G*, os pares de comandos que deveriam executar

simultaneamente ocorreriam de forma sequencial (*F1* antes de *G1,* ou *G1* antes de *F1*, por exemplo). Uma possível sequência de execução seria: $[F1 \Rightarrow G1]_t \Rightarrow [G2 \Rightarrow F2]_{t1} \Rightarrow [F3 \Rightarrow G3]_{t2} \Rightarrow [F4]_{t3}$.

Thread F

```
ThreadF {
    run() {
        F1;
        F2;
        method1();
    }

    method1() {
        F3;
        F4;
    }
}
```

Thread G

```
ThreadG {
    run() {
        G1;
        G2;
        G3;
    }
}
```

Uma possível sequência de execução

F1
⇩
F2
⇩
G1
⇩
F3
⇩
G2
⇩
G3
⇩
F4

Figura 4.3 – Exemplo de duas *threads* (*Thread F* e *Thread G*) sendo executadas simultaneamente. A sequência de execução dos comandos de uma *thread* sempre é respeitada de acordo com a sequência estabelecida pelo programador, mas a sequência relativa de execução dos comandos das *threads* em execução pseudo-paralela não pode ser pré-determinada, sendo estabelecida de acordo com políticas internas da JVM.

Contudo, além da sequência de execução de comandos entre as *threads* não poder ser estabelecida pelo programador, devido a política de execução das *threads* na JVM, a quantidade de comandos de cada *thread* executados por vez pode variar.

A sequência de execução dos comandos varia de acordo com a quantidade de *threads* e os tipos de *threads* no *pool* de *threads* (*threads em execução*). Portanto, apesar da sequência de comandos dentro de uma *thread* sempre ser executada respeitando a sequência estabelecida pelo programador, a sequência relativa de execução entre comandos de *threads* distintas não pode ser programada com estruturas ou comandos nativos da linguagem Java e, sendo assim, não pode ser estabelecida pelo programador.

Para se resolver esse problema, intrínseco ao modo como a JVM escalona as *threads*, o mecanismo de sincronização por barreiras pode ser utilizado.

4.4.3 – O controle do tempo na sincronização por barreiras

A sincronização por barreiras do modo com apresentada em seu modelo teórico pode ser utilizada diretamente na elaboração de simuladores *time-driven*, considerando cada barreira como o avanço de um determinado Δt no tempo de simulação. Nessa abordagem, as *threads* teriam instruções de barreira entre seus comandos, sendo que a existência de um bloco de comandos entre duas barreiras indicaria que aqueles comandos utilizariam o tempo de simulação de Δt ao serem executados.

A Figura 4.4 apresenta o pseudo-código de duas *threads* (*Thread H* e *Thread I*), com quatro instruções de barreira cada. Supondo que cada barreira corresponda ao intervalo de 1s de tempo de simulação (Δt=1s), a simulação apresentada teria um total 4s de tempo de simulação, sendo que no intervalo de 0 a 1s seriam executados os comandos *H1*, *I1* e *I2*, no intervalo de 1 a 2s seriam executados os comandos *H2*, *H3*, *I3* e *I4*, no intervalor de 2 a 3s seria executado o comando *H4* e finalmente no intervalo de 3 a 4s seriam executados os comandos *H5*, *H6*, *H7* e *I5*.

Independente do tempo de execução necessário à execução de cada comando no ambiente computacional, a marcação do tempo de simulação é cadenciada pelas instruções de barreira.

Apesar dessa implementação de barreiras *time-driven* poder ser utilizada para alguns tipos de simulação, ela não é adequada para simulações que apresentem *threads* com comandos que executem com intervalos de tempo de simulação Δt muito diferentes. Uma vez que o valor de Δt é determinado pelo menor intervalo de tempo de simulação requerido, as *threads* que apresentam comandos com intervalos de execução grandes (se comparado ao intervalo Δt) necessitarão de uma grande quantidade de barreiras (comandos *barrier*) nulas, isto é, sem comandos entre elas.

Thread H

```
ThreadH {
  run() {
    H1;
    barrier;
    H2;
    H3;
    barrier;
    H4;
    barrier;
    H5;
    H6;
    H7;
    barrier;
  }
}
```

Thread I

```
ThreadI {
  run() {
    I1;
    I2;
    barrier;
    I3;
    I4;
    barrier;
    barrier;
    I5;
    barrier;
  }
}
```

Figura 4.4 – Considerações sobre o comando de barreira. Considerando cada barreira (comando *barrier)* como um delimitador de 1s de tempo de simulação, os comandos presentes entre barreiras possuem um tempo de simulação 1s, independente do tempo de execução utilizado em cada comando. A execução síncrona da *Thread H* e a *Thread I* representa um total de 4s de tempo de simulação.

Essa quantidade de barreiras utilizadas uma após a outra sem a existência de comandos entre elas não é apenas não prática para o programador da *thread*, mas, como já foi visto no item 4.1, faz com que o controle da simulação execute uma quantidade de comandos de desbloqueio e bloqueio das *threads* (a cada instrução de barreira) sem haver comandos específicos dos objetos a serem executados entre barreiras, gastando um processamento que

poderia ser utilizado de forma mais relevante à simulação.

A primeira adaptação do modelo teórico de sincronização por barreiras tem como objetivo reduzir o tempo de processamento da simulação. Cada *thread* teria autonomia para criar as suas próprias barreiras temporais, independente da existência das barreiras das outras *threads* e também não dependendo de um *tick* global da simulação. Isso descaracterizaria o simulador como sendo do tipo *time-driven*, transformando-o em um simulador do tipo *event-driven*.

Assim, ao invés de existir uma única barreira lógica de controle temporal em que todas as *threads* deveriam parar, coexistiriam múltiplas barreiras estabelecidas de acordo com o avanço temporal específico de cada *thread* e, sendo assim, não há a obrigatoriedade de todas as *threads* pararem em todas as barreiras. Cada *thread* seria bloqueada somente nas barreiras que ela mesma tenha criado.

A Figura 4.5 apresenta em pseudo-código a *Thread J* e a *Thread K, além d*a execução dos comandos das mesmas ao longo do tempo. Nesse exemplo, as barreiras individuais são determinadas pelo método *wait()* que recebe como parâmetro uma quantificação temporal referente ao tempo de simulação requerido para que o comando (ou bloco de comandos) imediatamente anterior à barreira seja executado.

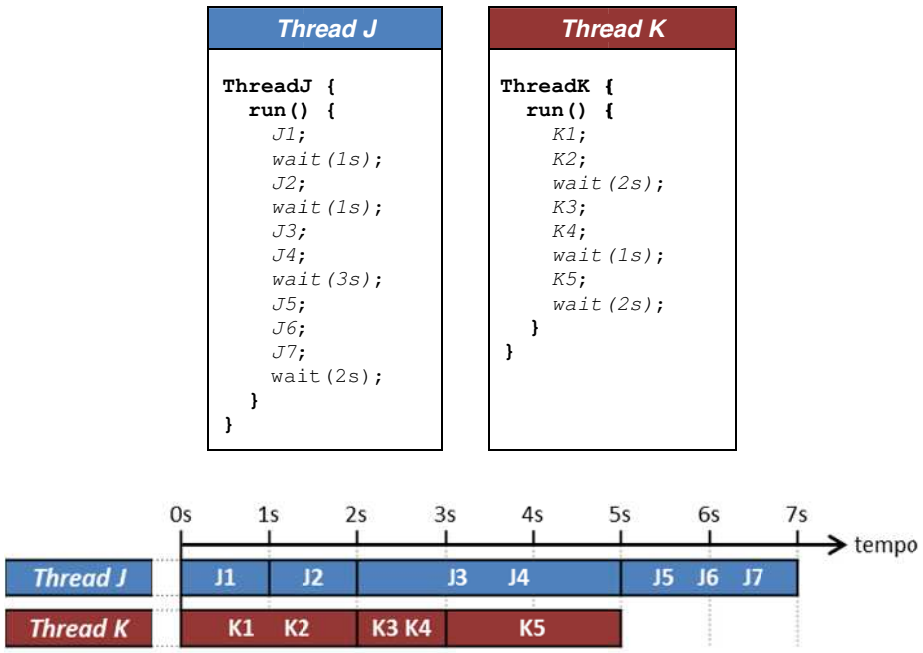


Figura 4.5 – Implementação de múltiplas barreiras. Cada comando ou bloco de comandos imediatamente acima do comando de barreira *wait()* requer a quantidade de tempo de simulação especificada no parâmetro desse método para ser executado no tempo de simulação.

A primeira barreira definida na *Thread J* (*wait(1s)*) determina que o comando J*1* necessita de 1s de tempo de simulação; o simulador executa o comando *J1* e cria uma barreira para a *Thread J* em 1s. A primeira barreira definida para a *Thread K* especifica que os comandos *K1* e *K2* utilizam 2s de tempo de simulação; o simulador executa os comandos *K1* e *K2* e bloqueia a *Thread K* na barreira temporal de 2s. A simulação segue desbloqueando e bloqueando as *threads* conforme as barreiras que foram estabelecidas pelo programador.

Como a progressão temporal não mais é realizada de forma discreta, isto é, o avanço temporal não é mais determinado por um intervalo de tempo constante Δt, as *threads* não mais são bloqueadas/desbloqueadas desnecessariamente. No exemplo da Figura 4.5, a *Thread J* não parou nas barreiras 3s, 4s e 6s e a *Thread K* não parou nas barreiras 1s e 4s. Além disso, as barreiras de 4s e 6s nem existem de fato, uma vez que nenhuma das duas *threads* as criou. E mais, a linha temporal apresentada com a divisão de segundos na Figura 4.5 não faz sentido do ponto de vista interno de uma simulação *event-driven*, visto que não existe uma barreira 4s, nem uma barreira 6s, assim como não existe uma barreira 3,75s, nem uma barreira 5,12s. O que existe são indicações temporais (as barreiras) do momento em que os comandos precedentes a essas indicações terminam de ser executados no tempo de simulação.

Cada conjunto de comandos entre barreiras pode, portanto, ser definido como um evento, caracterizando o avanço temporal da simulação utilizando múltiplas barreiras como um avanço determinado por evento (*event-driven*). Pela definição de evento (item 4.1), se a sequência de comandos entre as barreiras de um objeto não tiver influência com os eventos de outros objetos, a barreira posterior ao grupo de comandos do objeto pode ser deslocada, a depender única e exclusivamente da relação existente entre os objetos da simulação. Obviamente o programador pode utilizar quantas barreiras ele julgar necessário, mas a cada introdução de uma nova barreira, o tempo de execução da simulação aumenta. O entendimento da inter-relação entre os objetos do sistema simulado pode, portanto, ajudar a reduzir ainda mais a quantidade de barreiras temporais necessárias à simulação, reduzindo, assim, o tempo de execução da simulação.

Pela utilização de barreiras múltiplas, a sincronização do tempo de simulação dos comandos dos objetos pode ser realizada de forma conservativa e utilizando avanços temporais determinados pelos eventos. Sendo a interação entre objetos factível no contexto de programação orientada a objetos, isto é, os objetos poderem ser classificados como agentes por possuírem propriedades e métodos, a simulação utilizando objetos com a capacidade de criar múltiplas barreiras se caracteriza como *event-driven* baseado em agentes. Dessa forma, **o requisito (2)** é atendido.

4.4.4 – Sincronização por barreiras com prioridades

No esquema de barreiras múltiplas apresentado, algumas barreiras temporais podem ser compartilhadas por mais de uma *thread*, como é o caso da barreira do 2s no exemplo da Figura 4.5. Nesta situação, os comandos *J3* e *J4* deveriam ser executados simultaneamente aos comandos *K3* e *K4*, pois ambos os blocos de comandos iniciam suas execuções a partir do tempo de simulação de 2s. Contudo, isso não tem como ocorrer em uma arquitetura computacional monoprocessada.

Como apenas um comando pode ser executado por vez, sendo isso um limitante que não pode ser modificado por estratégias de programação, ao invés de se buscar uma solução em uma arquitetura multiprocessada, que por sua vez também é limitada pelo número de processadores disponíveis, pode-se identificar em que situações o não paralelismo efetivo provoca erros interpretativos ou de execução em um ambiente de programação pseudo-paralelo.

O fato de um comando *cmd1* ocorrer antes de um comando *cmd2* ou um comando *cmd2* ocorrer antes de um comando *cmd1*, sendo que eles pertencem a *threads* distintas e deveriam ocorrer simultaneamente, possui implicações computacionais apenas se houver uma dependência causal entre os comandos envolvidos, isto é, se o comando *cmd2* depender do resultado do comando *cmd1*, ou vice-versa.

O modelo de sincronização por barreiras não prevê uma forma de controle de precedência causal entre comandos de processos (*threads*) distintos. Uma segunda adaptação se faz necessária no modelo de sincronização por barreiras para que a precedência causal entre comandos que deveriam ser executados simultaneamente possa ser estabelecida.

Um esquema de prioridade entre as *threads* pode ser utilizado para determinar a sequência de execução de comandos pertencentes à *threads* bloqueadas em barreiras de mesmo valor temporal. Se o comando *cmd2* envolver a utilização de um dado gerado pelo comando *cmd1* e ambos os comandos deveriam ser executados simultaneamente, o comando *cmd1* necessariamente deve ser executado antes do comando *cmd2*. Utilizando um esquema de prioridades, a prioridade da *thread* que possui o comando *cmd1* deve ser maior que a prioridade da *thread* que possui o comando *cmd2*, fazendo com que o coordenador da simulação desbloqueie a *thread* com o comando *cmd1* antes da *thread* com o comando *cmd2*.

Dessa foram, a sequência de execução dos comandos que deveriam ser executados simultaneamente (*threads* bloqueadas em barreiras de mesmo tempo de simulação) pode ser estabelecida pelas prioridades atribuídas às *threads* que contém os

comandos, permitindo ao programador estabelecer uma sequência de execução entre comandos pertencentes à *threads* distintas de tal forma a garantir a precedência causal.

Cada *thread* criada deve possuir um número inteiro associado a ela que identifica a sua prioridade perante as demais *threads*. A Figura 4.6 apresenta cinco *threads* com prioridades e com barreiras definidas pelo comando *wait()*: a *Thread L* com prioridade 0 (menor), a *Thread M* com prioridade 3, a *Thread N* com prioridade 1, a *Thread O* com prioridade 4 (maior) e a *Thread P* com prioridade 2.

A Figura 4.7 mostra a sequência de execução paralela dos comandos das *threads* da Figura 4.6, enquanto que na Figura 4.8 a sequência de execução dos comandos é apresentada na forma com que de fato será executada pelo controlador da simulação, isto é, apenas uma *thread* sendo executada por vez. As prioridades atribuídas às *threads* resolvem o conflito no caso de duas ou mais *threads* se encontrarem bloqueadas em barreiras temporais de mesmo valor. O controlador da simulação desbloqueia as *threads* em ordem decrescente de prioridade.

Assim, se houver múltiplos bloqueios em barreiras temporais de mesmo valor, os comandos da *Thread O* (a de maior prioridade) executarão antes dos comandos da *Thread M*, que por sua vez executarão antes dos comandos da *Thread P*, em seguida serão executados os comandos da *Thread N* e por último serão executados os comandos da *Thread A* (a de menor prioridade).

Supondo que os comandos *O2*, *O4* e *O6* sejam envios de mensagem da *Thread O* para as *Thread M* e esta receberá e processará as mensagens pelos comandos *M1*, *M3* e *M6*, respectivamente. Sendo um par envio/recepção de mensagens um evento simultâneo, se o esquema de prioridade não existisse, possivelmente, por exemplo, o comando *M1* poderia ser executado antes do comando *O2*, não havendo obediência a precedência causal e o comando *M1* não teria a sua disposição a mensagem enviada pelo comando *O2* para realizar o processamento da mesma.

No caso de *threads* que possuam a mesma prioridade, não há garantia da ordem de execução entre elas. *Threads* com mesma prioridade, portanto, só devem ser utilizadas quando não há restrições de precedência causal entre as instruções destas *threads*.

A inclusão de prioridades no modelo de sincronização por barreiras possibilita ao programador controlar a sequência de desbloqueio de *threads* bloqueadas em barreiras de mesmo valor temporal, permitindo um controle de precedência causal entre comandos pertencentes à *threads* distintas, atendendo ao **requisito (3)**.

Thread L — Prioridade: **0**

```
ThreadL {
  run() {
    L1;
    L2;
    wait(1s);
    L3;
    wait(1s);
    L4;
    L5;
    L6;
    wait(2s);
    L7;
    L8;
    wait(1s);
    L9;
    wait(1s);
  }
}
```

Thread M — Prioridade: **3**

```
ThreadM {
  run() {
    M1;
    M2;
    wait(2s);
    M3;
    M4;
    M5;
    wait(1s);
    M6;
    wait(1s);
    M7;
    M8;
    wait(2s);
  }
}
```

Thread N — Prioridade: **1**

```
ThreadN {
  run() {
    N1;
    N2;
    N3;
    wait(3s);
    N4;
    N5;
    wait(2s);
  }
}
```

Thread O — Prioridade: **4**

```
ThreadO {
  run() {
    O1;
    O2;
    wait(2s);
    O3;
    O4;
    wait(2s);
    O5;
    O6;
    wait(2s);
  }
}
```

Thread P — Prioridade: **2**

```
ThreadP {
  run() {
    P1;
    wait(1s);
    P2;
    P3;
    wait(2s);
  }
}
```

Figura 4.6 – Pseudocódigo de cinco threads (Thread L, Thread M, Thread N, Thread O e Thread P) sincronizadas por barreiras (comando wait()) e com prioridades.

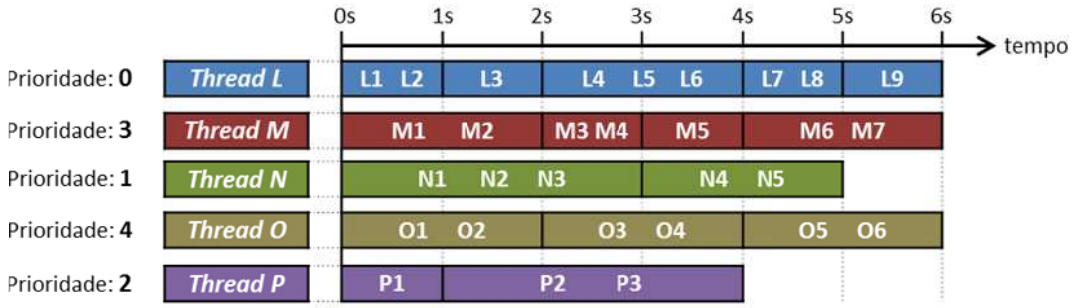


Figura 4.7 – Execução dos comandos das threads da Figura 4.6 (Thread L, Thread M, Thread N, Thread O e Thread P).

Figura 4.8 – Sequência de execução das cinco *threads* da Figura 4.6 controladas por sincronização de barreiras múltiplas e apresentando prioridades. Em barreiras temporais de mesmo valor, as *threads* com maior prioridade executam seus comandos antes das *threads* com menor prioridade.

4.4.5 – Sincronização por barreiras com atendimento a interrupções

Para que um objeto simulado possa responder à estímulos externos e internos, possibilitando a modelagem interativa do seu comportamento de forma semelhante aos objetos animados do mundo real, um mecanismos de atendimento à requisições (interrupções) deve estar disponível na estrutura de programação do simulador.

Quando uma *thread* está executando a sequência de comandos de um método, não existe um mecanismo de interrupção que permita desviar a execução sequencial desses comandos para executar outro conjunto de comandos (atendimento/tratamento de interrupções)

e posteriormente retornar à execução dos comandos da sequência principal, dando continuidade a partir de onde ela foi interrompida.

De maneira geral, o tratamento de interrupções ocorre no nível do sistema operacional, que controla a execução de processos e pode pausar a execução desses processos para atender a uma interrupção (receber dados de uma porta serial, por exemplo). Um mecanismo de interrupções semelhante é encontrado em microcontroladores, cuja execução dos comandos da função principal é interrompida na ocorrência de uma interrupção, deslocando o fluxo de execução para as instruções da função destinada ao tratamento da interrupção, retornando à execução das instruções da função principal ao término da execução da função de interrupção.

Apesar dos objetos (*threads*) em execução nos computadores não possuírem mecanismos que permitam o atendimento à interrupções, esta propriedade pode ser conseguida unindo a sincronização por múltiplas barreiras e uma lista de chamadas de interrupções (vetor de interrupções).

Quando uma *thread* receber uma chamada de interrupção, esta deverá ser registrada em um vetor de interrupções. Como o fluxo de execução dos comandos de uma *thread* sincronizada é controlado por comandos de barreira que a bloqueiam em tempos específicos, antes do desbloqueio da mesma, isto é, antes de completar o tempo de simulação necessário ao processamento dos comandos atualmente em execução e dar continuidade à execução dos comandos seguintes, o controlador de sincronismo deverá verificar se alguma interrupção foi chamada e, caso isso tenha ocorrido, inicia-se a execução do método que atenda àquela interrupção. A sequência de comandos do método de tratamento da interrupção também pode (e deve) ser sincronizada com comandos de barreira.

No exemplo da Figura 4.9, a *Thread Q* possui o método principal *run()* e o método *intA()* responsável pelo tratamento de uma interrupção identificada como *intA*.

Na ocorrência dessa interrupção no tempo de simulação 1,5 s, por exemplo, o fluxo de execução termina de executar os comandos *Q2* e *Q3* (utilizando o 1s necessário para a execução desses dois comandos) e quando atingem a barreira de sincronização após esses comandos (2s), há um deslocamento de fluxo para a execução do método de tratamento da interrupção (*intA()*), executando os comandos de *QA1* ao *QA4*. Ao final da execução do método *intA*, o método *run()* volta a ser executado dando sequência aos comandos a partir da barreira outrora alcançada na simulação, isto é, o controlador de simulação passa a executar os comandos *Q4*, *Q5* e *Q6*.

Após a execução do conjunto de comandos do método de interrupção, o controlador

de sincronismo verifica se outras interrupções foram chamadas e, caso isso ocorra, desloca o fluxo de execução para o método de tratamento das outras interrupções chamadas. Se não houver outras interrupções registradas, o controlador de sincronismo retornará à execução do conjunto de comandos do método principal outrora interrompida.

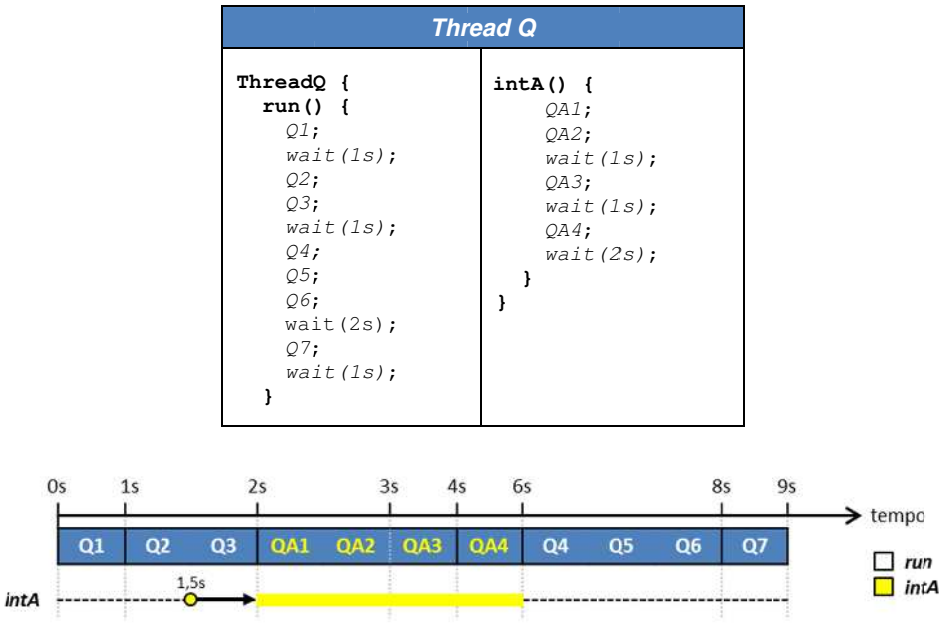


Figura 4.9 – Pseudocódigo e sequência de execução dos comandos da *Thread Q* com o método principal *run()* e um método de tratamento de interrupção *intA()*. A chamada à interrupção *intA* foi realizada por um agente externo no tempo de simulação de 1,5 s.

Pode haver diversas interrupções associadas a um mesmo tipo de *thread*, mas não são registradas múltiplas chamadas à uma mesma interrupção que está sendo atendida. Gerar uma interrupção significa sinalizar que o método associado à interrupção deve ser executado o mais rápido possível para atender à requisição. Quando a sinalização de uma interrupção já existe (apenas a sinalização ou o método de tratamento da interrupção está em execução), outras chamadas à mesma interrupção não geram sinalizações adicionais. Ao final da execução de uma interrupção, a sinalização de chamada da interrupção que foi tratada é excluída e novas chamadas poderão ser registradas.

Outra característica associada ao tratamento das interrupções é que o fluxo de execução de comandos só poderá ser desviado a partir de um conjunto de comandos de um método principal, isto é, os comandos de um método de tratamento de interrupção nunca são interrompidos pela chamada de outra interrupção. Portanto, os métodos de interrupção não se interlaçam, sendo preciso que um método de tratamento de interrupção finalize para que outro possa iniciar a sua execução.

A Figura 4.10 apresenta o exemplo da *Thread R* com o método de execução principal *run()* e dois métodos de interrupção (*intA()* e *intB()*) que podem ser chamados e interromper a execução do método principal. Supondo que foram realizadas três chamadas à *intB* nos tempos de simulação 2,7s, 7,2s e 9,6s e uma chamada à *intA* no tempo 1,6s, de acordo com as regras apresentadas de controle das chamadas à interrupções, a sequência de execução segue conforme ilustrado na mesma figura e é comentada a seguir.

Thread R

```
ThreadR {
  run() {
    R1;
    wait(1s);
    R2;
    wait(1s);
    R3;
    R4;
    wait(2s);
    R5;
    wait(1s);
    R6;
    wait(1s);
  }
```

```
intA() {
    RA1;
    RA2;
    wait(1s);
    RA3;
    wait(1s);
  }

intB() {
    RB1;
    wait(1s);
    RB2;
    wait(1s);
  }
}
```

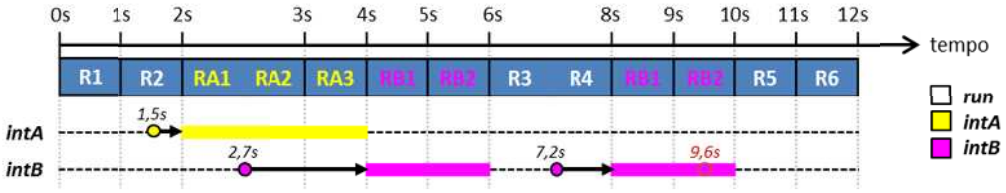


Figura 4.10 – Pseudocódigo e sequência de execução da *Thread R* com o método principal *run()* e os métodos de interrupção *intA* e *intB*. As interrupções foram geradas por agentes externos, sendo que a interrupção *intA* ocorreu no tempo de simulação de 1,5s e as chamadas à interrupção *intB* ocorreram nos tempos 2,7s, 7,2s e 9,6s.

Os comandos do método *intA* só começam a ser executados quando há a finalização do tempo de execução do comando *R2*, mesmo que a chamada à interrupção *intA* tenha ocorrido 0,4s antes. A interrupção *intB* foi chamada pela primeira vez em 2,7s, mas só pode ser atendida após a finalização total da execução dos comandos da interrupção *intA*, que estava sendo atendida no momento de ocorrência da interrupção *intB*, não havendo, portanto, entrelaçamento na execução de métodos de interrupção. A segunda e terceira chamadas à interrupção *intB* ocorrem em 7,2s e 9,6 s, respectivamente, mas como a terceira chamada ocorre durante a execução do método *intB* (referente à segunda chamada à interrupção *intB*), ela não é registrada e, como se pode observar na sequência de execução geral dos comandos, a sequência de comandos da interrupção *intB* só é executada duas vezes, apesar da chamada à interrupção *intB* ter sido realizada três vezes.

As interrupções podem ter uma origem externa ao objeto (gerada por outros objetos) ou ser requisitada a partir do próprio objeto. Um caso particular de interrupção gerada internamente ao objeto são os temporizadores (*timers*). Estes possuem um período de execução, no qual uma interrupção é chamada de forma automática. Enquanto o temporizador permanecer ativo, a interrupção é chamada indefinidamente a cada período.

A Figura 4.11 apresenta a definição da *Thread S* que possui o método principal *run()* e um método de temporização denominado *timerA*, cujo período é de 2s. Isso significa que a cada 2s uma interrupção é gerada automaticamente, requisitando que o objeto execute os comandos do método *timerA*.

Thread S

```
ThreadR {
  run() {
    S1;
    wait(1s);
    S2;
    wait(1s);
    S3;
    S4;
    S5;
    wait(3s);
    S6;
    S7;
    wait(1s);
    S8;
    wait(1s);
  }
```

```
timerA() {
    SA1;
    SA2;
    wait(1s);
  }
}
```

Interrupção	Prioridade	Período
timerA	1	2,0 s

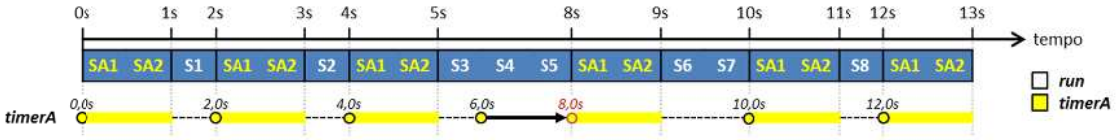


Figura 4.11 – Pseudocódigo e sequência de execução da *Thread S* com o método principal *run()* e o método de temporização *timerA* com período de repetição de 2,0s.

Ainda na Figura 4.11, a sequência de execução dos comandos mostrada permite identificar as interrupções geradas durante a execução dos comandos do método principal *run()*. Como o *timerA* está ativo desde o início da simulação, logo no instante 0s o método *timerA* é executado e a partir daí a cada 2s. No instante 6s, a interrupção do temporizador foi gerada durante a execução dos comandos *S3*, *S4* e *S5* do método principal *run()*, então a interrupção só é atendida ao se completar os 3s de tempo de simulação necessários à execução desses comandos. Somente no tempo de simulação de 8s é que a interrupção gerada no instante 6s é atendida. Contudo, neste mesmo instante (8s) ocorre outra interrupção

chamada pelo temporizador. Seguindo a mesma política de ocorrência de múltiplas interrupções geradas ao mesmo tempo ou durante a execução do método de tratamento da interrupção, a interrupção do instante 8s não será atendida.

Se o período estabelecido para repetição de um temporizador for menor que o tempo de simulação necessário para executar os comandos do método, a execução do método que trata a interrupção temporizada não será executada indefinidamente, como se poderia pensar, pois as interrupções geradas durante a execução do método serão anuladas.

A possibilidade de se associar diversas interrupções externas e *timers* à mesma *thread* pode gerar conflito no caso de duas ou mais chamadas à interrupções distintas ocorrerem no mesmo instante ou foram chamadas em momentos distintos mas ainda não tiverem sido atendidas. Nesses casos, um mecanismo de prioridade pode ser associado às interrupções, fazendo com que a escolha de qual interrupção será atendida primeiro seja baseada no critério da ordem decrescente das prioridades associadas às interrupções.

O exemplo da Figura 4.12 apresenta duas interrupções temporizadas (*timerA* e *timerB*) e uma interrupção externa (*intC*) pertencentes à *Thread T*. Cada interrupção possui uma prioridade específica, conforme indicado na figura.

No instante 0 s de execução, ocorre a interrupção do *timerA* e do *timerB*. Como o *timerA* possui uma prioridade maior que o *timerB*, os comandos daquele são executados primeiro e somente ao término dos comandos do método do *timerA* é que os comandos do *timerB* são executados. No instante 11 s há outra decisão a ser realizada, pois tanto a interrupção do *timerB* (levantada no instante 10 s) como a interrupção externa *intC* (levantada em 8,7 s) estão aguardando para serem atendidas. Ao término da execução do método *timerA* (11 s), o gerenciador da simulação permite que a *intC* seja atendida antes da interrupção temporizada *timerB*, pois àquela possui maior prioridade que esta.

Outro evento interessante que pode ser visualizado na Figura 4.12 diz respeito ao que acontece no instante 15s. A interrupção do temporizador *timerB* ocorre durante a execução do método *timerB*, cuja interrupção foi chamada no instante 10s, mas que naquele momento não apresentava condições de ser atendida devido às interrupções levantadas do *timerA* e *intC,* estas com maior prioridade. Como no caso de interrupções externas que ocorrem durante o atendimento do mesmo tipo de interrupção, no caso dos temporizadores também a segunda interrupção é descartada (não registrada).

Quanto mais refinada a execução, isto é, quanto mais comandos *wait()* forem utilizados entre os comandos dos método principal, mais chances das interrupções serem atendidas no momento em que de fato ocorrem.

Thread T

```
ThreadT {
  run() {
    T1;
    wait(1s);
    T2;
    T3;
    wait(2s);
    T4;
    T5;
    wait(1s);
    T6;
    wait(2s);
    T7;
    wait(1s);
  }
```

```
timerA() {
    TA1;
    wait(1s);
  }

timerB() {
    TB1;
    TB2;
    wait(2s);
  }

intC() {
   TC1;
   wait(2s);
  }
}
```

Interrupção	Prioridade	Período
timerA	3	3,0 s
timerB	1	5,0 s
intC	2	×

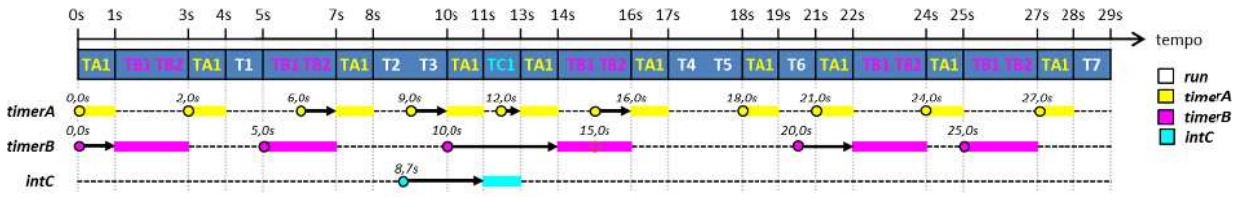


Figura 4.12 – Pseudocódigo e sequência de execução da *Thread T* com o método principal *run()*, os métodos de temporização *timerA* e *timerB* e a interrupção *intC*. A interrupção relacionada ao método intC foi realizada no instante 8,7s.

Propiciado principalmente pelo mecanismo de barreiras múltipla, o esquema de desvio de fluxo sequencial na execução de comandos de *threads* para atender requisições externas ou temporizadas faz com que o **requisito (4)** seja atendido, dando às *threads* a capacidade de atender ao chamado de interrupções.

4.4.6 – Fluxograma e exemplo de funcionamento do núcleo de simulação proposto

Como visto, o comportamento proposto para os objetos em simulação pode ser estabelecido pelo uso de *threads* em conjunto com adaptações do modelo de simulação por barreiras. Os itens sobrejacentes apresentaram de forma isolada os conceitos e as ideias de implementação dos quatro requisitos que dão aos objetos a capacidade de execução de comandos de forma sincronizada, além de permitir o relacionamento entre objetos no decorrer da simulação.

O fluxograma da Figura 4.13 apresenta a o andamento de uma simulação tendo

como base a estrutura de controle proposta. O Controle possui um relógio global (*t*) que registra o tempo de simulação e uma lista de *threads* (*LT*) que representa os objetos da simulação. Cada *thread* possui uma lista de interrupções/*timers* (*LI*) e um atributo de tempo de simulação (*next_t*) que indica quando a *thread* pode continuar a sua execução de comandos, isto é, quando termina o tempo de simulação associado aos comandos que precedem o bloqueio da *thread.* A atualização do valor de *next_t* e o comando *bloqueio()* são os comandos que compõem a instrução de barreira *wait()*, utilizadas nos exemplos até o momento.

No início da simulação o Controle zera o seu relógio global e em seguida realiza uma busca em *LT* da *thread* que esteja ativa (não tenha finalizado sua execução), possua o menor valor de *next_t* e a maior prioridade. Se nenhuma *thread* estiver ativa, a simulação termina, caso contrário, a *thread* que satisfaça a condição acima é selecionada e o valor de *next_t* é atribuído ao relógio global *t*. A *thread* selecionada é desbloqueada e o Controle é bloqueado até que receba um sinal de desbloqueio da *thread* que foi desbloqueada.

Todas as *threads* da simulação iniciam suas execuções ativando a si mesmas (*active* ← *true*), zerando seu relógio interno *next_t* e se autobloqueado (*bloqueio()*), permanecendo bloqueadas até que o Controlador as desbloqueie. Cada *thread* desbloqueada pelo Controlador encontra-se em um estado particular, de acordo com a sua última execução. Quando a *Thread* é desbloqueada, a primeira verificação realizada é se na sua execução anterior, isto é, antes do bloqueio, ela estava executando o método de alguma interrupção/*timer*. Isto é feito examinado o atributo *im*, que registra o método de interrupção/*timer* que estava sendo executado e ainda não foi finalizado. Caso *im* esteja registrando algum método de interrupção/*timer*, do próximo comando deste método é executado e é feita a atualização do relógio interno da *thread* (*next_t*). Se for o último comando do método, *im* recebe o valor de NULL, assim na próxima vez que a *thread* for desbloqueada estará registrado que nenhum método de interrupção/*timer* ainda está sendo executado. Segue o desbloqueio do Controle e bloqueio da *Thread* que estava em execução.

Se a *Thread et* for desbloqueada e o valor de *im* for NULL, é verificado se existe alguma interrupção/*timer* que foi chamada, e, em caso afirmativo, é selecionada aquela que possuir a maior prioridade e a execução do seu primeiro comando é realizada. Caso nenhuma interrupção/*timer* tenha sido chamado, o próximo comando do método principal é executado e é feita a atualização o valor de *time_t*, segundo o tempo de simulação necessário à execução dos comandos do método principal. Caso o comando executado seja o último comando do método principal, a *Thread* é desativada (*active* ← *false*), é feito o desbloqueio do Controle e a *Thread* chega ao fim de sua execução. Se o comando executado do método principal não for o último, a *Thread* desbloqueia o Controle e se bloqueia.

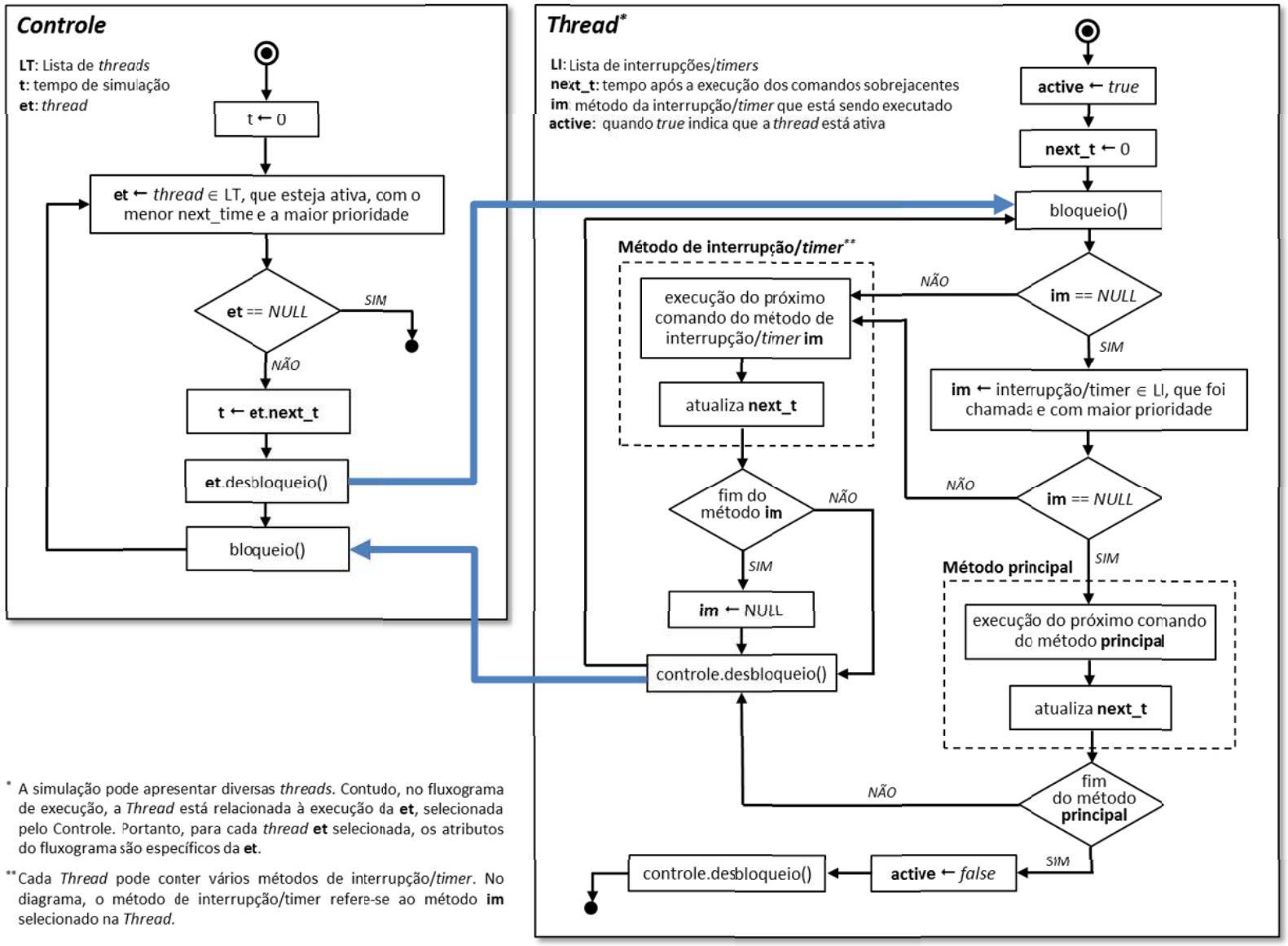


Figura 4.13 – Fluxograma de execução de uma simulação, cujos componentes são uma *thread* de Controle e *Threads* que representam os objetos simulados.

A Figura 4.14 apresenta a definição de quatro *threads* que serão tomadas como exemplo para a execução do algoritmo de controle de sincronismo apresentado na Figura 4.13. A *Thread U* possui prioridade 0 (menor prioridade) e um método de interrupção denominado *intU1*. A *Thread V* possui prioridade 1, um método de interrupção denominado intV1 (prioridade 1) e um método de interrupção temporizado denominado *timerV2* (prioridade 2 e período de chamada de 5,0s). A *Thread W* apresenta prioridade 2 e dois métodos de interrupção por temporização, *timerW1* (prioridade 1 e período 4,0s) e *timerW2* (prioridade 2 e período 8,0s). A última *thread* definida na figura é a *Thread X*, com prioridade 3 (maior prioridade).

Portanto, em uma mesma barreira temporal e seguindo a ordem de prioridade, os comandos da *Thread X* serão executados antes dos comandos da *Thread W*, que por sua vez serão executados antes dos comandos da *Thread V* e finalmente serão executados os comandos da *Thread U*. Cada *thread* apresenta um conjunto de comandos sincronizados pelo comando de barreira *wait()*. Além disso, o comando *call()* é utilizado para realizar a chamada de interrupção de uma *thread.*

Considerando as prioridades internas das interrupções de cada *thread*, se a interrupção *intV1* e a interrupção temporizada *timerV2* estiverem sinalizadas na *Thread V*, a interrupção *intV1* será executada primeiro por apresentar maior prioridade. Da mesma forma na *Thread W*, se ambos temporizadores *timerW1* e *timerW2* estiverem sinalizados no mesmo momento, o temporizador *timerW2* será executado primeiro, por apresentar maior prioridade em relação ao *timerW1*.

A Figura 4.15 apresenta a sequência de execução dos comandos de cada *thread* definida na Figura 4.14, indicando os momentos de ocorrência das chamadas de interrupção temporizadas (*timerW1* a cada 4,0s e *timerW2* a cada 8,0s na *Thread W* e *timerV2* a cada 5,0s na *Thread V*). As interrupções também estão indicadas de acordo com o momento em que são chamadas a partir de outras *threads.*

O atendimento às interrupções não ocorre imediatamente após a chamada da interrupção, conforme previsto pelo modelo apresentado. Para que uma *thread* atenda a uma interrupção é preciso que ela termine de executar as tarefas que estão sendo executadas naquele momento, finalizando o tempo necessário para essa execução.

Portanto, atrasos no atendimento à interrupções são frequentes. A interrupção *intV1,* por exemplo, pertencente à *Thread V*, foi chamada pela *Thread U* no tempo 6s, mas só pode ser atendida no tempo 8s, uma vez que a *Thread V* possui uma prioridade maior que a *Thread U* e no instante 6s a *Thread V* já havia iniciado a execução dos comandos *V6* e *V7*, que necessitam de 2s para serem executados.

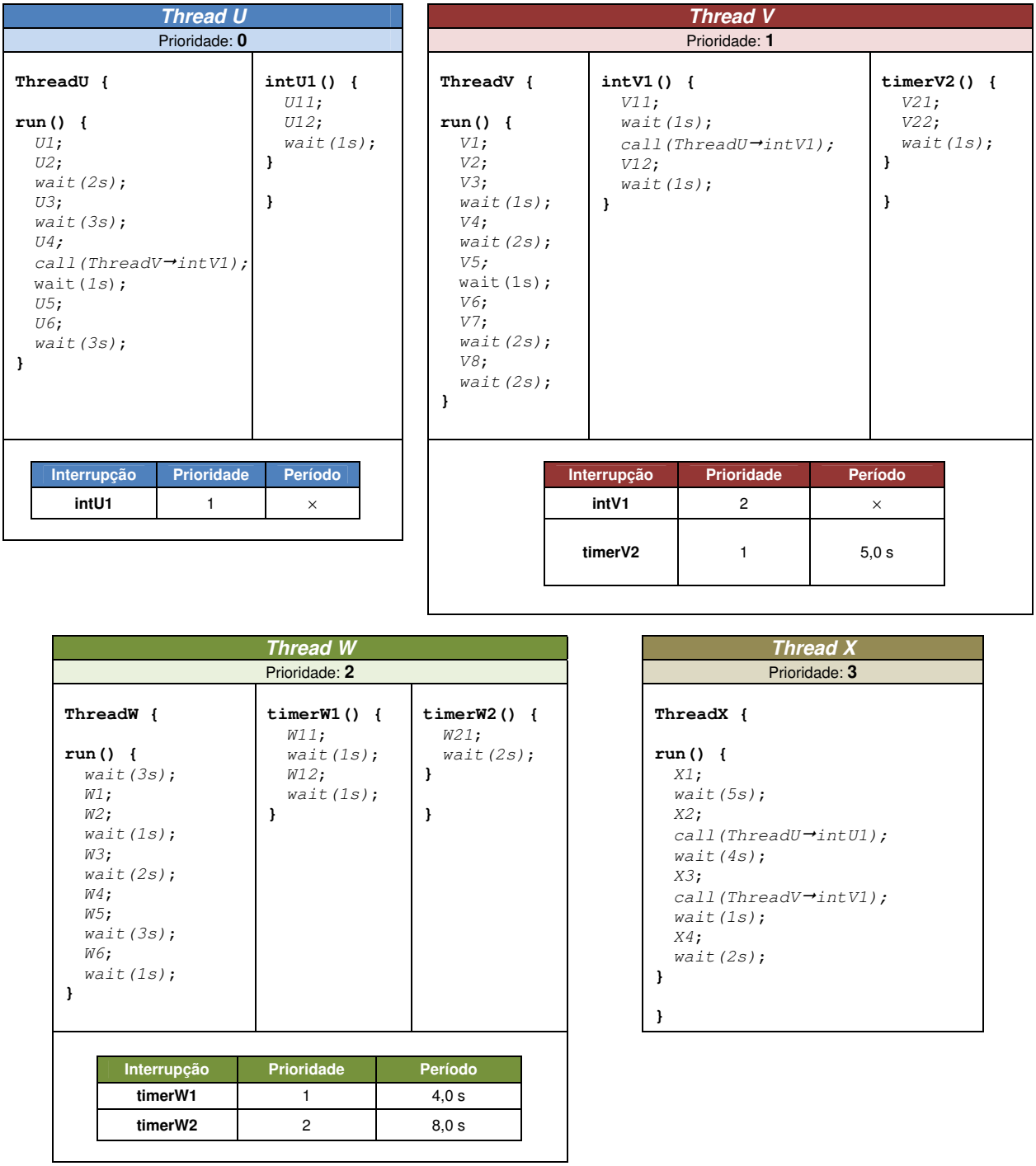


Figura 4.14 – Definição de quatro *threads* (*Thread U*, *Thread V*, *Thread W* e *Thread X*) apresentando prioridades e métodos de interrupção externas e interrupções temporizadas. A sequência de execução das *threads* é apresentada na Figura 4.15. O método *call*() é utilizado para realizara a chamada de uma interrupção.

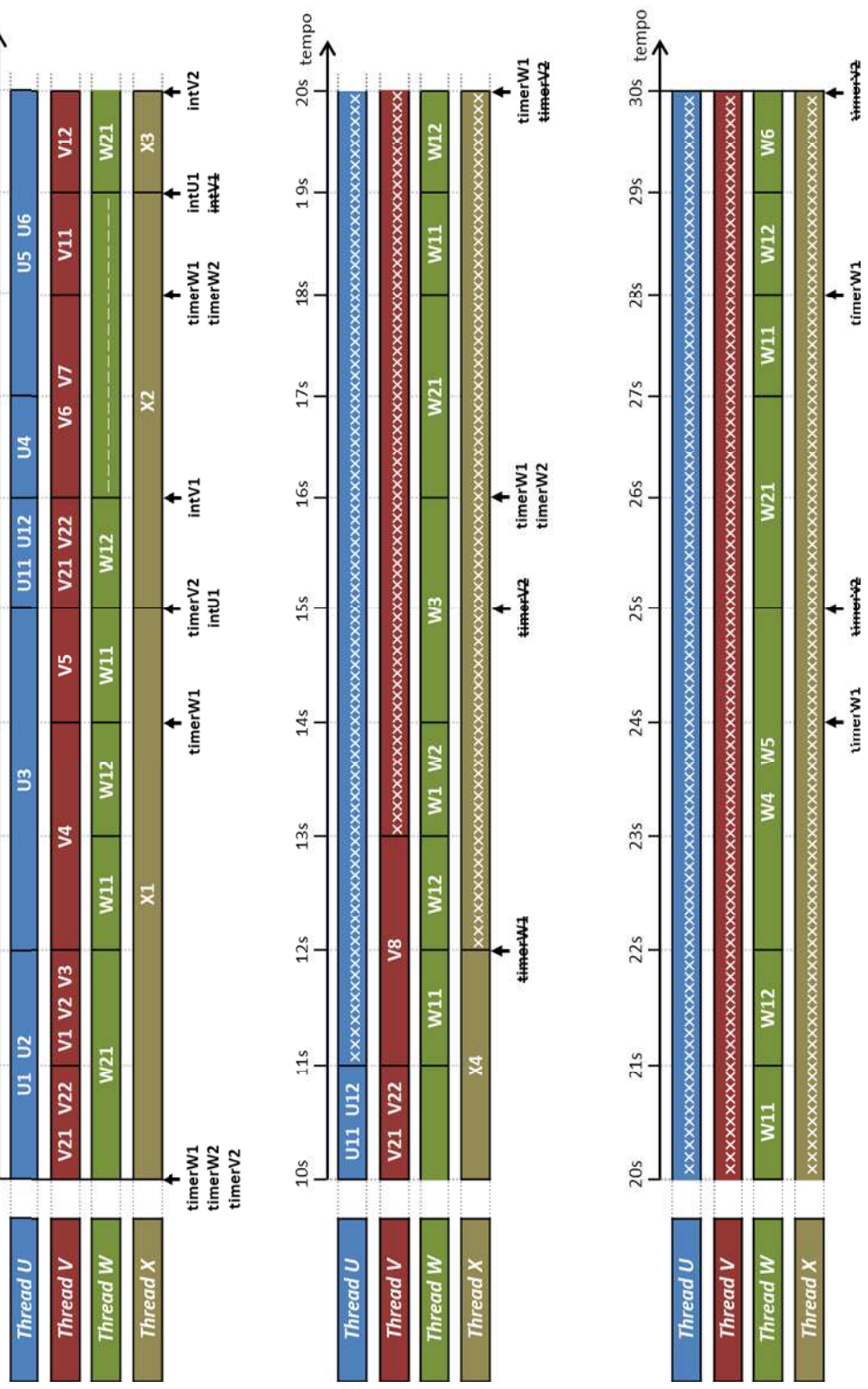


Figura 4.15 – Sequência de execução de comandos das quatro *threads* definidas na Figura 4.14. As chamadas de interrupção riscadas representam interrupções que não são atendidas por já estarem em execução no momento em que outra interrupção de mesma natureza ocorreu.

Outro ponto importante, também definido no modelo, diz respeito à ocorrência de interrupções enquanto o método da mesma está sendo executado ou já foi chamado anteriormente e ainda não foi atendido. Nesses casos, as interrupções subsequentes não são atendidas, identificadas com um risco na Figura 4.15. A chamada ao *timerW1* no instante 12s é enquadrada nessa situação, pois os comandos do método *timerW1* estavam em execução desde o instante 11s, permanecendo em execução até o instante 13s.

As interrupções também não são mais atendidas quando o método principal (*run()*) é finalizado, isto é, não existem mais comandos a serem executados no método principal. Na Figura 4.15 essas interrupções também aparecem riscadas, como é o caso do *timerV2* no instante 15s, uma vez que a *Thread V* finalizou no instante 13s.

A Figura 4.16 apresenta a sequência de processamento final dos comandos das *threads* da Figura 4.14, apresentada de forma paralela na Figura 4.15.

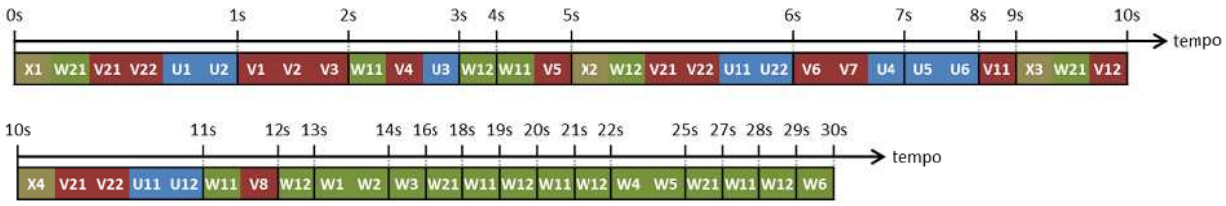


Figura 4.16 – Execução sequencial dos comandos das *threads* da Figura 4.14, apresentados de forma paralela na Figura 4.15.

4.5 – Frameworks existentes para o desenvolvimento de simuladores

Considerando o núcleo de simulação como um conjunto de regras bem definidas que controla a evolução temporal da simulação e que, ao mesmo tempo, pode ser programado para coordenar a interação entre os elementos simulados ao longo do tempo, ele pode ser apresentado como um *framework*, servindo de base para o desenvolvimento de simuladores de qualquer natureza. Quanto mais flexível e abrangente for o *framework*, maiores as possibilidades de um determinado tipo de simulador poder ser elaborado a partir do núcleo de simulação implementado no pacote de programação.

Existem diversos *frameworks* voltados para o desenvolvimento de simuladores. Cada qual com seu formato específico para descrever os objetos simulados e representar o tempo de simulação, além de apresentar limitações intrínsecas à implementação e linguagem de programação adotada. Contudo, grande parte das pesquisas que envolvem simulação de sistemas desenvolvem bibliotecas simples e não as descrevem ou fornecem o código fonte da programação [GOE05].

A Tabela 4.1 apresenta algumas das principais ferramentas utilizadas no desenvolvimento de simuladores. Foram selecionadas as sete ferramentas mais relevantes (mais utilizadas, citadas em artigos ou cuja arquitetura é semelhante ao núcleo de simulação proposto), em sua maioria *frameworks* ou bibliotecas de funções.

O SimJAVA [KRE97] é um framework que implementa um núcleo de simulação discreto do tipo *event-driven*. Desenvolvido por um grupo de pesquisadores da Nova Zelândia, o SimJAVA apresenta uma estrutura baseada em camadas (*layers*), que são definidas como classes que reagem ao recebimento de mensagens a partir de um protocolo específico da aplicação. A troca de mensagens é realizada por portas (*Sim_port*), sendo que cada mensagem dispara um evento, este associado a uma rotina (método) relacionado ao evento. O tempo de simulação para o processamento dos eventos é determinado dentro da rotina de tratamento do evento através do método *hold()*. O processamento dos eventos ocorre de forma serial, sendo que a fila de eventos é denominada "agenda" do sistema.

A motivação para o desenvolvimento do *framework* SimJAVA foi o modelamento de pilhas de rede. Contudo, a conexão entre as entidades (*Sim_entity*) através de portas remete a forma com que os circuitos integrados digitais são conectados entre si, sendo, portanto, um modelo de simulação coerente para esse tipo de aplicação.

Tabela 4.1 – Ferramentas para o desenvolvimento de simuladores

Ferramenta	Linguagem	Repositório na Internet
SimJAVA [KRE97]	Java	http://www.icsa.inf.ed.ac.uk/research/groups/hase/simjava/
J-Sim/JavaSim [TYA02]	Java	http://j-sim.cs.uiuc.edu/
JSDESLib [GOE05]	Java	[não disponível]
MASON [LUK05]	Java	http://cs.gmu.edu/~eclab/projects/mason/
SimKit [GOM95]	C++/Java	http://warp.cpsc.ucalgary.ca/Software/SimKit/simkit.php
Sim++ [LOM90]	C++	http://www.simplusplus.com/
SystemC [GEO02]	C++	http://www.systemc.org

J-Sim é um ambiente de simulação desenvolvido na linguagem de programação Java. Formalmente conhecido como JavaSim [TYA02], esta ferramenta foi desenvolvida com o propósito de modelagem e simulação de redes.

A arquitetura do J-Sim é do tipo *event-driven*, sendo que *threads* são criadas de acordo com a necessidade dos eventos serem processados em paralelo. Apesar de o tempo ser considerado contínuo na definição do modelo do J-Sim, existe uma escala temporal (*time_scale*) que pode ser configurada. Esta permite que o tempo seja discretizado ou contínuo, neste caso com o valor de *time_scale* igual a infinito.

A biblioteca JSDESLib [GOE05] também implementa um núcleo de simulação do tipo *event-driven*. Os eventos são colocados em uma fila e o método *body()* das entidades são os responsáveis pelo tratamento dos eventos, devendo, portanto, ser composto de uma estrutura de seleção (*switch* por exemplo). Cada evento possui um identificador do seu executor, um identificador do evento, o instante de tempo em que o evento deverá ser processado, as entidades-destino relacionadas ao evento e os dados referentes ao evento. A lista de eventos permanece ordenada por tempo de execução.

MASON [LUK05] é um núcleo de simulação baseado em eventos e agentes (pode haver uma interação dos elementos simulados com um ambiente no qual estão inseridos), sendo dividido em três camadas: utilidade, modelamento e visualização. A camada de utilidade é genérica e pode ser usada para qualquer finalidade; a camada de modelamento é composta do escalonador de eventos; a camada de visualização é composta de elementos de visualização dos objetos simulados e manipulação dos mesmos.

Para se utilizar a camada de visualização no MASON é necessário que a simulação seja executada até o fim sem a utilização de interface gráfica; somente então é possível executar a camada de visualização sobre o registro da simulação processada.

O Sim++ é definido em [LOM90] como uma linguagem de simulação paralela orientada a objeto. Trata-se de uma ferramenta comercial para criação de simuladores desenvolvida na linguagem de programação C++, por isso não pode ser considerada uma linguagem propriamente dita.

As simulações executadas no Sim++ são determinísticas e baseiam-se no conceito de entidade (*sim_entity*) e eventos (*sim_event*), podendo caracterizar esse *framework* de simulação como do tipo *event-driven*.

O SimKit [GOM95] é uma biblioteca de classes que permite o desenvolvimento de simuladores tanto com processamento sequencial como em paralelo, sem a necessidade de alterações na programação. Trata-se de um núcleo de simulação conservativo e otimista, com

um algoritmo de *rollback* baseado no mecanismo Time Warp.

Desenvolvido em apenas três classes (*sk_simulation*, *sk_lp* e *sk_event*), o SimKit pode ser utilizado tanto em simuladores desenvolvidos na linguagem de programação C++ como na linguagem Java. Apesar de ser um núcleo de simulação de propósito geral, o SimKit possui forte tendência a ser utilizado para modelamento de redes de comunicação, sistemas computacionais e de transporte, devido às bibliotecas específicas desenvolvidas para essas aplicações.

SystemC [GEO02] é uma biblioteca desenvolvida na linguagem de programação C++ com o objetivo de auxiliar o modelamento, no nível de sistema, de elementos de software e hardware, ou ainda uma combinação dos dois. O tempo de simulação é representado como um inteiro de 64 bits, impossibilitando a simulação de tempo contínuo, sendo necessária a determinação de um quantum de tempo (resolução temporal), caracterizando o núcleo de simulação como sendo do tipo *time-driven*. Qualquer valor de tempo menor que a resolução temporal é considerado como tempo nulo, isto é, arredondado para zero.

A organização dos objetos em SystemC é realizada através do conceito de módulo, nos quais são declaradas funções que são executadas como processos concorrentes. Quando a simulação inicia no tempo zero, os processos dos módulos são iniciados e executados até o seu final, sem nenhum tipo de interrupção. A comunicação entre os módulos é realizada através de uma fila que realiza a entrega das mensagens de acordo com a política FIFO (*Firt-In-First-Out*).

Devido a sua natureza de processos concorrentes e pela própria estrutura organizacional das funções, SystemC pode ser comparada às linguagens descritoras de hardware (VHDL e Verilog).

A descrição apresentada das principais características dos *frameworks* disponíveis na literatura não tem como objetivo listar seus componentes e explicar o funcionamento interno dos núcleos de simulação de maneira detalhada, mas expor as diferentes possibilidades de interpretação do problema e implementação de soluções.

O fato de existirem diversas ferramentas desenvolvidas para compor o núcleo de simuladores revela, portanto, as diferentes visões de cada grupo de pesquisa sobre a forma com que os objetos são modelados e o tempo de simulação é conduzido em ambiente computacional. Não é adequado considerar um *framework* ou uma biblioteca de funções como melhor que a outra, mas sim como mais adequada à determinada aplicação e também como uma ferramenta cujo programador possui maior familiaridade ou facilidade de uso, uma vez que o processo de aprendizado que leva ao domínio da ferramenta computacional de simulação é

uma tarefa por vezes penosa diante a documentação escassa e complexidade do modelo adotado. Essas características fazem com que uma determinada ferramenta seja eleita em determinado projeto ou abre caminho para o desenvolvimento de uma nova arquitetura de programação que difere das existentes ou as complementa.

As ferramentas para desenvolvimento de simuladores analisadas não contemplam os requisitos estabelecidos no modelo de comportamento dos objetos na simulação (item 4.4.1). Algumas implementações, como visto, utilizam uma lista de eventos a serem processados, havendo, portanto, um custo computacional em criar mensagens contendo a descrição dos eventos e manter uma estrutura que as coordene. Uma das diferenças do *framework* que será proposto é que abole essa abordagem, buscando, com isso, melhorar o desempenho dos simuladores desenvolvidos.

Contudo, a grande diferença do núcleo de simulação que é proposto neste trabalho é que ele permite a preempção, isto é, a interrupção na execução de uma determinada tarefa para atender a uma requisição e posterior retorno à tarefa interrompida. Além disso, o *framework* desenvolvido permite um controle de precedência causal, problema este que não é diagnosticado na descrição das ferramentas de desenvolvimento de simuladores estudadas.

Algumas das ferramentas de simulação destinadas ao desenvolvimento de simuladores do tipo *event-driven* apresentadas neste item são descritas e comparadas com mais detalhes em [LOW99].

4.6 – O *framework* desenvolvido

O BaSS (*Barrier Sinchronization Simulator*) é um *framework* desenvolvido na linguagem de programação Java que implementa um núcleo de simulação discreto do tipo *event-driven* baseado em agentes. O mecanismo de funcionamento do BaSS está em acordo com o núcleo de simulação proposto no item 4.4.

A opção pela linguagem de programação Java pode ser justificada principalmente pelo fato de ser uma linguagem verdadeiramente orientada a objetos, o que permite um modelamento dos objetos do mundo real no ambiente computacional da forma mais natural possível. Outras linguagens de programação orientadas a objeto, como C++, C# e Objective-C, também foram cogitadas por possuírem essa facilidade de modelamento, mas Java é a única que de fato foi concebida dentro do paradigma orientado a objetos, sendo as demais uma extensão da linguagem de programação C. A estruturação da linguagem Java possui, portanto, um direcionamento que facilita e organiza a programação das classes e a arquitetura interna da JVM (Máquina Virtual Java) garante um melhor aproveitamento do processamento disponível

para a execução dos objetos.

Dentro da perspectiva do simulador proposto, é fundamental que a linguagem de programação escolhida possua *threads*. A linguagem Java, além de possuir uma arquitetura de *threads* robusta e bem documentada, permite ao programador um razoável poder de controle sobre a sua execução, facilitando a programação de eventos que ocorrem simultaneamente no mundo real.

Além dos fatores específicos citados, optou-se pela linguagem Java devido a sua portabilidade (a independência de plataforma, permitindo a interpretação das classes compiladas em diferentes arquiteturas computacionais), à gratuidade do compilador e interpretador, à documentação facilitada e organizada (utilização da ferramenta Javadoc), ao reuso de classes e a facilidade de incorporação de novas classes ao projeto.

Com relação ao desempenho da linguagem Java em relação a outras linguagens de programação comumente utilizadas em aplicações científicas, o estudo apresentado em [BUL01] demonstra, através de testes de performance, que o desempenho da linguagem Java está equiparado à linguagem C e Fortran. Em [LUK05] os autores expressam que apesar da linguagem de programação Java possuir a reputação de ser lenta, uma vez que bem programada pode apesentar resultados surpreendentemente rápidos. Particularmente para simuladores de Redes de Sensores Sem Fio, [EGE05] salienta que alguns simuladores escritos em Java possuem um desempenho melhor que os escritos na linguagem C.

4.6.1 – Restrições e estratégias para a utilização da linguagem de programação Java na implementação do modelo de sincronização por barreiras proposto

Apesar da maior adequação da linguagem de programação Java para a implementação do modelo de sincronização por barreiras frente às outras linguagens de programação, ela também não possui todos os mecanismos necessários ao funcionamento pleno do núcleo de simulação proposto. Em alguns casos faz-se necessário a escolha de formas representativas (tipo de dados, objetos, etc.) mais adequadas para determinados elementos da simulação; em outros casos, a utilização de soluções pré-existentes deve ser evitada em virtude da sua incompletude ou limitação comportamental.

A representação do tempo de simulação é o primeiro parâmetro que deve ser cuidadosamente estabelecido no ambiente computacional. Trata-se de um parâmetro crítico no simulador, visto que toda a sequência de comandos é determinada pela comparação do tempo atual da simulação com o tempo do próximo evento de cada *thread* (vide os atributos *t* e *next_t* da Figura 4.13). O tipo de dado utilizado na representação do tempo, portanto, deve ser o mais

exato possível, uma vez que qualquer problema de representação numérica pode gerar inconsistências na simulação.

Como o tempo pode ser dividido em frações, o tipo de dado ponto flutuante (*float* ou *double*) poderia ser utilizado de forma satisfatória, apenas com a restrição da faixa de valores que poderia ser representada ($1,40129846432481707\times10^{-45}$ até $3,40282346638528860\times10^{38}$ para valores positivos ou negativos no tipo de dado *float* e de $4,94065645841246544\times10^{-324}$ a $1,79769313486231570\times10^{308}$ para valores positivos ou negativos no tipo de dado *double*). Contudo, a representação numérica do tipo ponto flutuante não é exata (vide Apêndice A), devendo ser evitada em virtude das diversas comparações realizadas no decorrer da simulação. Um pequeno desvio na representação dos números que representam os tempos na simulação pode gerar um resultado errado na comparação desses tempos e a decisão resultante dessa comparação poderá acarretar problemas na sequência de execução de comandos.

A representação de números inteiros (*int*) na linguagem Java, entretanto, é exata. Apesar disso, a sua utilização seria limitada, pois intervalos menores que a unidade (frações de tempo) não poderiam ser utilizados na simulação. Tal restrição limitaria a utilização do núcleo de simulação, descaracterizando a sua concepção de ser de propósito geral.

A classe *BigDecimal*, pertencente ao pacote *java.math*, seria a forma representativa numérica que mais se adequa às necessidades de simulação, uma vez que permite o armazenamento e processamento de números fracionários sem haver perdas representativas consideráveis. A exatidão dos números representados em objetos da classe *BigDecimal* é determinada de acordo com o padrão IEEE 754 [IEE08], sendo o tamanho do dado variável de acordo com a necessidade da aplicação desenvolvida: 32, 64 ou 128 bits (vide Apêndice A).

Com relação à necessidade de se ter diversos objetos sendo executados de forma pseudo-paralela, isso pode ser conseguido diretamente com a utilização da classe *Thread* da linguagem Java. Contudo, como visto no item 4.4.2, a instrução de barreiras que permitiria a sincronização na execução de comandos não é nativa das *threads* em Java.

Threads sincronizadas por barreiras de acordo com o esquema apresentado na Figura 4.2 já foram implementadas na linguagem Java na classe *CyclicBarrier* do pacote *Java.util.concurrent* [JAV12], sendo o método *await()* utilizado na definição das barreiras (correspondente à instrução *barrier* da Figura 4.1). Contudo, a classe *CyclicBarrier* implementa o modelo de sincronização por barreiras clássico, não oferecendo uma forma de controle da ordem de execução de comandos pertencentes a *threads* distintas (prioridades), não permitindo múltiplas barreiras independentes (todas a *threads* necessariamente bloqueiam em todas as barreiras) e tampouco existe um mecanismo que permita o recebimento e tratamento de

interrupções.

Uma adaptação da classe *CyclicBarrier* poderia ser feita de tal forma a permitir o controle síncrono com barreiras múltiplas temporizadas, mas devido ao fato da implementação da classe *CyclicBarrier* não ser de domínio público, uma alteração na sua estrutura interna torna-se infactível.

A implementação do comando de barreira que permite que todas as funcionalidades apresentadas na proposição sejam programadas, conforme o fluxograma apresentado na Figura 4.13, está baseada em comandos de bloqueio e desbloqueio de *threads*. Uma instrução de barreira implementada em uma *thread* suspenderia (bloquearia) a execução dos comandos de um método até que um "aviso de prosseguir" fosse recebido, isto é, um comando de desbloqueio fosse chamado.

A implementação de *threads* síncronas que representam objetos do mundo real e do próprio controle da simulação é feita baseando-se em classes que herdam as propriedades e métodos da classe *Thread* do pacote *java.lang*, adicionando métodos específicos e seguros capazes de realizar o bloqueio e o desbloqueio na execução dos comandos.

Uma forma simples de se efetuar o bloqueio de uma *thread* seria executar o comando `while ( blocked == true ) { /*não faz nada*/ }`, sendo o atributo *blocked* do tipo booleano. A *thread* ficaria bloqueada até que o valor de *blocked* fosse alterado para *false* por outra *thread* que se encontra em execução simultânea. Contudo, essa estratégia não funciona, uma vez que a JVM executa o comando *while* de forma atômica, fazendo a *thread* ficar executando o comando de forma "infinita", não dando oportunidade de execução para outras *threads*, que poderiam não só executar seus comandos, mas também desbloquear a *thread* pela alteração no valor do atributo *blocked*. A *thread* e todo o programa, portanto, ficariam bloqueados na estrutura de repetição *while*.

Uma segunda possibilidade de bloquear e desbloquear uma *thread* seria utilizar os métodos *suspend()* e *resume()*, respectivamente, disponíveis na classe *Thread*. A Figura 4.17 mostra o ciclo de vida de uma *thread* com os métodos utilizados para se realizar a mudança de estados. Os métodos *suspend*() e *resume*(), apesar de terem sido desenvolvidos com o propósito de suspender e retomar a execução dos comandos de um objeto da classe *Thread*, foram considerados obsoletos (*deprecated*) por serem propensos a entrar em estado de *deadlock* (as *threads* ficam inter-bloqueadas). De fato, foram realizados testes de bloqueio e desbloqueio de *threads* utilizando os métodos *suspend()* e *resume()*, resultando em bloqueios que não puderam ser desfeitos via programação, além de desbloqueios que não tinham sido programados.

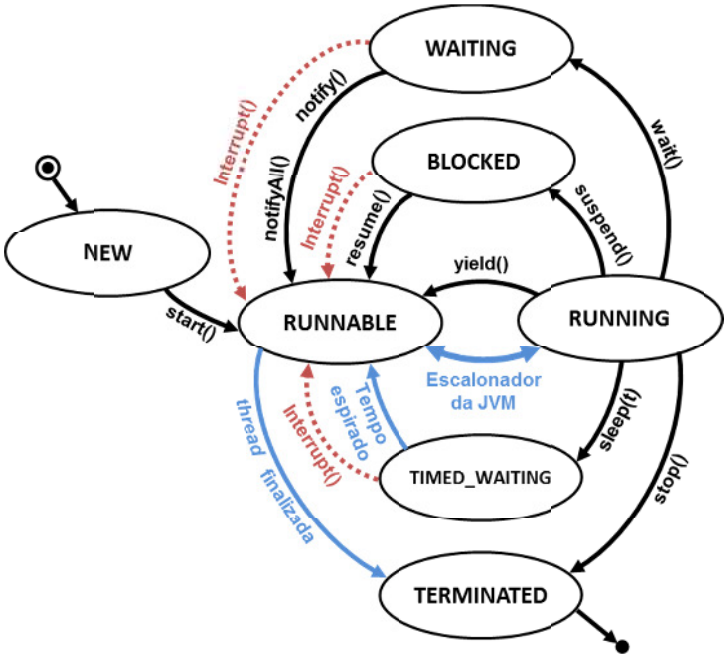


Figura 4.17 – Ciclo de vida de uma *thread*. Uma *thread* é criada no estado NEW e ao ser invocado o método *start*(), ela entra no estado RUNNABLE, isto é, fica disponível para ser executada. O escalonador de *threads* da JVM decide quando e por quanto tempo os comandos do método *run*() da *thread* são executados, entrando no estado RUNNING durante a execução. Uma vez em execução, a *thread* pode ser forçada a terminar pelo método *stop*(); pode "dormir" por um determinado tempo t pelo método *sleep*(t), entrando do estado TIMED_WAITING; pode ser suspensa (estado BLOCKED) pelo método *suspend*(), voltando a executar pelo método *resume*(); pode ficar aguardando uma notificação pelo método *wait*(), entrando no estado WAITING e saindo dele de volta para o RUNNING quando receber uma chamada do método *notify*() ou *notifyAll*(). O método *yield*() pode ser utilizado para sinalizar ao escalonador da JVM que a *thread* pode deixar de executar (sair do estado RUNNING), permitindo que outra *thread* seja executada. Estando a *thread* no estado TIMED_WAITING, WAITING ou BLOCKED, a utilização do método *interrupt*() faz com que uma exceção seja levantada na *thread* e ela retorne a estar disponível para execução (estado RUNNABLE).

Contudo, o mecanismo de interrupção existente nas *threads* de Java quando utilizado em conjunto com um mecanismo de exclusão mútua implementado por um semáforo binário mostrou-se uma forma adequada e segura de suspender a execução de uma *thread*. O método *wait*() e um dos métodos *interrupt*(), *notify*() ou *notifyAll*() são utilizados nessa abordagem.

A Figura 4.18 apresenta métodos seguros de bloqueio (*block()*) e desbloqueio (*unblock()*) de um objeto que implementa a interface *Runnable*, cujo método *run()* é executado de forma pseudo-paralela com outras *threads*. Ambos os métodos apresentam seus comandos sincronizados com o objeto *thread* (`synchronized( this.thread )`), pertencente à classe *Thread* do pacote *java.lang*, que representa a própria *thread* em execução. Esta sincronização de blocos de comandos referenciada ao objeto *thread* previne que a sequência de comandos utilizados para bloquear e desbloquear a *thread* sejam executadas simultaneamente.

```
01  void block() {
02     synchronized( this.thread ) {
03        this.block = true;
04        while ( this.block ) {
05           try {
06              this.thread.wait();
07           } catch ( Exception e ) { }
08        }
09     }
10  }
11
12  void unblock() {
13     synchronized ( this.thread ) {
14        this.block = false;
15        this.thread.interrupt(); //or notify()
16     }
17  }
```

Figura 4.18 – Métodos utilizados para bloquear (*block()*) e desbloquear (*unblock()*) uma *thread* de forma segura. A classe a qual esses métodos pertencem deve implementar a interface *Runnable* e objeto thread, pertence à classe *Thread*, é a própria *thread* que executa em pseudo-paralelismo e é sincronizada no bloqueio e desbloqueio.

O método de bloqueio atribui o valor *true* para o atributo booleano *block*, o semáforo binário da *thread*. A *thread* entra em estado de WAITING, isto é, bloqueia, utilizando o método *wait()*, nativo da classe *Thread*. Enquanto a *thread* permanecer no estado de WAITING, o *pool* de *threads* passa a execução de comandos para outras *threads*, permitindo que esta *thread* seja desbloqueada. Pode ocorrer que um sinal de *interrupt*(), *notify*() ou *notifyAll*() seja recebido pela *thread* bloqueada de forma não prevista (não programada), fazendo com que uma exceção seja levantada e a *thread* saia do estado de WAITING. Por isso o laço `while( this.block )` foi introduzido no método de bloqueio, tendo como controle o atributo *block* utilizado como semáforo. Ele garante que a *thread* só será desbloqueada se o atributo *block* estiver com valor *false*, sendo exatamente isso que ocorre no método de desbloqueio.

O método de desbloqueio (*unblock()*) atribui o valor *false* ao atributo *block*, e em seguida gera um sinal de interrupção (`interrupt()`) ou de notificação (`notify()`), fazendo com que uma exceção (*InterruptException*) seja levantada e o bloqueio outrora realizado pelo método *wait()* seja desfeito.

Tal mecanismo é imune ao *deadlock* devido à utilização da sincronização dos blocos de comandos de ambos os métodos com o objeto *thread* e também é imune a possíveis desbloqueios gerados por eventos internos à JVM que poderiam provocar exceções e desbloquear a *thread* de forma imprevista.

O bloqueio e o desbloqueio das *threads* pode ser realizado de forma segura pelo mecanismo apresentado, sendo o controle do bloqueio e do desbloqueio das *threads* realizado por *thread* de Controle, de acordo com a progressão temporal da simulação (vide fluxograma da Figura 4.13). O Apêndice B apresenta a definição de duas classes que exemplificam a

implementação de uma *thread* de controle e de *threads* controladas, tendo como base os comandos de bloqueio e desbloqueio.

Com relação às prioridades acrescentadas ao modelo de sincronização por barreiras e necessárias para o controle da precedência causal, a linguagem Java possui um esquema de prioridades já definido para as *threads*, no qual se pode atribuir um número inteiro de 1 a 10 para as *threads* instanciadas, sendo que as que possuem maior prioridade (maior número) são executadas antes das demais, conforme ilustrado na Figura 4.19.

Contudo, o esquema de prioridades nativo das *threads* de Java não pode ser utilizado para resolver o problema de dependência causal entre os comandos executados em *threads* distintas. As *threads* A e B representadas na Figura 4.19, por exemplo, por possuírem a mesma prioridade (10) teriam seus comandos executados de forma intercalada, escalonados pela JVM. A *thread* C só seria executada quando todos os comandos das *threads* A e B terminassem de executar, a não ser que houvesse mudança na prioridade das *threads* durante a execução.

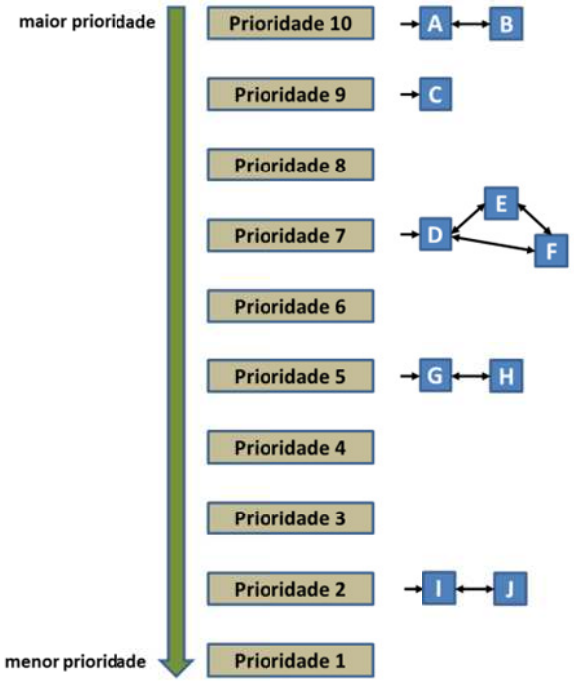


Figura 4.19 – Esquema de escalonamento de *threads* em Java. As *threads* de maior prioridade executam primeiro; somente após o término da execução das *threads* de prioridade n é que as *threads* de prioridade n-1 iniciam a sua execução. Adaptação de [DEI99].

No caso da existência de laços infinitos em *threads* de prioridades maiores, as *threads* com prioridades menores nunca teriam o seu código executado. Contudo, a documentação da linguagem Java [JAV12] especifica que teoricamente, em um dado instante, a *thread* de maior prioridade presente no *pool* de *threads* é a que deveria estar sendo executada,

mas o escalonador de *threads* pode optar por executar *threads* de menor prioridade para evitar esse efeito de *starvation* (morte por inanição).

Além de não resolver o problema de precedência causal conforme especificado no item 4.4.4, o esquema de prioridades nativo das *threads* em Java não garante uma sequencia de execução de *threads* de forma determinística e passível de ser utilizada para fins de sincronismo.

Métodos síncronos (*synchronized*) também não podem ser utilizados para resolver o problema de precedência causal entre comandos pertencentes à *threads* distintas, uma vez que se limitam a restrição de acesso à métodos dentro de um único objeto e não entre objetos pertencentes à mesma classe ou classes distintas.

A solução para o problema de dependência causal entre comandos de *threads* distintas não pode ser resolvida pelo mecanismo padrão de prioridades de *threads* oferecido pela linguagem Java. As *threads* adaptadas devem possuir outro atributo de prioridade específico para fins de simulação, de tal forma que permita ao controle da simulação identificar a *thread* que possua o menor próximo tempo de simulação e a maior prioridade para ser desbloqueada primeiro (conforme Figura 4.13). Tal atributo foi criado como um número inteiro, mantendo a lógica de prioridades com valores maiores significando *threads* com maior prioridade, isto é, que são desbloqueadas antes das de prioridades menores.

Uma vez resolvido o problema de implementação do bloqueio e desbloqueio de *threads* e o esquema de prioridades, resta a adaptação da *thread* de forma a permitir que ela trate exceções.

O esquema de execução sequencial dos comandos programados em uma *thread* foi concebido de forma a não permitir a preempção controlada, isto é, Java não possui mecanismos que permitam a programação de pausas na execução de um bloco de comandos para atender a uma requisição externa. Esta requisição externa deveria mover o ponteiro de execução da *thread* para o início de um método que contenha uma sequência de comandos que atendam à requisição externa (realizando a troca de contexto) e ao final da execução deste método o ponteiro de execução da *thread* voltaria à sequência que fora interrompida pela requisição externa, continuando a executar o próximo comando da sequência principal.

O método *interrupt()* pertencente à classe *Thread* permite que haja um desvio de execução de um determinado trecho de comandos para um outro, dentro de um bloco `try{} catch() {}`, mas não há retorno para continuidade de execução do trecho de comandos interrompido. Quando uma *thread* recebe uma chamada de interrupção pelo método *interrupt()* ocorre o levantamento de uma exceção (*InterruptException*), mesmo efeito utilizado na

implementação do desbloqueio da *thread*. Se os comandos que estão sendo executados pertencerem a um bloco de comandos *try*, a exceção levantada pela interrupção pode ser tratada localmente dentro de um bloco *catch*, mas, ao término de seu tratamento, não ocorre a continuação da execução dos comandos que foram interrompidos no bloco *try*.

A Figura 4.20 ilustra como ocorre o mecanismo de interrupção pelo método *interrupt()*, nativo na linguagem Java. Nesse exemplo, a interrupção foi gerada depois da execução do comando *C2*. O resultado é bem diferente ao que se deseja, uma vez que a sequência de execução dos comandos é interrompida (em *C2*) e não retorna a executar o comando *C3*, posterior à interrupção. Interrupções temporizadas também não teriam como ser implementadas a partir dessa abordagem.

Conforme exposto no item 4.4.5, o atendimento a interrupções pode ser implementado com o auxílio dos comandos de bloqueio e desbloqueio e uma lista contendo as chamadas de interrupções e temporizadores. Assim, após uma *thread* ser desbloqueada pelo comando *unblock()*, isto é, a barreira temporal (*wait()*) ter chegado ao fim, deve-se verificar se existe alguma interrupção levantada ou alguma temporização com tempo expirado. Se houver uma dessas duas situações, dá-se início à execução do método correspondente à interrupção chamada.

Na ocorrência de chamadas à interrupção que está sendo atendida, o registro da mesma não é realizado. Os métodos de interrupção devem ser registrados na *thread* e cada registro possui um atributo que marca quando a interrupção foi realizada; este atributo é desmarcado somente ao final da execução do método de interrupção.

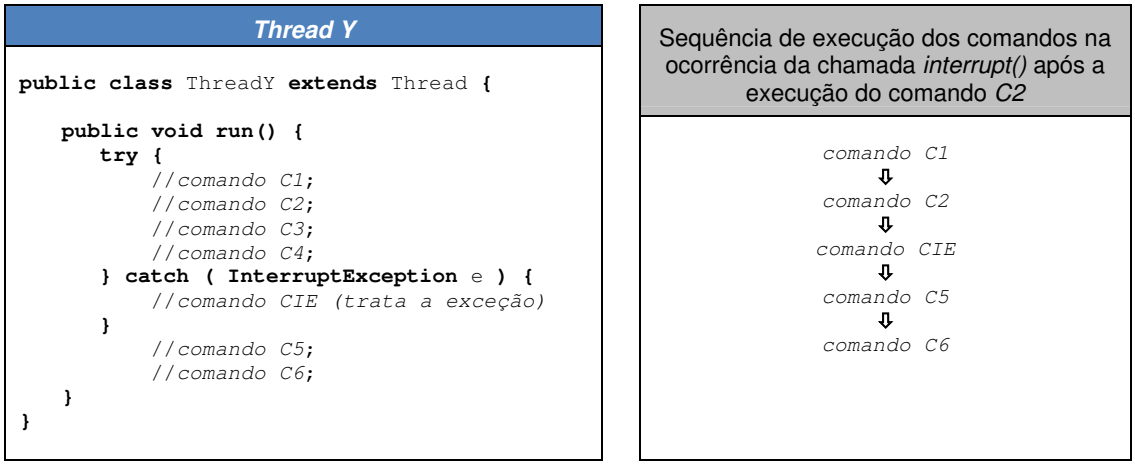


Figura 4.20 – Exemplo do código e execução de uma *thread* que trata a chamada de interrupção pelo método *interrupt()*. Na ausência de chamada de interrupção a sequência de comandos *C1* ⇨ *C2* ⇨ *C3* ⇨ *C4* ⇨ *C5* ⇨ *C6* é executada. Se ocorrer a chamada da interrupção após a execução do comando *C2*, os comandos *C3* e *C4* não serão executados. Após a chamada da interrupção, o comando de tratamento presente no bloco *catch* (comando *CIE*) é executado. Se os comandos *C5* e *C6* também estivessem agrupados no bloco *try*, eles também não seriam executados se ocorresse uma interrupção entre os comandos *C1* e *C4*.

Outro atributo de controle é utilizado para registrar quando os comandos que estão sendo executados pertencem a um método de interrupção (e não ao método principal), pois desse modo não é permitido que outras interrupções sejam executadas de forma entrelaçada. Na ocorrência de chamadas à interrupções diferentes da que estão sendo atendidas, o registro da interrupção é realizado, mas a sua execução só será realizada a posteriori, de acordo com o especificado no item 4.4.6.

4.6.2 – Estrutura e funcionamento do *framework* BaSS

O *framework* BaSS (*Barrier Synchronization Simulator*) é composto por catorze classes desenvolvidas na linguagem de programação Java e agrupadas em um único pacote denominado `br.eng.rsalustiano.bass`. A Tabela 4.2 apresenta as classes em ordem alfabética e as descrições sucintas de seus papéis no contexto de uma simulação. O diagrama de classes apresentado na Figura 4.21 (notação UML – *Unified Modeling Language* [MUL97]) ilustra a relação entre as classes do *framework* BaSS.

Tabela 4.2 – Classes do framework BaSS

Classe	Descrição
BaSSControl	Classe utilizada para realizar o controle da simulação.
BaSSControl.State	Classe do tipo enumeração utilizada para identificar os estados da simulação. Os estados podem ser: **COMPLETED** (a simulação chegou ao final), **FINISHED_BY_SIMULATION_TIME_LIMIT** (a simulação terminou antes do final devido a um tempo de simulação limite estabelecido pelo programador), **FINISHED_BY_WAITING_FOREVER** (a simulação chegou ao final, pois todas as *threads* ainda em execução estão paradas), **FINISHED_BY_WALLCLOCK_TIME_LIMIT** (a simulação terminou antes do final devido a um tempo de execução limite estabelecido pelo programador), **NOT_STARTED** (a simulação ainda não iniciou), **PAUSED** (a simulação está pausada), **RUNNIG** (a simulação está em execução) e **STOPPED** (a simulação foi forçada a parar antes do seu término).
BaSSDelay	Classe utilizada para inserir um atraso (*delay*) na execução do simulador.
BaSSDelay.Type	Classe do tipo enumeração utilizada para identificar o tipo de atraso (*delay)* relacionado a um objeto da classe BaSSDelay. Os tipos são: **EVENT_DELAY** (o atraso ocorre sempre que algum evento correr) ou **SIMULATION_TIME_DELAY** (o atraso ocorre toda vez que o tempo de simulação avançar).
BaSSInterruption	Classe utilizada para se registrar uma interrupção em um objeto do tipo *BaSSThread*.

BaSSInterruptionEvent	Superclasse abstrata que contém a descrição dos métodos comuns aos eventos de interrupção. As classes *BaSSTimer* e *BaSSInterruption* são as duas subclasses implementadas a partir desta superclasse.
BaSSLogger	Interface que contém a descrição dos métodos necessários para a implementação de *loggers* no simulador.
BaSSRandom	Classe que gera números pseudoaleatórios para serem utilizados na simulação.
BaSSStandardTime	Subclasse de *BaSSTime*, utilizada para representar o tempo de execução e podendo ser utilizada para representar o tempo de simulação no formato `YYyMMmDDdHH:mm:ss.SSS`, onde YY refere-se a anos, MM meses, DD dias, HH horas, mm minutos, ss segundos e SSS milésimos de segundos.
BaSSThread	Classe abstrata que contém a implementação da lógica base para o desenvolvimento de *threads* síncronas utilizadas na simulação.
BaSSTime	Classe que implementa o tempo da simulação. A representação do tempo nesta classe como um número é exata, de acordo com o padrão IEEE 754 [IEE08] de 128 bits (números com 34 dígitos a arredondamento HALF-EVEN).
BaSSTimer	Classe utilizada para registrar um temporizador (*timer*) a uma *BaSSThread*. É uma subclasse de *BaSSInterruptionEvent*.
BaSSUpdater	Interface que contém a descrição dos métodos necessários para a implementação de atualizadores (*updaters*) que poderão ser utilizados na simulação. Geralmente esses *updaters* são necessários para atualização de interfaces gráficas.
BaSSUpdater.Type	Classe do tipo enumeração utilizada para identificar o tipo atualizador (*updater)* relacionado a um objeto da classe BaSSUpdater. Os tipos são: **EVENT_UPDATER** (o método *update()* da classe BaSSUpdater é chamado sempre que algum evento correr) ou **SIMULATION_TIME_UPDATER** (o método *update()* da classe BaSSUpdater é chamado toda vez que o tempo de simulação avançar).

Das catorze classes do pacote, apenas três necessariamente devem estar presentes em todos os simuladores: uma subclasse de *BaSSThread*, a classe *BaSSControl* e a *BaSSTime* (ou subclasse desta, como a *BaSSStandardTime*, por exemplo).

De acordo com a necessidade de cada simulador desenvolvido, as outras classes são necessárias para a implementação de determinadas funcionalidades, como, por exemplo, a geração de sequências de números pseudoaleatórias (*BaSSRandom*), inclusão de interrupções (*BaSSInterruption*) e temporizadores (*BaSSTimer*), entre outras. Segue um detalhamento de como utilizar as classes do pacote no desenvolvimento de simuladores.

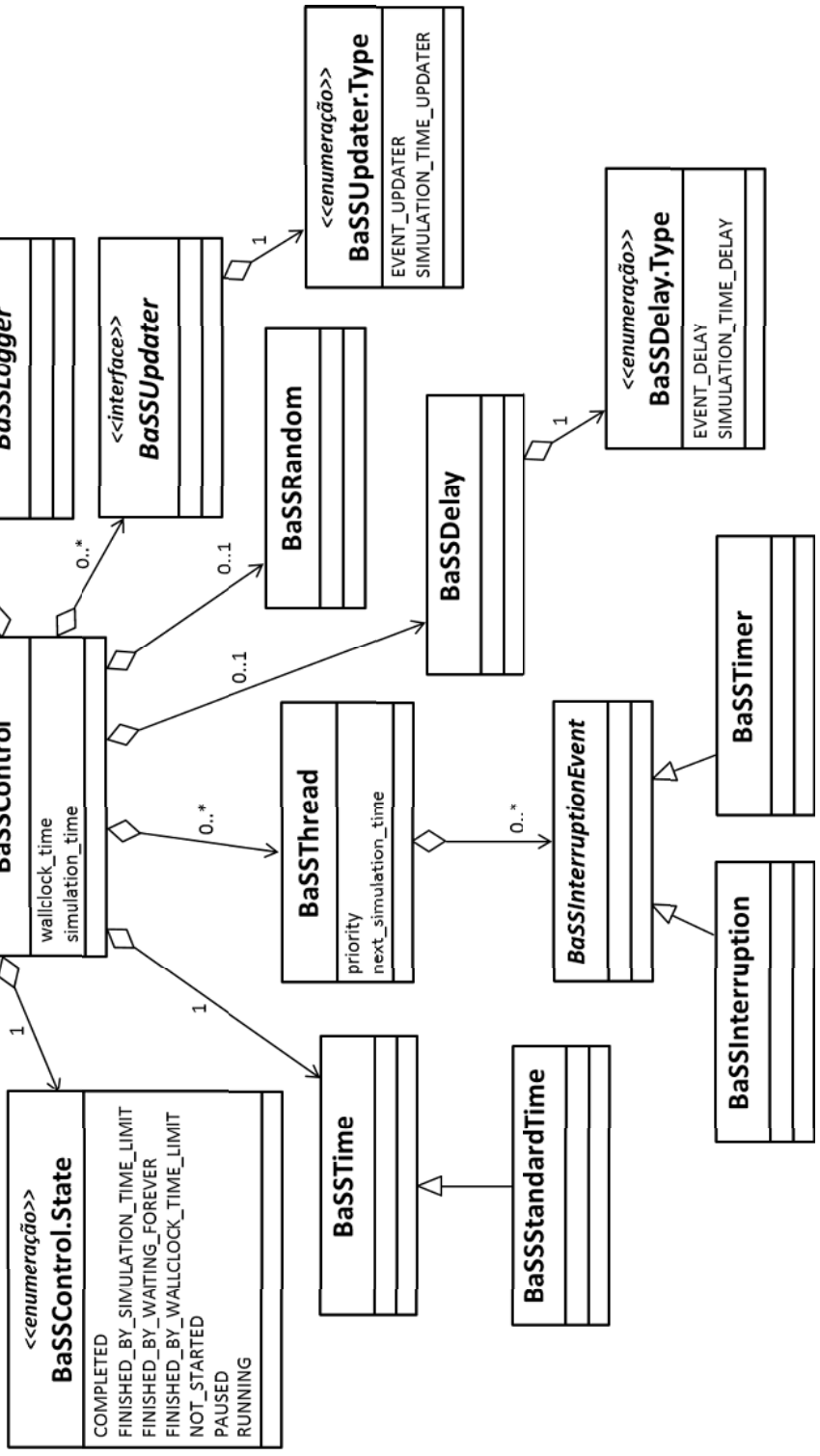


Figura 4.21 – Diagrama de classes do *framework* BaSS. Neste diagrama são exibidos apenas os valores das enumerações (classes BaSSControl.State, BaSSUpdater.Type e BaSSDelay.Type) e os atributos relacionados ao tempo e prioridade: *wallclock_time* refere-se ao tempo de execução da simulação; *simulation_time* refere-se ao tempo de simulação; *priority* refere-se à prioridade da *BaSSThread*; *next_simulation_time* refere-se ao tempo de simulação em que a *BaSSThread* termina de executar os comandos precedentes à barreira temporal e está disponível para o desbloqueio.

4.6.2.1 – Organização, funcionamento e controle da simulação

O primeiro parâmetro a ser considerado na criação de uma nova simulação é o tipo (ou escala) de tempo que será adotado. Tal seleção está diretamente ligada à natureza da simulação. Se os objetos da simulação executam seus comandos na escala de microssegundos e executam seus comandos durante pouco tempo de simulação, seria conveniente considerar a unidade numérica definida na classe *BaSSTime* como microssegundo ou segundo. A formatação do número poderia incluir a unidade (μs ou s) para efeitos de exibição do valor temporal, sendo isso conseguido através da implementação do método `toString()` em uma subclasse de *BaSSTime*. Uma simulação geológica, por outro lado, necessitaria de escalas de tempo da ordem de milhões/bilhões de anos, sendo conveniente criar uma subclasse de *BaSSTime* para esse propósito.

A classe *BaSSStandardTime* (subclasse de *BaSSTime*) representa o tempo da forma mais usual, isto é, na escala dos eventos em que o homem está acostumado a vivenciar no dia-a-dia. O tempo pode ser determinado em anos, meses, dias, horas, segundos e milissegundos. Tanto o método construtor como o método *toString()* consideram o tempo no formato `YYyMMmDDdHH:mm:ss.SSS`, onde YY refere-se a anos, MM minutos, DD dias, HH horas, mm minutos, ss segundos e SSS milésimos de segundos. A chamada do construtor `BaSSStandardTime( "11d03:25:32.264" )`, por exemplo, cria um objeto com valor temporal de 11 dias, 3 horas, 25 minutos e 32,264 segundos. Deve-se ressaltar que o valor armazenado internamente ao objeto *BaSSTime* (superclasse) é sempre um número representado no padrão IEEE 754, conforme já discutido no item 4.6.1, independente da forma com que o tempo é inserido nas subclasses.

O controle da simulação é realizado pela classe *BaSSControl*. Para se criar um objeto dessa classe deve-se fornecer como parâmetro a classe que será utilizada como referência de formatação e escala do tempo de simulação. Se, por exemplo, a classe *BASSStandardTime* for utilizada, a criação de um novo objeto se daria pelo comando `new BaSSControl( BaSSStandardTime.class )`.

Com relação ao controle do avanço temporal na simulação e a determinação de qual *BaSSThread* deve ser desbloqueada pelo *BaSSControl*, são utilizados os atributos *simulation_time* (tempo de simulação) do objeto da classe *BaSSControl*, *next_simulation_time* (próximo tempo de simulação) e *priority* (prioridade) dos objetos instanciados da classe *BaSSThread* (vide Figura 4.21). A cada desbloqueio do *BaSSControl*, há uma busca da *BaSSThread* que possua o menor *next_simulation_time* e a maior *priority*, sendo esta a próxima

thread a ser desbloqueada. Após esta determinação, o atributo *simulation_time* é atualizado com o valor de *next_simulation_time* da *BaSSThread* que será desbloqueada, avançando, assim, o tempo de simulação. Portanto, o avanço do tempo de simulação é realizado por saltos, conforme o paradigma de programação *event-driven*.

A lista contendo os objetos da classe *BaSSThread* não fica ordenada por tempo de simulação, mas sim por prioridade, uma vez que é mais eficiente (em termos computacionais) buscar a *BaSSThread* com menor *next_simulation_time* para desbloqueá-la do que ordenar a lista por *next_simulation_time* toda vez que uma *BaSSThread* atualiza esse atributo e é bloqueada. Como a lista de *BaSSThread* está ordenada de forma decrescente por prioridade, sempre será selecionada a *thread* que possui a maior prioridade. Contudo, se houver a troca de prioridades das *BaSSThreads* com frequência, o que depende da aplicação, a performance da simulação pode ser prejudicada, uma vez que a cada alteração de prioridade toda a lista de *BaSSThread* é reordenada.

O tempo de execução da simulação é calculado a partir do início da mesma. O valor presente no relógio do computador é registrado e, a cada consulta ao *wallclock_time*, uma subtração é realizada do valor presente no relógio no momento da consulta com o valor registrado no início da simulação. O tempo de execução é um objeto do tipo *BaSSStandardTime*.

No decorrer de uma simulação, o estado da mesma se modifica de acordo com os eventos envolvidos no processo de simulação, mas também pode ser alterado por ações externas à mesma (intervenção do usuário, por exemplo). A Figura 4.22 ilustra o diagrama de estados de simulação implementados na classe *BaSSControl*.

Quando um objeto do tipo *BaSSControl* é instanciado, o estado inicial de simulação é *NOT STARTED* (não iniciado). A partir do momento que é feita a chamada ao método `start()` do objeto de controle *BaSSControl*, a simulação se inicia, desbloqueando e bloqueando as *BaSSThreads* registradas no controle, passando o estado da simulação para *RUNNING* (executando). É possível pausar a simulação (parada no procedimento de bloqueio e desbloqueio das *BaSSThreads*), pelo método `pause()`, fazendo com que a simulação entre no estado *PAUSED* (pausado). O método `resume()` é utilizado para retornar à simulação ao estado *RUNNING*, depois de uma pausa. O tempo em que o simulador fica pausado não é contabilizado no tempo de simulação.

O método `pause()` deve ser utilizado com cautela quando chamando a partir de uma *BaSSThread*. Uma vez que a simulação é pausada de dentro de uma *BaSSThread*, somente um elemento externo à simulação poderá chamar o método `resume()`, pois toda a

simulação, isto é, todas as BaSSThreads da simulação, ficam bloqueadas.

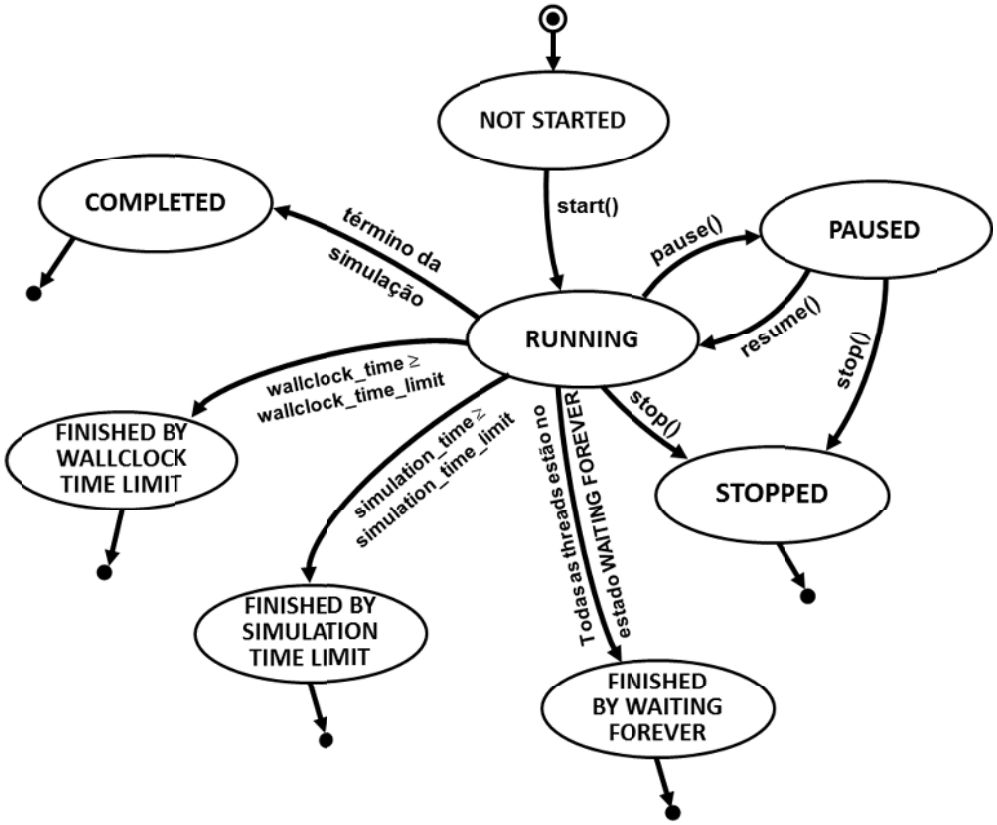


Figura 4.22 – Diagrama de estados de simulação implementados na classe *BaSSControl*.

Estando a simulação em execução (estado *RUNNING*) ou pausada (estado *PAUSED*), é possível finalizar a simulação na "força bruta" chamando o método `stop()`. Neste caso, a simulação termina no estado final STOPPED (parado), indicando que a simulação não chegou ao final, mas foi interrompida por um agente externo, possivelmente o usuário do simulador.

O método `stop()` não pode ser chamado a partir de uma BaSSThread, pois este método tentaria "destruir" a própria *BaSSThread* que o chamou, podendo provocar um travamento na execução da simulação ou havendo o levantamento de exceções relacionadas ao mecanismo de controle das *threads* na JVM.

Existem outras quatro formas da simulação chegar ao fim. A primeira ocorre quando a simulação de fato terminou, isto é, todos os comandos das *BaSSThreads* foram executados e não há mais nenhuma ação a ser realizada. Neste caso, o estado final da simulação é *COMPLETED* (concluída). A segunda situação ocorre quando todas as *BaSSThreads* estão no estado *WAITING FOREVER* (ver item 4.6.2.2), ou seja, estão ociosas aguardando alguma possível interrupção que as faça executar alguma ação. Mas se todas as *BaSSThreads* estão

aguardando indefinidamente, isso significa que nenhuma delas vai gerar uma interrupção para interromper outra, configurando um caso de *starvation*, levando a simulação ao término pelo estado *FINISHED BY WAITING FOREVER* (término por esperar para sempre).

O terceiro e quarto casos em que a simulação pode chegar ao fim referem-se a atributos que limitam o tempo de execução ou o tempo de simulação. Se o usuário/programador desejar que a simulação dure um determinado tempo de execução (20 minutos, por exemplo), ele deve atribuir esse tempo limite ao atributo *wallclock_time_limit* através do método *setWallclockTimeLimit()* da classe *BaSSControl*, passando como parâmetro um *BaSSStandardTime* que limitará o tempo de execução da simulação. Assim, quando o *wallclock_time* for maior ou igual ao *wallclock_time_limit*, a simulação termina, levando a simulação para o estado FINISHED BY WALLCLOCK TIME LIMIT (término por limite de tempo de execução). Da mesma forma, se o limitante de tempo não for baseado no tempo de execução da simulação, mas no tempo de simulação, deve-se conferir ao atributo *simulation_time_limit* o valor limite através do método *setSimulationTimeLimit()*. Se o *simulation_time* for maior ou igual ao *simulation_time_limit*, então a simulação termina e seu estado final será FINISHED BY SIMULATION TIME LIMIT (término por limite de tempo de simulação).

Uma característica particular das *threads* em Java é que elas não podem ser reiniciadas uma vez que tenham completado. A classe *BaSSThread* herda essa limitação. Portanto, uma vez que uma simulação tenha chegado ao fim, por qualquer motivo, ela não pode ser reiniciada a não ser o programa de simulação seja executado novamente. Uma estratégia que utiliza a propriedade reflexiva da linguagem Java (Java *reflection*) pode ser empregada para contornar esse problema. Tal estratégia será abordada no capítulo 5.

A classe *BaSSDelay* pode ser utilizada para adicionar um atraso no processamento dos comandos que estão em simulação. Este atraso é útil nas situações em que o processamento é demasiadamente rápido e não se consegue acompanhar os eventos em interfaces gráficas, por exemplo. Um *BaSSDelay* pode ser adicionado à uma *BaSSControl* através do método *setDelay()*. Existem dois tipos de atrasos, mas ambos possuem um determinado tempo (*wallclock time*) estabelecido em milissegundos que é inserido entre cada comando ou conjunto de comandos executados. O primeiro tipo é o EVENT_DELAY, no qual um atraso é inserido toda vez que uma *BaSSThread* da simulação é bloqueada. O segundo tipo é o SIMULATION_TIME_DELAY, no qual um atraso é criado apenas quando o *simulation_time* é alterado, ou seja, o tempo de simulação avança.

Quase a totalidade dos simuladores necessita que registro (*log*) seja criado para processamento dos dados gerados pelas simulações. A interface *BaSSLogger* permite que

mensagens provindas de todos os objetos pertencentes à simulação sejam registradas. Tal registro é realizado através do método `log( BaSSControl control, Object sender, Object... related )` que devem ser implementados nas classes implementadas a partir da interface *BaSSLogger*. O parâmetro *control* é o objeto que está realizando o controle da simulação e pode ser utilizado na obtenção do tempo de simulação no qual o registro ocorreu, por exemplo; o parâmetro *sender* diz respeito ao objeto que está realizando o registro do evento; e o parâmetro *related* pode ser utilizado para enviar objetos (quantos forem necessários) relacionados ao registro.

Todos objetos da simulação que desejam realizar um registro devem utilizar o método `log( Object sender, Object... related )` disponível na classe *BaSSControl*. Pode haver mais de um *BaSSLogger* registrado no *BaSSControl* (registro realizado pelo método `addLogger()`), sendo que todos eles recebem os registros quando o método *log* do *BaSSControl* é invocado.

As implementações da interface *BaSSLogger* também devem implementar os métodos `init()` e `finalize()`. Como os próprios nomes sugerem, esses métodos são chamados no ínicio e no final das simulações, respectivamente.

O *BaSSLogger* também é útil para registrar as exceções e erros reportados pela linguagem de programação Java. Ao invés de utilizar a saída padrão para exibir as mensagens reportadas pela JVM, pode-se redirecioná-las para este registro interno à simulação.

Além de registros para análise dos dados simulados, foi acrescentado ao modelo do BaSS uma classe de atualizadores, denominada *BaSSUpdater*. Elas são úteis para se realizar a atualização de interfaces gráficas durante a simulação. Assim como no caso da classe *BaSSDelay*, a atualização pode ser realizada a cada novo evento processado na simulação (EVENT_UPDATER) ou a cada avanço do tempo de simulação (SIMULATION_TIME_UPDATER). O método `getUpdaterType()` deve ser implementado e retornar um dos dois valores pertencentes à classe de enumeração *BaSSUpdater.Type*, EVENT_UPDATER ou SIMULATION_TIME_UPDATER.

Seja qual for o tipo de atualização, as ações relacionadas à ela devem ser realizadas no método *update(BaSSControl control)* da classe que será desenvolvida para esse fim a partir da interface *BaSSUpdater*. Os métodos `init()` e `finalize()` também devem ser implementados, referindo-se a ações que devem ser realizadas antes do início da simulação e ao término da mesma, respectivamente.

Como no caso dos registros (*BaSSLogger*), pode-se incluir mais de um *BaSSUpdater* nas simulações. Para adicioná-los no *BaSSControl*, deve-se utilizar o método

`addUpdater()` e toda a vez que o método `update()` da classe *BaSSControl* for chamado, o método `update( BaSSControl control )` de cada *BaSSUpdater* cadastrado no *BaSSControl* é invocado.

A diferença entre um evento de registro realizado pelo *BaSSLogger* e uma atualização realizada pelo *BaSSUpdater* está, portanto, relacionada a quando o método de registro `log( Object sender, Object... related )` da classe *BaSSLogger* é chamado e quando o método de atualização `update()` da classe *BaSSUpdater* é chamado. No primeiro caso, quem chama o método de registro são os objetos pertencentes à simulação, ficando a cargo do programador identificar a necessidade de se realizar um registro qualquer. Tal registro pode ser realizado, portanto, por qualquer classe pertencente ao simulador que tenha acesso ao objeto de controle da simulação (objeto da classe *BaSSControl*). No caso dos atualizadores, quem chama o método de registro é o próprio controle da simulação, sendo o momento de chamada controlado pelo tipo de *BaSSUpdater.Type* associado ao *BaSSUpdater*.

Tanto a classe *BaSSLogger* como a classe *BaSSUpdater* foram criadas como interfaces para permitir o desenvolvimento de classes que assumam diferentes papeis numa simulação. Uma subclasse de *BaSSThread* (uma classe que *extends BaSSThread*), por exemplo, pode ser uma classe que implementa um *BaSSLogger* (uma classe que *implements BaSSLogger*), que implementa um *BaSSUpdater* (uma classe que *implements BaSSUpdater*) ou ainda que implementa um *BaSSLogger* e um *BaSSUpdater* simultaneamente.

Tal flexibilidade, conseguida pelo modo com que as interfaces operam, permite a escrita de simuladores com um menor número de classes. Contudo, o registro dessas diferentes funcionalidades atribuídas a um mesmo objeto deve ser informado ao simulador, isto é, por mais que uma classe seja descrita na linguagem Java como uma *BaSSThread*, um *BaSSLogger* e um *BaSSUpdater* simultaneamente, deve-se adicionar ao controle da simulação (objeto da classe *BaSSControl*) o mesmo objeto instanciado a partir dessa classe utilizando os métodos `addThread( BaSSThread thread )`, `addLogger( BaSSLogger logger)` e `addUpdater( BaSSUpdater updater )` para que o objeto em questão assuma os três papeis na simulação. No caso de uma classe implementar simultaneamente as interfaces *BaSSLogger* e *BaSSUpdater*, os métodos `init( BaSSControl control )` e `finalize( BaSSControl control )` serão compartilhados pela implementação dessas interfaces, mas só são executados uma única vez, respectivamente, no início e no fim de cada simulação.

A Figura 4.23 apresenta em pseudocódigo a sequência de eventos realizados pelo *BaSSControl* no início e no fim de cada simulação, isto é, quando o método `start()` é chamado para iniciar a simulação e quando a simulação termina (`finalize()`), independente

do motivo do término (chamada ao método `stop()`, finalização por tempo limite, simulação concluída ou finalizada por espera infinita).

O BaSS ainda fornece uma classe que gera números pseudoaleatórios, muito útil em simulações nas quais eventos aleatórios devem ocorrer, mas que se faz necessário que esses eventos aleatórios se repitam em diversas execuções da simulação. A classe *BaSSRandom* pode ser introduzida no *BaSSControl* pelo método `setRandom( BaSSRandom random )` e acessada por qualquer objeto da simulação pelo método `getRandom()`. O construtor da classe *BaSSRandom* recebe a semente (um número inteiro) para geração de números pseudoaleatórios, garantindo assim uma repetição controlada dos número gerados a cada simulação realizada. Os métodos `nextInt()`, `nextDouble()` e `nextBaSSTime()`, pertencentes à classe *BaSSRandom* podem ser utilizados para gerar valore aleatórios para números inteiros, ponto flutuantes do tipo double e valores temporais do tipo BaSSTime, respectivamente.

```
1   init() {
1.1     Invoca o método init() de todos os BaSSLoggers;
1.2     Registra o horário de início da simulação (wallclock time);
1.3     Zera o tempo de simulação (simulation time);
1.4     Invoca o método init() de todas as BaSSThreads;
1.5     Invoca o método init() de todos os BaSSUpdaters;
1.6     Inicia todas as BaSSThreads, bloqueando-as;
1.7     Inicia a simulação (bloqueios e desbloqueios);
    }

2   finalize() {
2.1     Finaliza a simulação (bloqueios e desbloqueios);
2.2     Finaliza todas as BaSSThreads que ainda não estiverem finalizadas;
2.3     Invoca o método finalize() de todas as BaSSThreads que ainda não foram finalizadas;
2.4     Invoca o método finalize() de todos os BaSSUpdaters;
2.5     Invoca o método finalize() de todos os BaSSLoggers;
2.6     Força a JVM a fazer uma limpeza na memória (chama o garbage collector);
    }
```

Figura 4.23 – Pseudocódigo das ações realizadas no início (*init*) e finalização (*finalize*) da simulação.

4.6.2.2 – Estrutura e ciclo de vida de uma *BaSSThread*

A classe *BaSSThread* foi desenvolvida tendo como base a classe *Thread* do pacote *java.lang*, a qual permite a execução de comandos do método `run()` de forma pseudo-paralela com outras *threads*. Por se tratar de uma classe abstrata, para se instanciar um objeto com as funcionalidades da *BaSSThread* é preciso que seja definida uma subclasse que implemente os métodos abstratos `init()`, `main()` e `finalize()`.

Esses métodos fazem parte do ciclo de vida da *BaSSThread* (Figura 4.24). O método `init()` sempre é executado no início da simulação, antes que qualquer atualização no tempo de simulação seja realizada. A utilidade desse método reside na possibilidade de se realizarem configurações iniciais no objeto antes do início da simulação. Como podem existir diversas *BaSSThreads* na simulação e no início todos os relógios estão zerados, então a ordem de execução dos métodos `init()` de cada *BaSSThread* segue a ordem decrescente de suas prioridades (atributo *priority*).

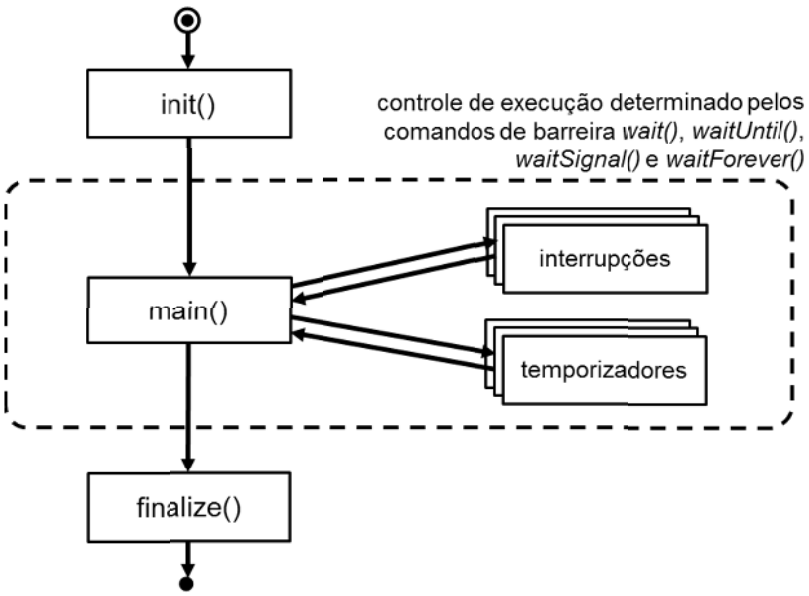


Figura 4.24 – Ciclo de vida de uma *BaSSThread*.

Após a execução do método `init()` de todas as *BaSSThreads* presentes na simulação, todas elas são bloqueadas pelo *BaSSControl*. A partir do primeiro desbloqueio, o método `main()` começa a ser executado.

Dentro do método `main()` devem ser criadas as barreiras temporais para sincronizar a execução dos comandos com os comandos das outras *BaSSThreads*. Foram criadas quatro tipos de barreiras, estabelecidas a partir dos métodos `wait( BaSSTime time )`,

`waitUntil( BaSSTime time )`, `waitSignal()` e `waitForever()`.

O método `wait( BasSSTime time )` determina um certo intervalo de tempo de simulação para a criação da próxima barreira temporal. Assim, deve-se passar como parâmetro um objeto da classe *BaSSTime* (ou uma subclasse desta), como por exemplo `wait( new BaSSStandardTime( "00:01:00" ) )`, que faz com que uma barreira seja criada 1s adiante no tempo, a partir do tempo de simulação atual (*simulation_time* + 1s). Esse intervalo de tempo está relacionado ao tempo utilizado no processamento dos comandos imediatamente anteriores à barreira estabelecida (o mesmo raciocínio explorado no item 4.4.3).

O método `waitUntil( BaSSTime time )` também recebe um objeto do tipo *BaSSTime* (ou subclasse desta), contudo o tempo fornecido como parâmetro especifica o instante temporal da próxima barreira e não um intervalo de tempo até lá. O comando `waitUntil( new BaSSStandardTime( "00:04:32" ) )` cria uma barreira temporal no instante 4 minutos e 32 segundos. Deve-se ressaltar, nesse caso, que se a barreira temporal criada tiver um instante temporal que já ocorreu no tempo de simulação, isto é, o instante fornecido como parâmetro é menor ou igual ao atributo *simulation_time*, a barreira não é criada e o objeto continua a sua execução como se nenhum comando de barreira existisse.

O terceiro método de barreira, `waitSignal()`, faz com que a *BaSSThread* fique bloqueada até que algum objeto chame o método `signal()`. Trata-se, portanto, de uma barreira não-temporal, uma vez que o seu desbloqueio não depende do tempo de simulação, mas sim de uma ação externa (provinda de outro objeto) que resulte na chamada do método `signal()` do objeto bloqueado.

A quarta e última forma de se criar uma barreira é através do método `waitForever()`. Este método faz com que a *BaSSThread* fique bloqueada indeterminadamente, sendo útil nos casos em que a *BaSSThread* não executa nenhum comando no método `main()`, mas precisa estar "viva" para atender à chamadas de interrupções ou temporizadores.

Quando o método `main()` é executado até o final, o ciclo de vida da BaSSThread atinge o seu último estágio, no qual são executados os comandos do método `finalize()`. Trata-se de um método análogo ao `init()`, mas sendo executado no final da execução dos comandos da *BaSSThread*. A utilização de qualquer método de barreira tanto dentro do método `finalize()` como dentro do método `init()` é proibitiva.

A Figura 4.25 apresenta o exemplo da classe *BThread01*, implementação de uma *BaSSThread*, na qual são definidos os métodos `init()`, `main()` e `finalize()`. Os "comandos" da simulação presentes são métodos que imprimem na tela (`System.out.println()`) o tempo

de simulação (`super.getControl(). getSimulationTime().toString()`), o nome da *BaSSThread* e uma letra.

```
01  import br.eng.rsalustiano.bass.BaSSThread;
02  import br.eng.rsalustiano.bass.BaSSControl;
03  import br.eng.rsalustiano.bass.BaSSStandardTime;
04
05  public class BThread01 extends BaSSThread {
06
07      public BThread01( String name, int priority ) {
08          super( name );
09          super.setPriority( priority );
10      }
11
12      public void init() {
13          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                            super.getName() + " A" );
14      }
15
16      public void main() {
17          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                            super.getName() + " B" );
18          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                            super.getName() + " C" );
19          super.wait( new BaSSStandardTime( "00:01:00" ) );
20          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                            super.getName() + " D" );
21          super.wait( new BaSSStandardTime( "00:02:30" ) );
22          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                            super.getName() + " E" );
23          super.wait( new BaSSStandardTime( "00:05:00" ) );
24          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                            super.getName() + " F" );
25          super.wait( new BaSSStandardTime( "00:01:30" ) );
26      }
27
28      public void finalize() {
29          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                            super.getName() + " G" );
30          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                            super.getName() + " H" );
31      }
32
33  }
```

Figura 4.25 – Definição da classe *BThread01*.

```
01  public static void main( String args[] ) {
02      BaSSControl bt = new BaSSControl( BaSSStandardTime.class );
03      bt.addThread( new BThread01( "TH1", 0 ) );
04      bt.addThread( new BThread01( "TH2", 0 ) );
05      bt.addThread( new BThread01( "TH3", 0 ) );
06      bt.start();
07  }
```

Figura 4.26 – Método *main()* que realiza a criação de uma simulação com três objetos da classe *BThread01* e executa a simulação.

```
00:00:00.000 TH1 A
00:00:00.000 TH2 A
00:00:00.000 TH3 A
00:00:00.000 TH1 B
00:00:00.000 TH1 C
00:00:00.000 TH2 B
00:00:00.000 TH2 C
00:00:00.000 TH3 B
00:00:00.000 TH3 C
00:01:00.000 TH1 D
00:01:00.000 TH2 D
00:01:00.000 TH3 D
00:03:30.000 TH1 E
00:03:30.000 TH2 E
00:03:30.000 TH3 E
00:08:30.000 TH1 F
00:08:30.000 TH2 F
00:08:30.000 TH3 F
00:10:00.000 TH1 G
00:10:00.000 TH1 H
00:10:00.000 TH2 G
00:10:00.000 TH2 H
00:10:00.000 TH3 G
00:10:00.000 TH3 H
```

Figura 4.27 – Execução (saída na tela) da simulação criada na Figura 4.26.

4.6.2.3 – Prioridades, interrupções e temporizadores nas *BaSSThreads*

O programa apresentado na Figura 4.26 considera que todos os objetos instanciados da classe *BThreads01* possuem a mesma prioridade (zero). De acordo com o modelo de simulação por barreiras proposto e implementado no *framework* BaSS, é possível atribuir prioridades às *BaSSThreads* com o intuito de ordenar a sequência de execução de comandos quando mais de uma *BaSSThread* estão bloqueadas em barreiras de mesmo tempo de simulação.

A Figura 4.28 cria os mesmos três objetos da classe *BThread1* da Figura 4.25, mas agora o objeto com nome TH1 possui prioridade 1 (menor prioridade), o objeto com nome TH2 possui prioridade 3 (maior prioridade) e o objeto TH3 possui prioridade 2. O resultado da execução dessa simulação está na Figura 4.29.

```
01  public static void main( String args[] ) {
02      BaSSControl      bt      =      new      BaSSControl(
03  BaSSStandardTime.class );
04      bt.addThread( new BThread01( "TH1", 1 ) );
05      bt.addThread( new BThread01( "TH2", 3 ) );
06      bt.addThread( new BThread01( "TH3", 2 ) );
07      bt.start();
    }
```

Figura 4.28 – Método *main()* que realiza a criação da simulação com três objetos da classe *BThread01*, cada um com uma prioridade.

```
00:00:00.000 TH2 A
00:00:00.000 TH3 A
00:00:00.000 TH1 A
00:00:00.000 TH2 B
00:00:00.000 TH2 C
00:00:00.000 TH3 B
00:00:00.000 TH3 C
00:00:00.000 TH1 B
00:00:00.000 TH1 C
00:01:00.000 TH2 D
00:01:00.000 TH3 D
00:01:00.000 TH1 D
00:03:30.000 TH2 E
00:03:30.000 TH3 E
00:03:30.000 TH1 E
00:08:30.000 TH2 F
00:08:30.000 TH3 F
00:08:30.000 TH1 F
00:10:00.000 TH2 G
00:10:00.000 TH2 H
00:10:00.000 TH3 G
00:10:00.000 TH3 H
00:10:00.000 TH1 G
00:10:00.000 TH1 H
```

Figura 4.29 – Execução (saída na tela) da simulação criada na Figura 4.28. A sequência de execução mudou em relação à Figura 4.27 devido à introdução de prioridades diferentes aos objetos da classe *BThread01*.

A classe *BThread02* da Figura 4.30 apresenta uma *BaSSThread* com uma interrupção definida na linha 16 pelo nome `int0` e que referindo-se à execução do método `IntMethod01`. A classe *BThread03* apesentada na Figura 4.31 faz uma chamada à interrupção `int0` de uma *BaSSThread* denominada *TH2* (linha 19). O método `getThreadByName()` da classe *BaSSControl* permite que se busque uma *BaSSThread* cadastrada na simulação tendo como argumento o nome dessa *thread*. Já o método `callIterruption()` da classe *BaSSThread* realiza a chamada da interrupção.

A Figura 4.32 apresenta a configuração e execução de uma simulação que insere em um *BaSSControl* duas *BaSSThreads*, uma da subclasse *BThread02* (cujo nome é definido como *TH2*) e outra da subclasse *BThread03* (cujo nome é *TH3*). A execução da simulação é apresentada na Figura 4.33. A sequência de comandos executados segue conforme estabelecido nos métodos `main()` de cada *BaSSThread* até a chamada da interrupção realizada da *TH3* para a *TH2*, dois segundos após o início da simulação. Contudo, entre o primeiro e o terceiro segundo de simulação, a TH2 está executando o comando que imprime na tela a letra B. Portanto, a interrupção gerada pela TH3 só poderá ser atendida no tempo de 3s.

Com relação aos temporizadores, a Figura 4.34 apresenta a definição da classe *BThread04* que possui um temporizador denominado `timer0` que faz a requisição de execução do conteúdo do método *TimerMethod* a cada 1s de simulação (vide linha 15). Com a criação de uma simulação com uma *BThread04* (Figura 4.35), teremos a sequência de execução dos

comandos conforma a Figura 4.36. Nota-se que o atendimento à temporização não ocorre de 1 em 1 segundo, mas assim que a barreira na qual a *thread* está bloqueada é desbloqueada, conforme já estabelecido no item 4.4.5.

```
01  import br.eng.rsalustiano.bass.BaSSThread;
02  import br.eng.rsalustiano.bass.BaSSControl;
03  import br.eng.rsalustiano.bass.BaSSInterruption;
04  import br.eng.rsalustiano.bass.BaSSInterruptionEvent;
05  import br.eng.rsalustiano.bass.BaSSStandardTime;
06
07  public class BThread02 extends BaSSThread {
08
09      public BThread02( String name ) {
10          super( name );
11          super.setPriority( 2 );
12      }
13
14      public void init() {
15          try {
16              super.addInterruptionEvent( new BaSSInterruption( "int0", "IntMethod01",
                                                                  0, true ) );
17          } catch( Exception e ) {
18              e.printStackTrace();
19          }
20      }
21
22      public void main() {
23          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                                super.getName() + " A" );
24          super.waitFor( new BaSSStandardTime( "00:01:00" ) );
25          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                                super.getName() + " B" );
26          super.waitFor( new BaSSStandardTime( "00:02:00" ) );
27          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                                super.getName() + " C" );
28          super.waitFor( new BaSSStandardTime( "00:01:00" ) );
29          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                                super.getName() + " D" );
30          super.waitFor( new BaSSStandardTime( "00:01:00" ) );
31      }
32
33      public void IntMethod01() {
34          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                                super.getName() + " Int0.A" );
35          super.waitFor( new BaSSStandardTime( "00:01:00" ) );
36      }
37
38      public void finalize() { }
39
40  }
```

Figura 4.30 – Definição da classe *BThread02*.

```
01  import br.eng.rsalustiano.bass.BaSSThread;
02  import br.eng.rsalustiano.bass.BaSSControl;
03  import br.eng.rsalustiano.bass.BaSSStandardTime;
04
05  public class BThread03 extends BaSSThread {
06
07      public BThread03( String name ) {
08          super( name );
09          super.setPriority( 1 );
10      }
11
12      public void init() { }
13
14      public void main() {
15          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                                super.getName() + " A" );
16          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                                super.getName() + " B" );
17          super.wait( new BaSSStandardTime( "00:02:00" ) );
18          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                                super.getName() + " C" );
19          super.getControl().getThreadByName( "TH2" ).callInterruption( "int0" );
20          super.wait( new BaSSStandardTime( "00:03:00" ) );
21          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                                super.getName() + " D" );
22          super.wait( new BaSSStandardTime( "00:01:00" ) );
23          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                                super.getName() + " E" );
24          super.wait( new BaSSStandardTime( "00:02:00" ) );
25      }
26
27      public void finalize() { }
28
29  }
```

Figura 4.31 – Definição da classe *BThread03*.

```
01  public static void main( String args[] ) {
02      BaSSControl bt = new BaSSControl( BaSSStandardTime.class );
03      bt.addThread( new BThread02( "TH2" ) );
04      bt.addThread( new BThread03( "TH3" ) );
05      bt.start();
06  }
```

Figura 4.32 – Método *main()* que realiza a criação da simulação com objetos da classe *BThread02* e *BThread03*.

```
00:00:00.000 TH2 A
00:00:00.000 TH3 A
00:00:00.000 TH3 B
00:01:00.000 TH2 B
00:02:00.000 TH3 C
00:03:00.000 TH2 Int0.A
00:04:00.000 TH2 C
00:05:00.000 TH2 D
00:05:00.000 TH3 D
00:06:00.000 TH3 E
```

Figura 4.33 – Execução (saída na tela) da simulação criada na Figura 4.32.

```
01  import br.eng.rsalustiano.bass.BaSSThread;
02  import br.eng.rsalustiano.bass.BaSSControl;
03  import br.eng.rsalustiano.bass.BaSSStandardTime;
04  import br.eng.rsalustiano.bass.BaSSTimer;
05
06  public class BThread04 extends BaSSThread {
07
08      public BThread04( String name, int priority ) {
09          super( name );
10          super.setPriority( priority );
11      }
12
13      public void init() {
14          try {
15              super.addInterruptionEvent( new BaSSTimer( "timer0", "TimerMethod",
                                        new BaSSStandardTime( "00:01:00" ), 1, true ) );
16          } catch( Exception e ) {
                e.printStackTrace();
17          }
18      }
19
20      public void main() {
21          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                            super.getName() + " A" );
22          super.wait( new BaSSStandardTime( "00:02:00" ) );
23          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                            super.getName() + " B" );
24          super.wait( new BaSSStandardTime( "00:03:00" ) );
25          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                            super.getName() + " C" );
26          super.wait( new BaSSStandardTime( "00:01:00" ) );
27          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                            super.getName() + " D" );
28          super.wait( new BaSSStandardTime( "00:02:00" ) );
29      }
30
31      public void TimerMethod() {
32          System.out.println( super.getControl().getSimulationTime().toString() + " " +
                            super.getName() + " timer" );
33          super.wait( new BaSSStandardTime( "00:00:30" ) );
34      }
35
36      public void finalize() { }
37
38  }
```

Figura 4.34 – Definição da classe *BThread04*.

```
01  public static void main( String args[] ) {
02       BaSSControl bt = new BaSSControl( BaSSStandardTime.class );
03       bt.addThread( new BThread04( "TH4", 0 ) );
04       bt.start();
05  }
```

Figura 4.35 – Método *main()* que realiza a criação da simulação com um objeto da classe *BThread04*.

O Apêndice C apresenta um exemplo de simulação mais completo e complexo, no qual são instanciadas quatro objetos da classe *BaSSThread* contendo interrupções e temporizadores. A classe *TestThread* (Figura C.1) definida nesse apêndice é a implementação das threads definidas com pseudocódigo na Figura 4.14. A execução da simulação, conforme esperado, resulta na sequência de comandos apresentada na Figura 4.16.

```
00:00:00.000 TH4 timer
00:00:30.000 TH4 A
00:02:30.000 TH4 timer
00:03:00.000 TH4 timer
00:03:30.000 TH4 B
00:06:30.000 TH4 timer
00:07:00.000 TH4 timer
00:07:30.000 TH4 C
00:07:30.000 TH4 D
00:09:30.000 TH4 timer
00:10:00.000 TH4 timer
```

Figura 4.36 – Execução (saída na tela) da simulação criada na Figura 4.35.

4.6.3 – Testes: verificação, desempenho e validação do BaSS

O *framework* BaSS foi submetido a diversos tipos de testes para verificar se a sua implementação está em acordo com as especificações propostas no modelo de simulação. As referências [MAY90] e [SOM01] foram utilizadas como principal guia para o procedimento de teste de software.

Inicialmente foi realizada uma inspeção minuciosa nas classes do *framework*, pela releitura do código em busca de possíveis inconsistências e ausência de elementos. Em [FAG86] é ressaltado que mais de 60% dos erros podem ser detectados pela simples inspeção informal do programa e, de fato, a inspeção do BaSS levou a alteração da estrutura de muitas classes e pode-se corrigir alguns erros de inicialização de atributos, eliminação de atributos e variáveis desnecessárias, adição de métodos faltantes (principalmente os *gets* e *sets*), entre outros.

Uma análise estática foi realizada tendo como base o fluxograma da Figura 4.13 e a releitura das classes *BaSSControl* e *BaSSThread*. Tal análise manual permite um aumento da confiabilidade, constatando de forma mais adequada que a programação realizada está em acordo com o modelo proposto [MAY90]. O próximo passo foi a realização de uma análise dinâmica, envolvendo o teste estrutural (*White Box Test*) e funcional (*Black Box Test*).

O teste estrutural foi realizado com a execução de diversas subclasses da *BaSSThread* contendo temporizadores (*BassTimer*) e interrupções (*BaSSInteruption*) cadastradas. Ao controle da simulação (*BaSSControl*) foram adicionados objetos de subclasses de *BaSSLogger*, *BaSSUpdater* e *BaSSRandom*, sendo que o acesso a essas últimas a partir dos objetos das classes de *BaSSThread* ocorre através dos métodos de `log( Object sender, Object... related )`, `update()` e `getRandom()` disponíveis no objeto da classe *BaSSControl*. Toda *BaSSThread* tem acesso ao controle (*BaSSControl*) pelo método `getControl()` quando é inserida em uma simulação. Esta estrutura de software se mostrou eficiente uma vez que não há a necessidade de replicação de objetos de registro, atualização e

gerador de números aleatórios para cada *BaSSThread* da simulação.

A realização do teste funcional não é um procedimento simples, uma vez que a própria natureza do núcleo de simulação é complexa. Contudo, as funcionalidades do BaSS puderam ser testadas de forma semelhante à descrição e explicação teórica do funcionamento do núcleo de simulação realizada neste capítulo. Valendo-se de sequências de execução de comandos de *threads* e uma linha de tempo na qual se pode alocar de forma manual os comandos executados por elas, pode-se comparar o resultado obtido com o resultado de uma simulação realizada pelo BaSS com a mesma sequência de comandos das *threads* realizadas por objetos da classe *BaSSThread*.

Apesar de parecer uma solução de teste um tanto quanto limitada, um teste automatizado para o BaSS seria algo inviável, uma vez que não se pode produzir uma expressão analítica do seu funcionamento por se tratar de um conjunto de classes em execução paralela e também não se pode criar um programa que produza sequências de execução de comandos idênticas a que se espera na saída de uma simulação, visto que a elaboração de tal programa também teria uma complexidade semelhante.

A Tabela 4.3 apresenta um conjunto de testes funcionais criados para averiguar o funcionamento do BaSS. Buscou-se executar as funcionalidades do BaSS de forma isolada e em seguida combinar as funcionalidades. Para cada teste foram criados cinco cenários em prancheta e programados em subclasses da *BaSSThread*, verificando, em seguida, a execução da simulação no computador com o desenvolvido na prancheta.

Esse método de teste, por mais simples que possa parecer, conduziu a etapas de correção de erros cíclicas, uma vez que todas as inconsistências encontradas entre a execução manual do algoritmo e a execução do BaSS no computador gerava correções tanto no modelo, como na implementação nas classes. Apesar da demora na execução devido a sua parte manual e verificação da igualdade nos resultados, esse tipo de teste funcional gerou resultados importantes no que diz respeito ao aperfeiçoamento do software.

Com relação aos testes de desempenho, eles foram realizados em dois computadores pessoais: (1) um *desktop* com processador AMD Phenom II X6 (3,30GHz, *cache* L1 128kB x6, *cache* L2 512kB, *cache* L3 6144kB) e 8GB de memória RAM; e (2) um *notebook* com processador Intel i7-3520M com dois núcleos de processamento capazes que executar 4 *thread* simultaneamente pelo mecanismo de *Hyper-Threading* (2,9GHz, *cache* L1 128kB x2, *cache* L2 256kB x2 e *cache* L3 4MB) e 6GB de memória RAM. O sistema operacional utilizado em ambas as máquinas foi o Windows Seven 64-bit com o JDK (*Java Development Kit*) e o JRE (*Java Runtime Environment*) instalados na versão 1.7.0_02 (64-bit). Todos os processos não necessários ao funcionamento do sistema operacional foram desabilitados e os adaptadores de

rede desativados antes da execução dos testes.

Tabela 4.3 – Testes funcionais realizados no BaSS

Teste	Objetivo
uma *BaSSThread*	Verificar a sequência de comandos gerada por uma única *BaSSThread*.
N *BaSSThreads* com prioridades diferentes	Verificar a sequência de comandos gerada com diversas *BaSSThreads*, cada uma com uma prioridade diferente. *BaSSThreads* com prioridades maiores devem ser executadas primeiro quando desbloqueadas de uma mesma barreira temporal.
duas *BaSSThreads* com prioridades diferentes, uma realizando interrupções na outra	Verificar o mecanismo de interrupção e preempção associado. Enquanto uma *BaSSThread* executa os comandos do método principal (*main()*), a outra *BaSSThread* gera uma interrupção que interrompe a execução do método principal, executa o método relacionado à interrupção e em seguida retorna à execução dos comandos do método principal interrompido.
N *BaSSThreads* com prioridades diferentes, algumas realizando interrupções nas outras; interrupções com diferentes prioridades.	Verificar o mecanismo de interrupção e preempção associado, utilizando diversas *BaSSThreads*. Verificar a sequência de execução de interrupções que ocorram simultaneamente de acordo com suas prioridades.
uma *BaSSThread* com um temporizador	Verificar o mecanismo de temporização e preempção associado, utilizando apenas uma *BaSSThread* e um temporizador.
uma *BaSSThread* com N temporizadores com prioridades diferentes	Verificar o mecanismo de temporização e preempção associado, utilizando diversos temporizadores com prioridades diferentes. Na ocorrência de eventos de temporização na mesma barreira temporal, deve-se verificar que o temporizador que possuir a maior prioridade deve ser executado primeiro.
N *BaSSThreads* com prioridades diferentes e com N temporizadores cada, também com prioridades diferentes	Verificar a sequência de execução dos comandos na existência de diversos temporizadores com prioridades diferentes em cada uma das *BaSSThreads*.
duas *BaSSThread* com prioridades diferentes, uma realizando interrupção na outra e esta também possuindo temporizador. A interrupção e a temporização possuem prioridades distintas.	Verificar a sequência de comandos executada na ocorrência de interrupção e temporização em uma mesma *BaSSThread*. Checar que eventos (interrupção ou temporizador) com maior prioridade devem ser atendidos antes dos demais.
N *BaSSThread*, com prioridades diferentes, algumas realizando interrupções nas outras e algumas com temporizadores	Neste caso todas as possibilidades podem ocorrer. Deve-se verificar a sequência de execução de cada *BaSSThread*, respeitando suas prioridades; a ocorrência de temporizadores e interrupções, também respeitando suas prioridades.

O desempenho do *framework* BaSS foi medido em termos de tempo de execução da classe *TestTime* (Figura 4.37). A execução dessa *BaSSThread* consta apenas de uma operação de incremento do atributo *v* a cada 1s de tempo de simulação. A simulação foi executada durante 24 horas de tempo de simulação, resultando, portanto, 86.400 operações de incremento do atributo *v*. Para a execução de apenas uma *TestTime* foram utilizados 2,79s (AMD) e 2,52s (Intel) de tempo de execução para completar a simulação e uma média de 32,2939µs (AMD) e 29,2009µs (Intel) por operação de incremento. Este valor médio engloba todas as operações de controle da simulação envolvidas, não sendo, portanto, o tempo gasto exclusivamente pela operação de incremento.

A maior questão envolvida no desempenho de um software *multi-thread* como o BaSS está relacionada ao aumento no tempo de execução proporcionado pela introdução de uma nova *BaSSThread* na simulação. Sendo assim, foram realizadas 500 simulações tendo como base objetos da classe *TestTime*, sendo que cada simulação continha uma quantidade de objetos *TestTime* variando de 1 a 500.

```
01  import br.eng.rsalustiano.bass.BaSSThread;
02  import br.eng.rsalustiano.bass.BaSSControl;
03  import br.eng.rsalustiano.bass.BaSSStandardTime;
04
05  public class TestTime extends BaSSThread {
06
07      private BaSSStandardTime delay = new BaSSStandardTime( "00:00:01.000" );
08      private int v = 0;
09
10      public TestTime( String name ) {
11          super( name );
12      }
13
14      public void init() {
15      }
16
17      public void main() {
18          while ( true ) {
19              v++;
20              super.waitFor( delay );
21          }
22      }
23
24      public void finalize() {
25          System.out.println( super.getControl().getWallclockTime().getTime() );
26      }
```

Figura 4.37 – Definição da classe *TestTime*.

O resultado do teste de desempenho do BaSS com a variação do número de *BaSSThreads* pode ser analisado nos gráficos das Figuras 4.38, 4.39, 4.40 e 4.41.

O tempo de execução necessário para completar a simulação das 24h de tempo de simulação em relação ao número de *BaSSThreads* executadas no simulador é apresentado na Figura 4.38. O comportamento assintótico da curva permite classificar a função de crescimento

como *O(n)*, podendo ser considerado satisfatório no contexto de *multi-threads*.

Os artigos nos quais os *frameworks* de simulação são apresentados (vide item 4.5) não realizam uma análise de desempenho comparável à realizada e cujo resultado é apresentado na Figura 4.38. Esta ausência desse tipo de teste pode ser explicada pela natureza do *framework* desenvolvido não se basear em *threads* ou pela simples omissão de tal análise.

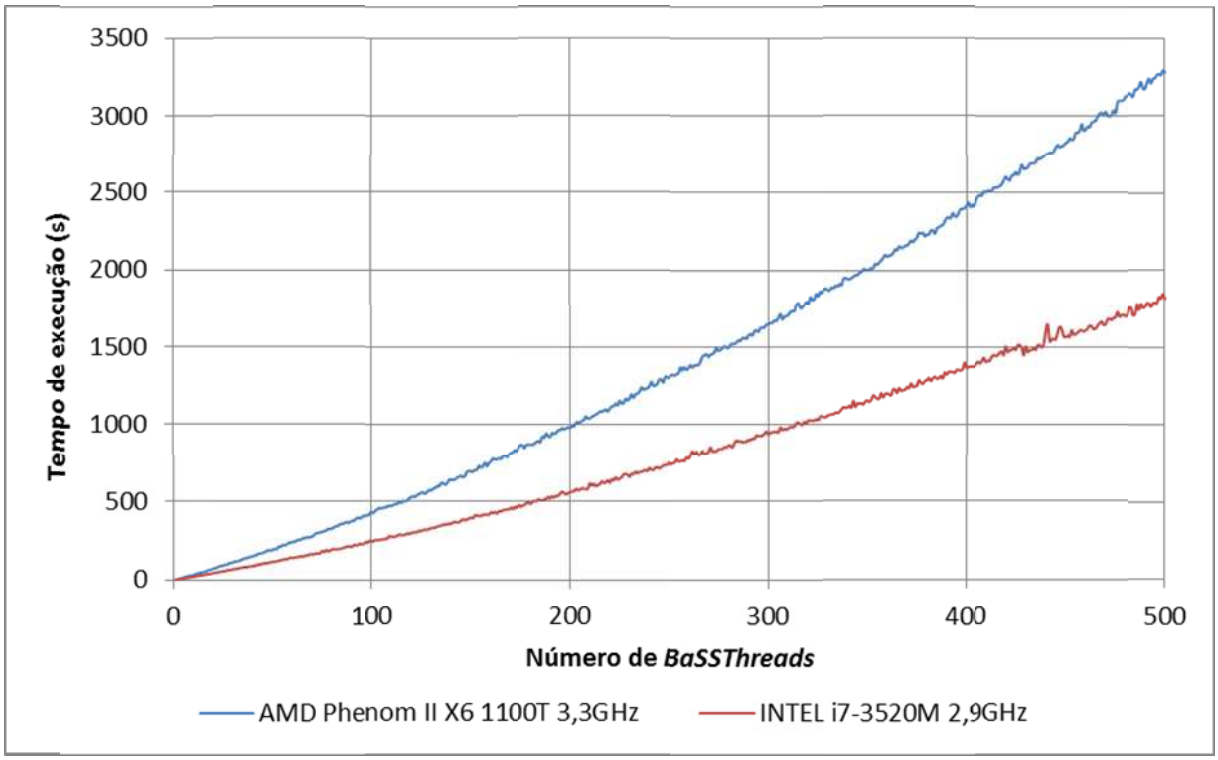


Figura 4.38 – Tempo de execução utilizado para simular 24h de tempo de simulação de acordo com o número de *BaSSThreads* (*TimeThread* definida na Figura 4.37).

Os gráficos das Figuras 4.39 e 4.40 são complementares. O primeiro apresenta o tempo de execução médio para a execução de uma operação de incremento e o segundo o número médio de operação por segundo em tempo de execução.

Em [GOE05], artigo no qual é apresentado a biblioteca JSDESLib, uma análise do tempo de resposta para a execução de eventos (não especificados) em função do número de entidades (similar ao número de *threads*), resultou em valores na escala de milissegundos desde uma entidade até 1024 entidades. Tais tempos são maiores se comparados aos valores apresentados no gráfico da Figura 4.39, cuja ordem de grandeza é o microssegundo.

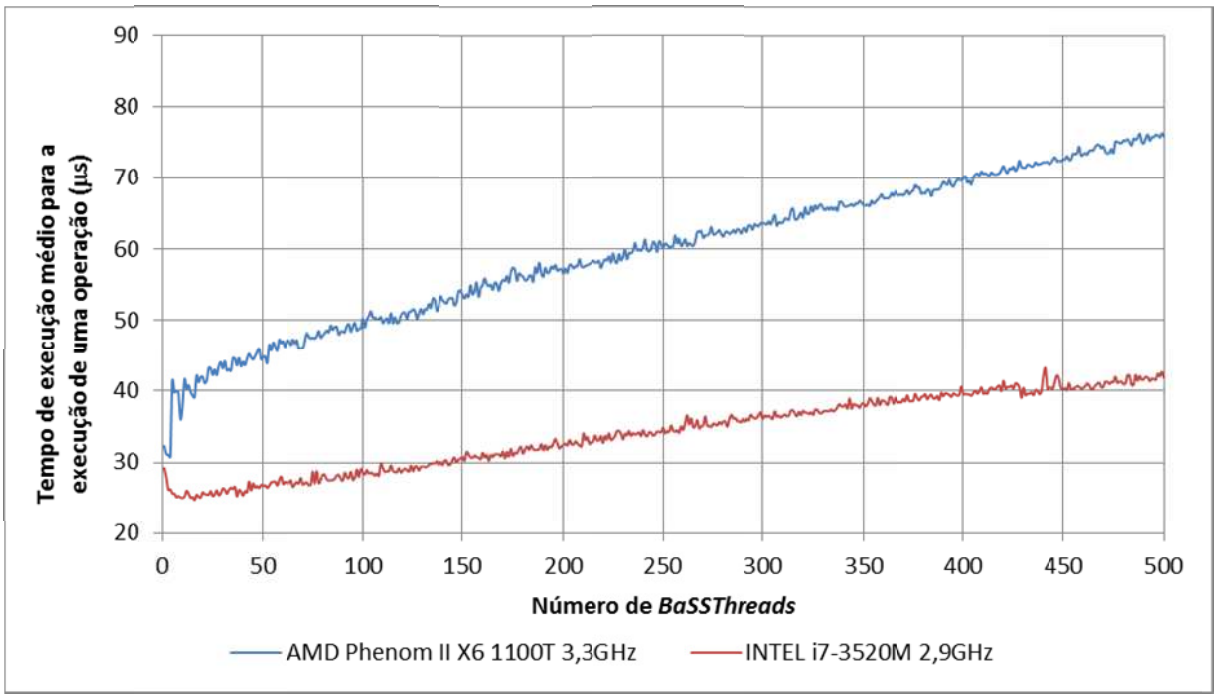


Figura 4.39 – Tempo de execução médio de uma operação de acordo com o número de *BaSSThreads* (*TimeThread* definida na Figura 4.37).

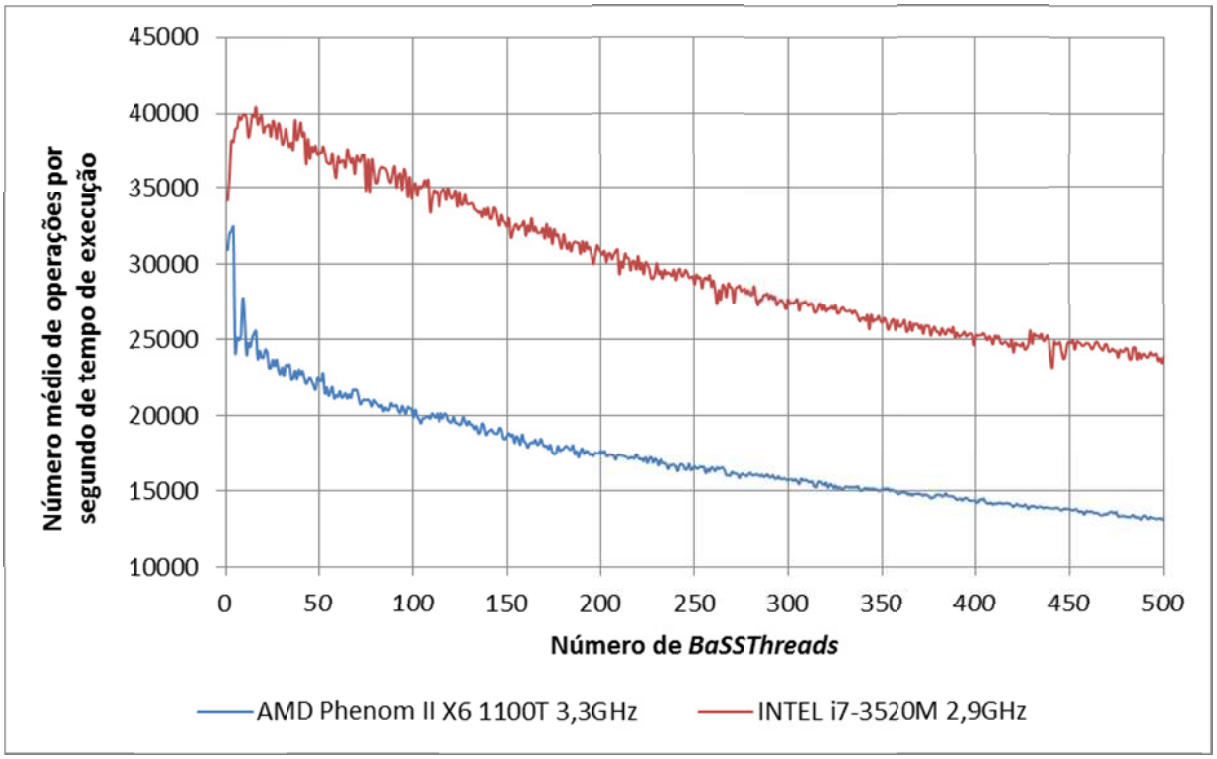


Figura 4.40 – Número médio de operações por segundo (tempo de execução) de acordo com o número de *BassThreads* (*TimeThread* definida na Figura 4.37).

[BAL03] apresenta uma comparação de desempenho entre os tipos de barreiras (vide item 4.3.2), variando o número de *threads* de 1 a 24. O resultado também pode ser comparado ao gráfico da Figura 4.39, no qual a análise do tempo de iteração (mudança no contexto de execução de uma *thread* para outra) em função do número de *threads* para o tipo Barreira Central resultou em 800µs para 24 *threads.* No caso do *framework* BaSS, o tempo necessário é de 43,20µs (AMD) e 25,99µs (Intel) para 24 *threads*. Os outros tipos de barreiras apresentaram um desempenho melhor que o da Barreira Central, com melhores valores para Barreira de Disseminação com tempo de aproximadamente 20µs para 24 *threads*. Apesar do melhor desempenho da Barreira de Disseminação, a sua utilização não permitiria a implementação dos requisitos apresentados item 4.4.

Deve-se considerar, também, que o estudo apresentado em [BAL03] foi realizado em uma máquina da Sun Microsystems, modelo Fire 6800 RISC com 24 processadores UltraSPARC III de 750MHz. Como a combinação da arquitetura UltraSPARC, sistema operacional SunOS e JVM para Sun é diferente da arquitetura AMD (x64) ou Intel (x64), sistema operacional Windows e JVM para Windows, tal comparação de resultados de desempenho em arranjos computacionais distintos não é adequada, apesar de ser uma prática frequente em estudos comparativos de performance de algoritmos e sistemas.

Apesar de nenhum artigo que se tenha tido contato da área de *frameworks* para o desenvolvimento de simuladores apresentar o cálculo da razão tempo de execução/tempo de simulação em função do número de *threads*, tal análise parece ser mais adequada do que se realizar uma comparação que envolva apenas o tempo de execução das operações. Considera-se que tal análise é mais útil do ponto de vista da usabilidade do simulador para determinada aplicação (vide item 4.2).

A Figura 4.41 apresenta o resultado da razão tempo de execução/tempo de simulação para a execução da classe *TestTime* (Figura 4.37) durante 24h de tempo de simulação em função do número de *BaSSThreads*.

Assim, para 250 *BaSSThreads*, a relação tempo de execução/tempo de simulação é de $15{,}046 \times 10^{-3}$ (AMD) e $8{,}713 \times 10^{-3}$ (Intel), isto é, o simulador executa as tarefas programadas cerca de 66 vezes (AMD) e 115 (Intel) vezes mais rápido que se forem executadas no "mundo real" (lembrando-se que cada operação de incremento da *TestThread* da Figura 4.37 é considerada como uma operação que utilizaria 1s de tempo de simulação, isto é, o tempo no "mundo real").

O teste de desempenho executado com até 500 *threads* pode também ser considerado um teste de *stress* simplificado. Considerando que a quantidade máxima de

threads em execução simultânea em um JVM é teoricamente ilimitada, não sendo determinada pela documentação da JVM e estando restrito apenas à quantidade de memória disponível no computador, o desempenho, conforme visto na tendência crescente dos gráficos apresentados, piora com a introdução de novas *threads*, mas não limita a execução da simulação a ponto de impedi-la de ser executada. Testes com menor tempo de duração que 24h de tempo de simulação foram executados com até 10.000 *threads*, sem haver travamento do programa de simulação ou instabilidade operacional da JVM.

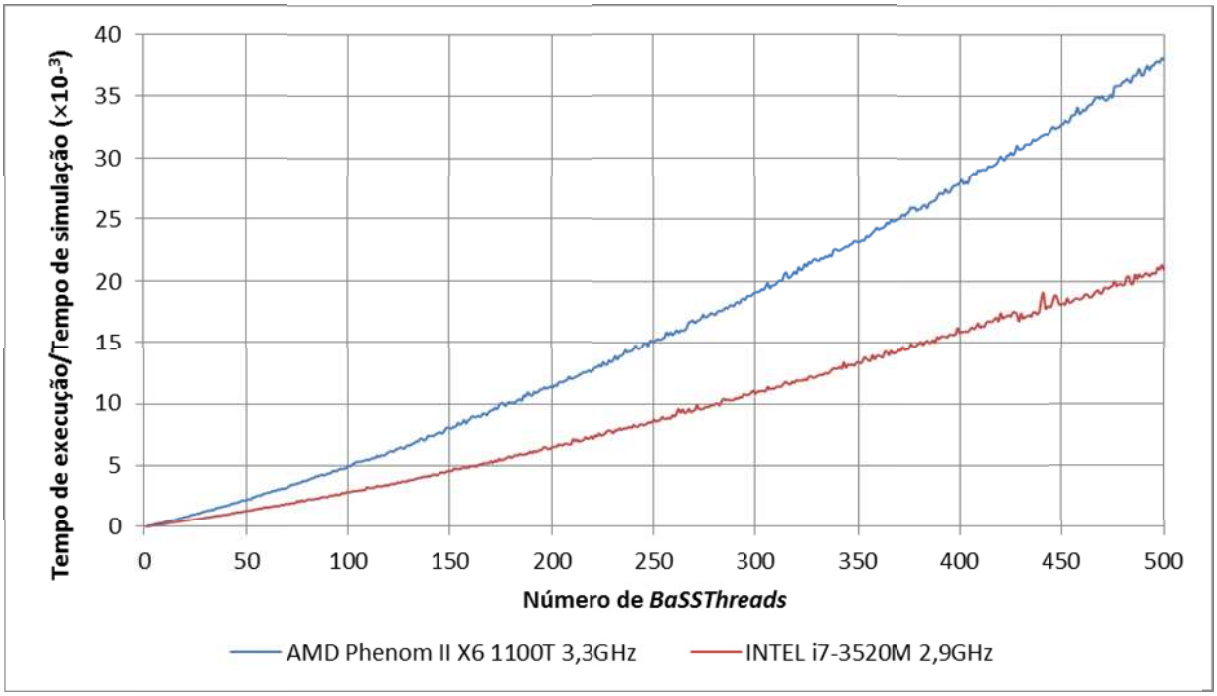


Figura 4.41 – Razão Tempo de execução / Tempo de simulação de acordo com o número de *BassThreads* (*TimeThread* definida na Figura 4.37 em execução durante 24h de tempo de simulação).

A Figura 4.42 apresenta o resultado da simulação de um conjunto de *BaSSThreads* que executam no método `run()` uma operação de incremento a cada 1ms de tempo de simulação (programação similar à da Figura 4.37, modificando apenas alterando o tempo do atributo *delay* para 00:00:001). A simulação foi programada para durar 1s de tempo de simulação, sendo que cada simulação apresentava um número de *BaSSThreads* variando de 1 a 10.000. O tempo de execução necessário para simular uma *BaSSThread* foi de 0,029s (AMD) e 0,017s (Intel) e para as 10.000 *BaSSThreads* foi de 123,028s (AMD) e 78,769s (Intel). No decorrer da simulação das 10.000 BaSSThreads, a utilização da CPU ficou em torno 35% da capacidade total e a quantidade de memória RAM utilizada foi de aproximadamente 650kB (processo java), tanto para o computador com processador AMD como para o com processador Intel.

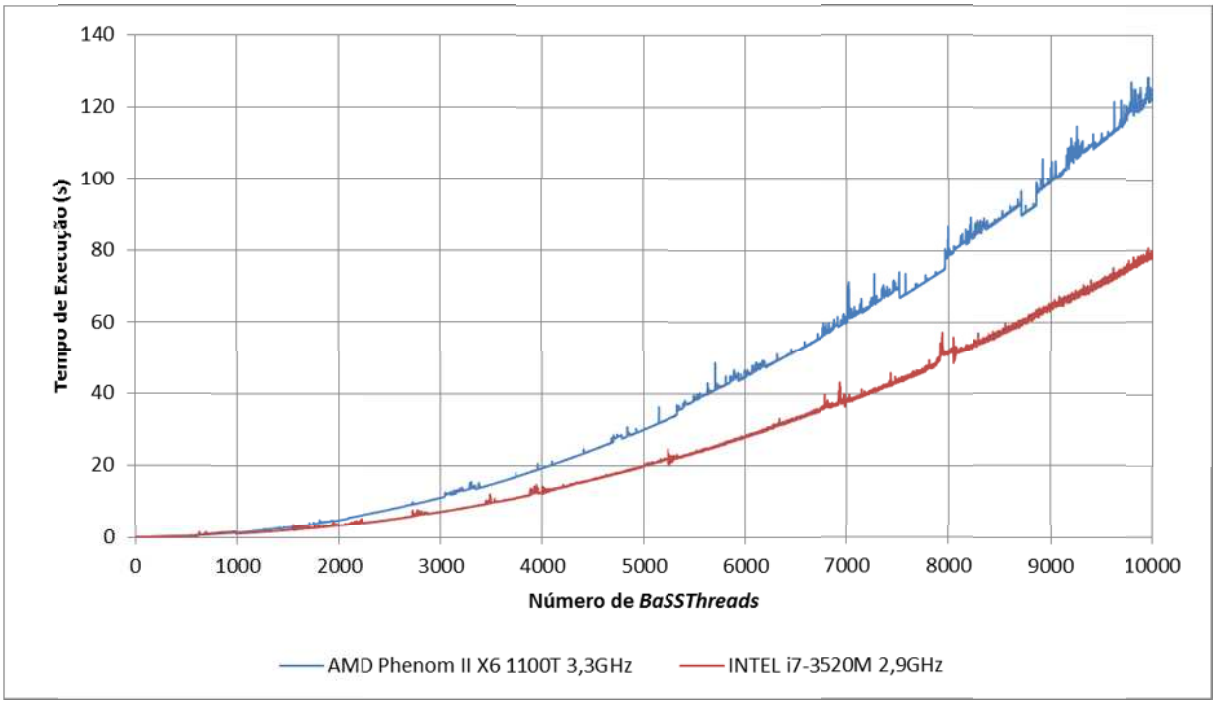


Figura 4.42 – Tempo de execução utilizado para simular 1s de tempo de simulação de acordo com o número de *BaSSThreads*. Cada *BaSSThread* executa uma operação de incremento a cada 1ms de tempo de simulação.

A validação do *framework* BaSS, assim como de qualquer tipo de software complexo, só é possível pela utilização do mesmo em diversos projetos. Em particular, um *framework* voltado para o desenvolvimento de simuladores deve ser utilizado para o desenvolvimento de uma diversidade de simuladores para se ter uma validação tanto do seu modelo teórico como da sua implementação. A corretude do BaSS, portanto, não pode ser plenamente determinada a partir de testes e de uso limitado. Certamente erros serão descobertos com a distribuição e novas adaptações do modelo serão necessárias frente à utilização por programadores de áreas diversas.

Sendo o BaSS um núcleo de propósito geral, não é tarefa difícil a sua utilização como base para qualquer tipo de simulador. Em particular, o capítulo 5 o utiliza como núcleo de simulação para um simulador de redes de sensores sem fio, principal motivação desse projeto. Contudo, outros simuladores de pequeno porte foram desenvolvidos para ampliar a quantidade de testes realizadas sobre o BaSS, entre eles um singelo simulador de circuitos lógicos, apresentado em [SAL09a].

4.6.4 – Divulgação e documentação do *framework* BaSS

A divulgação do *framework* BaSS é feita através da Internet, estando disponível no seguinte repositório: http://www.lpm.fee.unicamp.br/~rsalusti/bass. A língua inglesa foi utilizada nas páginas de divulgação do *framework* por ser a língua adotada mundialmente na difusão do conhecimento científico, ampliando, assim, a possibilidade de utilização do *framework*.

As páginas estão organizadas em quadros (*frames*), sendo acessadas por um *menu* localizado à esquerda, o qual contém os seguintes itens: Introdução, Exemplos, Documentação e *Download*.

A página de Introdução apresenta uma breve descrição do Projeto BaSS (vide Figura 4.43). Nas páginas de Exemplos são apresentadas de maneira explicativa e com exemplos de programas a utilização das classes *BaSSControl*, *BaSSThread* (vide Figura 4.44), *BaSSInterruption*, *BaSSTimer*, *BaSSLogger* e *BaSSUpdater*. As páginas de Exemplos podem ser utilizadas como um tutorial para o aprendizado do *framework*.

A página de Documentação (Javadoc) reúne de forma sistemática e padronizada os métodos e atributos das catorze classes do *framework* BaSS (pacote *br.eng.rsalustiano.bass*): *BaSSControl* (Figura 4.45), *BaSSControl.State,* *BaSSDelay*, BaSSDelay.Type, *BaSSInterruption*, *BaSSInterruptionEvent*, *BaSSLogger*, *BaSSRandom*, *BaSSStandardTime*, *BaSSThread*, *BaSSTime*, *BaSSTimer*, *BassUpdater* e *BaSSUpdater.Type*. Exemplos de utilização das classes também são apresentados no corpo do Javadoc, complementando as explicações encontradas nos itens do *menu* Exemplos.

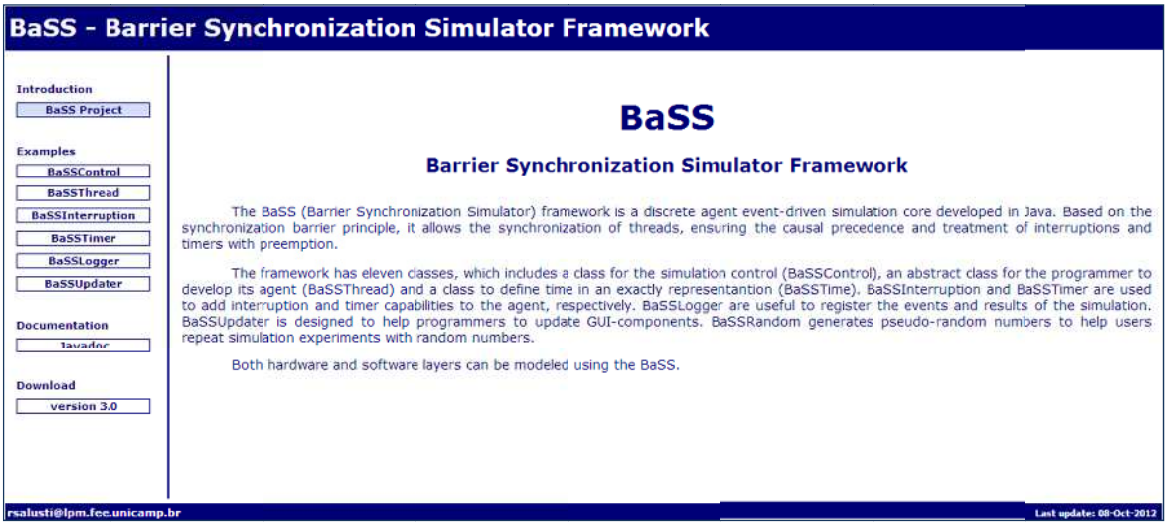


Figura 4.43 – Aspecto visual da página de divulgação do BaSS. O texto exibido refere-se ao item Bass Project (Projeto BaSS) que apresenta uma introdução ao *framework*.

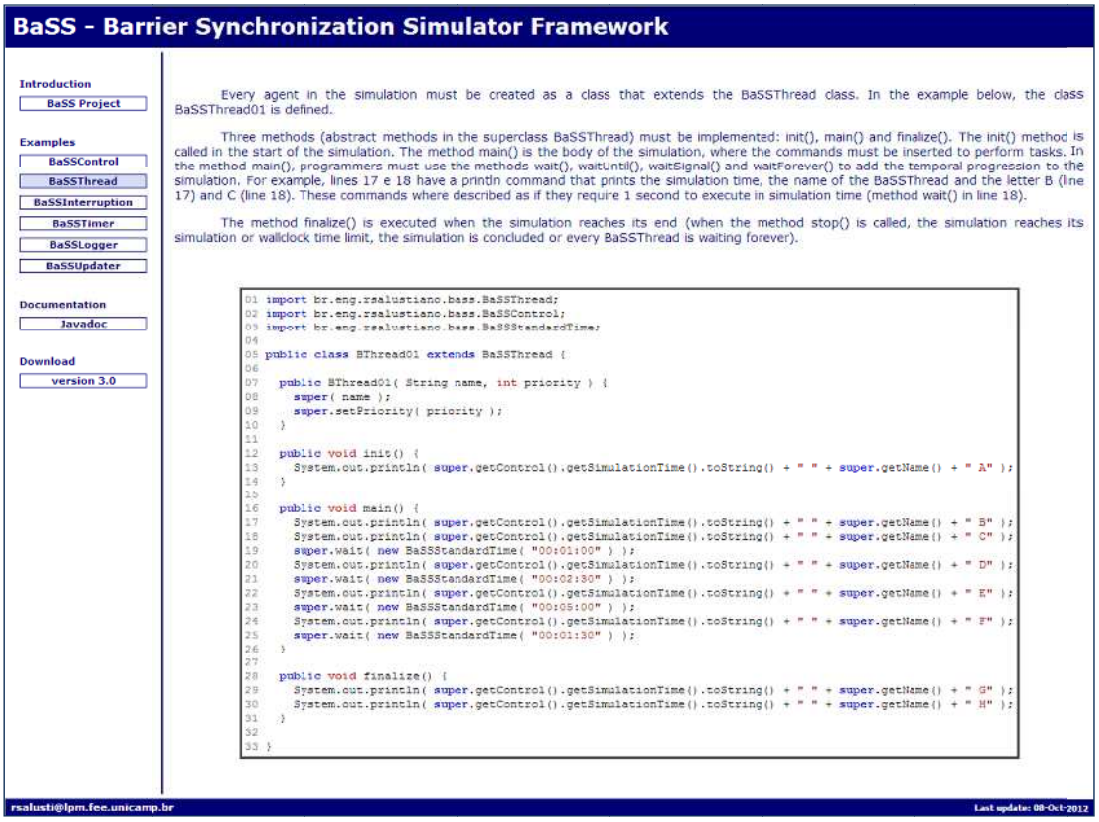


Figura 4.44 – Texto e programa explicativos da utilização da classe *BaSSThread*.

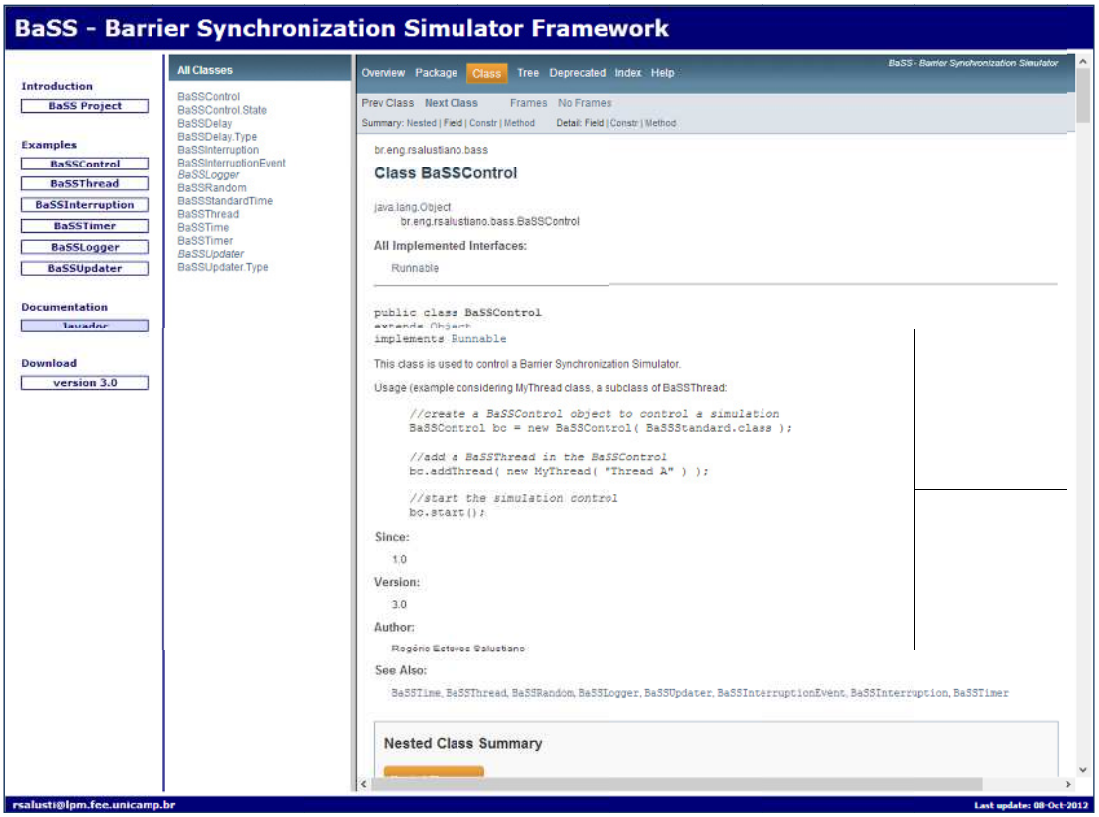


Figura 4.45 – Javadoc com as classes do *framework* BaSS. Texto selecionado referente à classe BaSSControl.

O último item do *menu*, *Download*, permite que o usuário faço o *download* do arquivo BaSS.jar, que contém o pacote do *framework* BaSS (br.eng.rsalustiano.bass). Esta forma de distribuição dos arquivos não só facilita a portabilidade (inerente à arquitetura Java, na qual o pacote pode ser utilizado em diferentes plataformas na qual a JVM possui uma implementação), mas o formato JAR (*Java ARchive*) garante a preservação da estrutura do pacote de classes e facilita a reprodução (cópia) do seu conteúdo.

Resumo

Este capítulo descreveu o BaSS, um framework desenvolvido na linguagem de programação Java que implementa um núcleo de simulação conservativo do tipo event-driven baseado em agentes. Tendo como referência no modelo teórico de sincronização por barreiras, o BaSS possui os seguintes diferenciais em relação a outros frameworks ou bibliotecas de programação destinadas ao desenvolvimento de simuladores:

- pequena quantidade de classes (catorze classes, sendo que o programador precisa conhecer a estrutura de apenas três para poder executar uma simulação simples);
- usa o paradigma de múltiplas barreiras, que acelera o tempo de simulação pela redução no número de bloqueios realizados nas threads.
- permite que elementos da simulação atendam à interrupções de modo preemptivo e possuam temporizadores internos;
- permite a atualização de interfaces gráficas durante a simulação (uso dos atualizadores – *BaSSUpdater*), não havendo a necessidade de finalizar a simulação para visualizar o resultado.

O desempenho do BaSS mostrou-se satisfatório diante os resultados apresentados por outros *frameworks* para o desenvolvimento de simuladores, embora tal comparação não possa ser realizada de forma completa devido a falta de informações na literatura sobre o desempenho dos demais *frameworks* utilizando procedimentos semelhantes de análise quantitativa.

A distribuição do BaSS é gratuita (pacote br.eng.rsalustino.bass) e realizada através de um repositório da Internet apresentado na língua inglesa. No mesmo repositório pode-se aprender a utilizar o BaSS valendo-se de um singelo tutorial.

5

WSN-BaSS
Um simulador para Rede de Sensores Sem Fio

"A imaginação é mais importante que a ciência,
porque a ciência é limitada, ao passo que a
imaginação abrange o mundo inteiro".
Albert Einstein

Conforme explorado no capítulo 3, a comunidade científica e as empresas de desenvolvimento de software dispõem de uma série de simuladores para Redes de Sensores Sem Fio. Apesar disso, diante a variedade de definições, requisitos e possibilidades de aplicação das WSN apresentada no capítulo 2, uma ferramenta de simulação para WSN suficientemente abrangente e flexível seria uma contribuição relevante para esta área de estudo.

Este capítulo apresenta a arquitetura do WSN-BaSS (*Wireless Sensor Network - Barrier Sinchronization Simulator*), um simulador desenvolvido para Redes de Sensores Sem Fio.

Baseado no núcleo de simulação proposto pelo *framework* BaSS no capítulo 4, o WSN-BaSS, também desenvolvido na linguagem de programação Java, foi modelado para permitir a descrição de elementos pertinentes às Redes de Sensores sem Fio (nós sensores, nós estação radio base, comunicação, etc.) e elementos relacionados ao ambiente de inserção dos nós da rede (localização dos nós, interferências na comunicação, obstáculos, etc.).

O processo de criação de uma simulação utilizando o WSN-BaSS pode ser dividido

em duas partes. A primeira é composta pela descrição do funcionamento dos nós e parâmetros de comunicação, sendo necessário para isso a programação na linguagem Java e o conhecimento da arquitetura do BaSS. A segunda consta da descrição da rede e o procedimento de simulação propriamente dito, que pode ser realizado tanto utilizando programação como através de uma interface gráfica desenvolvida para a execução das simulações de forma mais dinâmica.

A descrição da rede, isto é, o número de nós e suas posições geográficas, além das variáveis ambientais (temperatura, humidade, etc.) e barreiras atenuadoras de sinal, pode ser feita por meio de um arquivo XML (*eXtensible Markup Language*) e lida tanto através de programação como pelo ambiente gráfico.

5.1 – Características do simulador desenvolvido

O WSN-BaSS foi concebido tendo como base o BaSS, núcleo de simulação cujo funcionamento, desenvolvimento e estrutura foram descritos no capítulo 4. Trata-se de uma estrutura de classes que é ao mesmo tempo rígida, no sentido de que as principais funcionalidades de simulação (sincronização), transmissão de dados e registro de ocorrências de eventos já estarem previamente estabelecidas, e suficientemente flexível para que novas funcionalidades possam ser inseridas facilmente.

A forma com que a estrutura das classes de simulação foi desenvolvida permite que, uma vez entendida sua organização e funcionalidades (representação dos nós, mecanismos de trocas de mensagens e registros dos dados de simulação), o nível de detalhamento possa ser determinado pelo desenvolvedor. A quantidade de camadas de redes necessárias, o tamanho e os campos das mensagens trocadas pelos nós e a descrição pormenorizada dos elementos de hardware e software podem ser implementados tendo como base as classes WSN-BaSS. Quanto mais detalhadas as descrições, melhores serão os resultados das simulações, contundo há a penalização de uma quantidade maior de tempo necessário para a execução das mesmas.

Além dessa flexibilidade no detalhamento descritivo e funcional dos nós, outra característica que o WSN-BaSS possui na sua arquitetura organizacional é isolar os conhecimentos da rede. Um dos principais problemas que se pode levantar nos artigos publicados em Redes de Sensores Sem Fio, principalmente naqueles que realizam descrições e testes de protocolos de comunicação, é que em grande parte dos casos não é realizada uma diferenciação do conhecimento da rede e do conhecimento do simulador. Principalmente nos quesitos de informação de posição dos nós, conectividade da rede e quantidade de energia

disponível por nó, muitas simulações partem do pressuposto que cada nó sabe a sua localização, quais são seus vizinhos, conhece as trajetórias para a entrega das mensagens até as estações rádio base e sabe as condições de energia de todos os nós da rede. Tais informações não estão disponíveis diretamente para os nós da rede, sendo um erro considerá-los de total ou parcial conhecimento dos nós sensores no estudo de desempenho das Redes de Sensores Sem Fio.

Não há impedimento para que os nós tenham todo o conhecimento da rede necessário para executar determinada ação, mas este conhecimento tem que ser enviado até o nó através de mensagens, o que envolve uma série de transmissões, gasto de tempo e de energia dos nós envolvidos nessas transmissões.

Contudo, se por um lado o conhecimento da rede é algo que tem o seu custo e nem sempre é viável ou possível de ser determinado para os nós da rede, o conhecimento do simulador é de extrema importância para a análise do funcionamento, do desempenho da rede e da eficiência dos protocolos de comunicação. É, na verdade, o próprio sentido maior da existência do simulador, isto é, saber o que se passa na rede como um todo, como se um observador externo soubesse a cada instante o que está acontecendo na rede. Em grande parte das aplicações, por exemplo, os nós são lançados no meio sem saber onde estão. Inicialmente, portanto, a informação da localidade de cada nó só deve ser sabida pelo simulador.

A abstração da rede como um grafo é outra informação relevante na análise da estrutura da rede e sua conectividade. É uma característica importante para um simulador de redes, uma vez que toda rede pode ser vista como um grafo, seja ele representando a conectividade entre os nós (arestas), seja ele representando a disponibilidade energética (nós). Grafos de redes também são informações exclusivas do simulador, não pertencendo ao domínio de informações acessível diretamente pelos nós da rede.

Há, portanto, dois “mundos” dentro da própria simulação. O “mundo” das informações pertinentes ao simulador (conhecimento global de tudo) e o “mundo” das informações pertencentes a cada nó (restrito e real). Neste último “mundo”, cada nó só sabe de si mesmo enquanto dispositivo eletrônico; qualquer outra informação que ele possua foi resultado da sua prévia programação ou da troca de mensagens efetuada pela rede de comunicação. Os conhecimentos externos podem ser recebidos pelos nós da rede utilizando a própria rede de comunicação sem fio pertencente à rede ou a partir de outras redes (determinação da posição do nó utilizando a rede GPS, por exemplo). Seja qual for a forma com que a informação for obtida, há um custo de energia associado, o que deve ser contabilizado no tempo de vida nó e da rede como um todo.

Toda a arquitetura do simulador para Redes de Sensores Sem Fio, isto é, a sua estrutura de classes, foi desenvolvida baseado nessa distinção entre os “mundos” de conhecimento (apesar de o programador poder violar essa separação se assim desejar). Além disso, a contabilidade do consumo energético dos nós sensores sempre esteve presente no decorrer do projeto do simulador, uma vez que este tipo de análise é o principal objetivo na realização de simulações para o estudo das Redes de Sensores Sem Fio.

Segue uma relação das principais características do simulador desenvolvido. Na forma com que foi concebido e implementado, o WSN-BaSS difere de todos os simuladores de Redes de Sensores Sem Fio estudados no capítulo 3, apresentando um conjunto de propriedades que auxiliam no desenvolvimento de simulações e na avaliação de seus desempenhos.

(1) Controle automático da sincronização dos objetos simulados;

(2) Representação de diferentes tipos de nós, especificando seus comportamentos tanto no que se refere ao hardware como ao software e ainda permitindo um maior ou menor detalhamento descritivo de acordo com as necessidades do projeto. Diferenciação dos nós como sendo nós sensores ou estações rádio base;

(3) Descrição do funcionamento de diferentes tipos de rádios que operam com frequência, modulação e taxas de transmissão distintas e associá-los aos nós da rede;

(4) Definição de modelos de propagação para as ondas de rádios, fixando para a simulação as regras que determinam quando o rádio de um nó “está no alcance” do rádio de outro nó, isto é, quando há a transmissão entre um par de nós da rede;

(5) Descrição do comportamento de sensores com funcionamentos e naturezas sensoriais distintas;

(6) Descrição do comportamento de diferentes tipos de baterias;

(7) Utilização de nós da rede com baterias e sensores diferentes;

(8) Descrição do comportamento de diferentes variáveis ambientais com valores dinâmicos no tempo e espaço, permitindo que os seus valores possam ser registrados pelos sensores dos nós de acordo com a sua posição geográfica, tipo de sensor e momento da leitura;

(9) Descrição do comportamento de sinais de interferência com valores dinâmicos no tempo e espaço de forma a corromper as mensagens transmitidas entre os

nós;

(10) Inserção de obstáculos no ambiente, cada qual com características de atenuação de sinal particulares;

(11) Movimentação dos nós pelo ambiente;

(12) Registro do gasto energético dos nós, retirando-o da rede automaticamente na sua falência energética;

(13) Registro para cada nó da rede a quantidade de mensagens enviadas, recebidas, corrompidas por interferência ou que sofreram colisão com outras transmissões;

(14) Análise visual da rede como um grafo, cujos valores das arestas e nós podem ser configurados de acordo com as necessidades dos projetos, além de determinar o melhor caminho entre dois nós obedecendo a critérios estabelecidos pelo programador;

(15) Execução de simulações utilizando diferentes topologias de rede (pré-determinadas ou aleatórias);

(16) Utilização de documentos XML para salvar as simulações, permitindo, assim, a repetitividade na execução das simulações e a realização de pequenas alterações para verificar mudanças nos resultados experimentais;

(17) Execução das simulações sem nenhum tipo de visualização (execução mais rápida) ou executando-a em um ambiente computacional gráfico no qual as ações executadas na rede podem ser observadas e analisadas durante o processo de execução da simulação (execução visual e possivelmente interativa);

(18) Criação, utilização e o processamento de diferentes tipos de registros (*logs*), tanto na forma de arquivos de texto como de elementos visuais na interface gráfica;

(19) Escalabilidade, isto é, utilização de uma quantidade considerável (milhares) de nós do mesmo tipo ou de tipos diferentes na simulação;

Os itens a seguir descrevem a forma com que essas dezenove características foram implementadas no WSN-BaSS. Primeiramente um panorama geral da estrutura de classes será abordado, seguido de um detalhamento de como implementar as funcionalidades dos nós, criação das redes e execução das simulações. Finalmente será abordada a interface gráfica do WSN-BaSS.

5.2 – Estrutura de classes do WSN-BaSS

O WSN-BaSS (*Wireless Sensor Network Barrier Synchronization Simulator*) foi desenvolvido na linguagem de programação Java e suas principais classes estão agrupadas no pacote `br.eng.rsalustiano.wsn`. As dezessete classes públicas deste pacote compõem toda a estrutura necessária para o desenvolvimento de uma simulação para Redes de Sensores Sem Fio, apresentando dependência com as classes do pacote `br.eng.rsalustiano.bass`, responsáveis pela coordenação da simulação.

O pacote `br.eng.rsalustiano.wsn` possui três subpacotes. O pacote `br.eng.rsalustiano.wsn.graph` contém quatro classes que implementam a estrutura de um grafo utilizado para análise da relação entre os nós de uma Rede de Sensores Sem Fio. O pacote `br.eng.rsalustiano.wsn.parser` possui apenas uma classe, responsável pela representação de uma simulação no padrão XML [DEI00]. O pacote `br.eng.rsalustiano.wsn.gui` possui as classes que produzem a interface gráfica, permitindo a visualização da rede, a criação de novas redes de forma gráfica, execução da simulação com acompanhamento visual da rede e dos relatórios (*logs*) durante a execução da simulação. Os subpacotes do pacote `br.eng.rsalustiano.wsn` são, portanto, especializações que adicionam funcionalidades às simulações.

A Figura 5.1 apresenta o diagrama de dependência de pacotes do WSN-BaSS na notação UML (*Unified Modeling Language* [MUL97]).

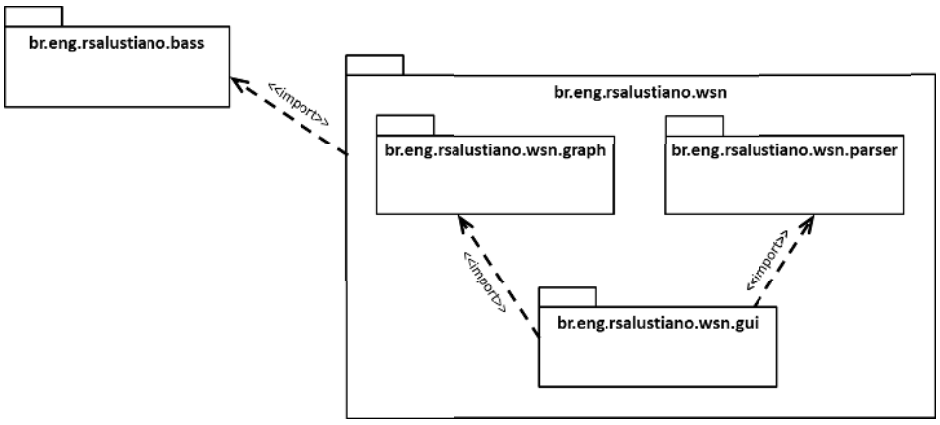


Figura 5.1 – Digrama de dependência de pacotes do WSN-BaSS.

A Tabela 5.1 apresenta as classes do pacote `br.eng.rsalustiano.wsn` com uma breve descrição das suas funcionalidades, enquanto que a Figura 5.2 mostra o diagrama de classes deste pacote (notação UML) e as classes dos subpacotes

`br.eng.rsalustiano.wsn.graph` e `br.eng.rsalustiano.wsn.parser`, além das classes do pacote `br.eng.rsalustiano.bass` cujas classes do pacote `br.eng.rsalustiano.wsn` apresentam dependência direta. Ressalta-se que algumas classes utilizam classes privadas definidas internamente, sujas funcionalidades restringem-se à organização estrutural específica dessas classes, não sendo apresentadas na Tabela 5.1 e Figura 5.2. Apenas a classe *WSNSimulator.SimulatorNetworkNode* merece uma atenção especial devido ao seu significado no contexto geral da estrutura da simulação, mesmo que sua utilização seja restrita à classe *WSNSimulator*.

Tabela 5.1 – Classes do pacote br.eng.rsalustiano.wsn

Classe	Descrição
WSNAntenna	Interface que contém a declaração dos métodos necessários para a definição de um objeto antena, necessário à comunicação entre os nós da rede. O ganho da antena (em dBi), a direção e a largura do feixe (*beam width*) são os parâmetros a serem definidos para cada tipo de antena. Um objeto do tipo *WSNRadio* deve possuir um *WSNAntenna* para que possa haver comunicação entre os rádios dos nós da rede.
WSNAttributes	Classe que permite o armazenamento de um conjunto de pares (atributo, valor) utilizados para configurar qualquer tipo de objeto configurável na simulação. Os pares são armazenados em uma tabela de dispersão (*hashtable*). Ver descrição da classe *WSNConfigurable*.
WSNBaseStation	Subclasse de *WSNNetworkNode*. Trata-se de uma classe abstrata que define uma estação rádio base. Este tipo de nó da rede não possui bateria (sua fonte de energia pode ser considerada infinita) e não possui sensores.
WSNBattery	Interface que define uma bateria a ser utilizada como fonte de energia para um nó sensor. Um objeto que implementa *WSNBattery* deve fornecer os seguintes parâmetros: a quantidade de carga restante na bateria em Coulombs, a quantidade de carga restante na bateria em percentual e a indicação se ainda há carga disponível na bateria.
WSNConfigurable	Interface que permite com que um objeto seja configurável na simulação, isto é, se uma determinada classe implementa *WSNConfigurable*, os atributos e valores mais importantes do objeto podem ser configurados quando o simulador chama o método *setAttribute(atributo, valor)* e podem ser lidos pelo simulador através da implementação do método *getAttributes()*, que retorna um objeto do tipo *WSNAttibutes*.
WSNLogger	Subclasse de *BaSSLogger*. Não apresenta nenhum tipo de modificação estrutural em relação à superclasse. O objetivo da existência dessa classe é não haver a necessidade do programar importar a superclasse pertencente a outro pacote no desenvolvimento do simulador.

WSNNetworkNode	Classe abstrata que define um nó da rede. Esta classe implementa de maneira abstrata uma *WSNThread*. Para se desenvolver um nó da rede (estação rádio base ou nó sensor) deve-se implementar uma classe a partir das classes abstratas *WSNBaseStation* ou *WSNSensorNode*.
WSNObstacle	Classe que define um obstáculo no meio de transmissão, isto é, uma barreira que reduz o alcance ou impede a passagem das ondas eletromagnéticas. Geometricamente, um obstáculo é definido como uma reta, sendo criada através de dois pontos em um espaço bidimensional. O valor da atenuação de sinal deve ser fornecido em dBm.
WSNPosition	Classe utilizada para estabelecer a posição de um nó da rede no ambiente (espaço bidimensional). Um nó da rede (implementação de *WSNBaseStation* ou *WSNSensorNode*) é inserido na simulação (*WSNSimulator*) em uma determinada posição geográfica, mas esta posição não é conhecida pelo nó, não podendo ser acessada diretamente pelo objeto que o representa na simulação.
WSNPropagationModel	Interface que define o modelo de propagação das ondas eletromagnéticas. O usuário deve implementar o método *getRxPowerInMilliWatt()* que recebe como parâmetro dois rádios (objetos implementados a partir de *WSNRadio*), um transmissor e outro receptor, e a distância entre eles, devendo retornar a potência da onda eletromagnética recebida pelo rádio receptor.
WSNRadio	Interface utilizada para determinar os parâmetros de comunicação entre dois nós da rede. A implementação dessa interface deverá definir se um rádio é compatível com outro (verificação se ambos estão operando na mesma frequência, canal, etc.), fornecer a antena associada ao rádio, a frequência de operação do rádio em Hz, a potência de transmissão em dBm, a sensibilidade do rádio em dBm, fazer a leitura do seu *buffer* de armazenamento de mensagens, colocar e retirar uma mensagem do *buffer*, indicar se o rádio está recebendo ou transmitindo mensagens e chamar um método de interrupção quando uma mensagem é recebida. Além disso, caso ocorra qualquer problema na transmissão de uma mensagem, uma implementação da classe *WSNRadio* deve ser capaz de criar uma mensagem corrompida, isto é, simulando um erro no seu conteúdo. Todo objeto implementado a partir das classes abstratas *WSNBaseStation* e *WSNSensor* Node devem possuir um objeto implementado de *WSNRadio* associado.
WSNSensor	Interface que define um sensor. O método *getValue()* deve ser implementado, passando como parâmetros o simulador, o nó sensor e o tipo de variável que se deseja obter o valor (uma vez que existem sensores com vários transdutores internos). O retorno deste método é um número do tipo *double*, que pode ser definido na implementação do *WSNSensor*, ou, de maneira mais "realística", pode ser feita uma consulta do valor da variável no ambiente na posição do nó sensor. Neste último caso, o programador deve utilizar o método *getVariableValueByClass()* da classe WSNSimulator.

WSNSensorNode	Subclasse de *WSNNetworkNode*. Trata-se de uma classe abstrata que define um nó sensor. Este tipo de nó da rede possui uma bateria e um conjunto de sensores associados. Além disso, uma implementação de *WSNSensorNode* deve fornecer a quantidade de energia que está sendo utilizada no seu estado atual em mA, o que envolve tanto o seu consumo, como o dos seus sensores e de outros dispositivos de hardware associados ao nó sensor.
WSNSimulator	Classe utilizada para criar e controlar uma simulação. Ela é definida como uma subclasse de *BaSSControl* e implementa *BaSSUpdater*. Sendo uma subclasse de *BaSSControl*, ela possui todos os mecanismos de controle de simulação pertencentes ao BaSS. Como uma *BaSSUpdater* (do tipo SIMULATION_TIME_UPDATER), em toda mudança no tempo de simulação é verificado se os nós sensores ainda possuem energia nas suas baterias (caso contrário, são retirados da simulação) e se a simulação chegou ao fim (todos os nós sensores ficaram sem energia). Esta classe também é responsável pela adição e remoção de nós de rede (*WSNBaseStation* e *WSNSensorNode*), obstáculos (*WSNObstacle*), variáveis de ambiente (*WSNVariable*). Além disso, ela controla todo o procedimento de envio e recebimento de mensagens e possui a informação da possibilidade de troca de mensagens entre os nós. Esta classe possui uma visão geral da simulação, possuindo o "conhecimento global da rede".
WSNSimulator. SimulatorNetworkNode	Principal classe privada da classe *WSNSimulator*. Ela estabelece a relação entre o nó da rede e sua posição no ambiente. Na arquitetura do WSN-BaSS, objetos da classe *WSNNetworkNode* (implementados através da *WSNNetworkNode* ou *WSNBaseStation*) não tem conhecimento da suas posições no ambiente. Essa associação (nó da rede, posição) é realizada, e pode ser consultada, somente dentro da classe *WSNSimulator*.
WSNThread	Classe abstrata e subclasse de *BaSSThread*. Além dos métodos da *BaSSThread*, o método *getSimulator()* foi adicionado para permitir que os sub-objetos implementados a partir desta classe tenham acesso de forma direta ao objeto simulador (*WSNSimulator*).
WSNUpdater	Subclasse de *BaSSUpdater*. Não apresenta nenhum tipo de modificação estrutural em relação à superclasse. O objetivo da existência dessa classe é não haver a necessidade do programar importar a superclasse pertencente a outro pacote no desenvolvimento do simulador.
WSNVariable	Interface utilizada para definir uma variável do ambiente (temperatura, humidade, pressão, interferência eletromagnética, etc.). A implementação dessa classe deve fornecer o nome, unidade e valores máximo e mínimo para a variável. Além disso, deve-se indicar se essa variável é do tipo interferência, ou seja, é uma variável que terá uma influência na comunicação de dados pela rede sem fio. Finalmente o método *getValue()* deve ser definido, tendo como parâmetro de entrada uma posição (*WSNPosition*) e retornando o valor da variável (valor numérico tipo *double*) naquela posição.

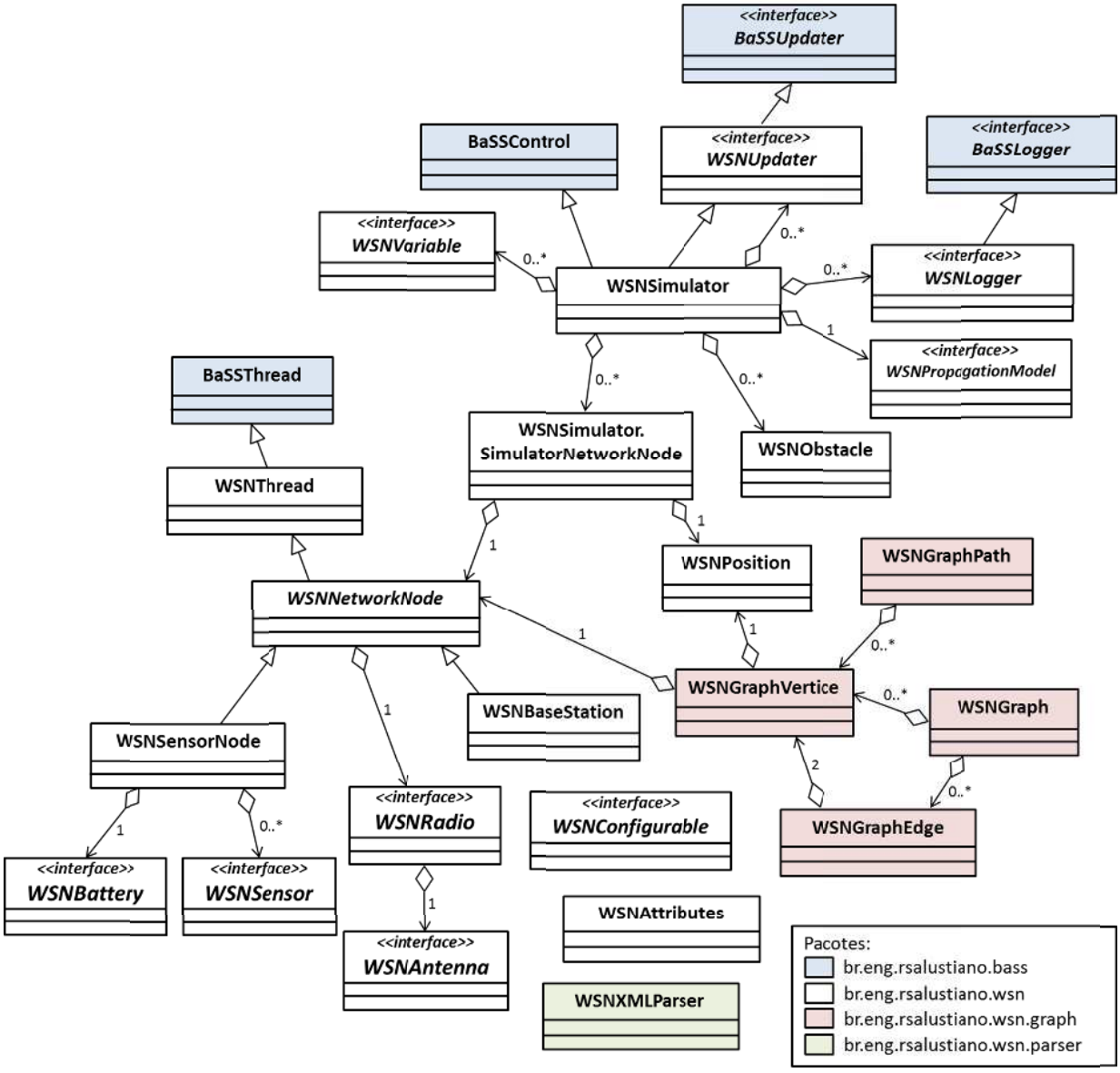


Figura 5.2 – Digrama de classes dos pacotes *br.eng.rsalustiano.wsn*, *br.eng.rsalustiano.wsn.graph* e *br.eng.rsalustiano.wsn.parser*. Neste diagrama também são exibidas as classes do pacote *br.eng.rsalustiano.bass* cujas classes do pacote *br.eng.rsalustiano.wsn* apresentam dependência direta.

A maioria das classes do pacote `br.eng.rsalustiano.wsn` que deverão ser implementadas para a simulação de uma Rede de Sensores Sem Fio são interfaces. Uma estratégia diferente, do ponto de vista conceitual, seria defini-las como classes abstratas, possuindo métodos abstratos a serem implementados nas classes específicas do projeto. A decisão de declará-las com interfaces é justificada pelo fato da arquitetura Java não permitir herança múltipla.

Supondo que um programador deseje desenvolver uma classe que represente um objeto do tipo sensor que, além de atender às definições de um sensor, execute algumas ações de forma sincronizada com outros objetos da simulação. Dessa forma, a classe desenvolvida

deveria ser tanto uma classe do tipo *WSNSensor* como uma classe do tipo *WSNThread*, o que seria impossível na arquitetura Java se ambas fossem definidas como classes "puras" ou classes abstratas. Portanto, para permitir esse tipo de implementação, a maioria das classes modeladas no WSN-BaSS são interfaces. A exceção está na classe *WSNNetworkNode* e suas subclasses *WSNBaseStation* e *WSNSensorNode*, que foram definidas como abstratas e não como interfaces por já serem subclasses de *WSNThread*.

Por outro lado, pelo fato da linguagem Java permitir a definição de uma classe implementando diversas interfaces, é possível que uma classe seja ao mesmo tempo um nó sensor (por herança da superclasse *WSNSensorNode*), uma bateria (implementação da interface *WSNBattery*), uma antena (implementação da interface *WSNAntenna*), um rádio (implementação da interface *WSNRadio*), etc.

Contudo, esta abordagem de definição conjunta de interfaces em uma mesma classe deve ser evitada, tanto do ponto vista teórico como do ponto de vista prático. Teoricamente é um erro conceitual unir funcionalidades que não possuem relação direta entre si na definição de uma mesma classe. Na prática, o reuso das classes desenvolvidas em diferentes projetos de Redes de Sensores Sem Fio ficaria limitado, uma vez que a utilização das combinações de rádios, sensores e tipos de nós sensores ficaria restrita às implementações conjuntas das interfaces já programadas.

As classes pertencentes aos subpacotes `br.eng.rsalustiano.wsn.graph`, `br.eng.rsalustiano.wsn.parser` e `br.eng.rsalustiano.wsn.gui` serão abordados na medida que suas funcionalidades forem apresentadas nos próximos itens.

5.3 – Arquitetura funcional do WSN-BaSS

O diagrama de classes da Figura 5.2 permite uma visualização geral da estrutura e da relação entre as classes que compõem o WSN-BaSS. Contudo, o funcionamento adequado de uma simulação necessita que certas regras de projeto sejam obedecidas e certas sequências de comandos sejam executadas para se atingir o resultado desejado.

Os subitens a seguir apresentam de maneira mais detalhada a estrutura e o funcionamento dos principais elementos que compõem uma simulação de Redes de Sensores Sem Fio utilizando a arquitetura do WSN-BaSS.

Ao longo do texto são apresentados exemplos de implementações de classes e métodos na linguagem Java que não estão completas e extremamente detalhadas, pois o objetivo principal é apresentar a arquitetura do WSN-BaSS, não os detalhes da implementação de componentes específicos. Tal abordagem mais detalhada tornaria o código dos programas

extremamente grandes e aumentaria a complexidade para o seu entendimento, por isso foi evitada.

5.3.1 – Estrutura e funcionamento de um nó da rede

Um nó da rede deve ser implementado em uma subclasse de *WSNSensorNode* se ele for um nó sensor ou em uma subclasse de *WSNBaseStation* se ele for uma estação rádio base. Tanto a *WSNSensorNode* como a *WSNBaseStation* são subclasses de *WSNNetworkNode* e, por isso, devem possuir, necessariamente, um objeto do tipo *WSNRadio*. Cada *WSNRadio* necessariamente deve possuir um objeto do tipo *WSNAntenna*.

A diferença entre um *WSNBaseStation* e um *WSNSensorNode*, é que este deve possui um objeto do tipo *WSNBattery*, representando a bateria do nó e pode possuir um conjunto de objetos do tipo *WSNSensor*, representado os sensores do nó sensor. A Figura 5.3 ilustra a composição de objetos do tipo estação radio base e nó sensor.

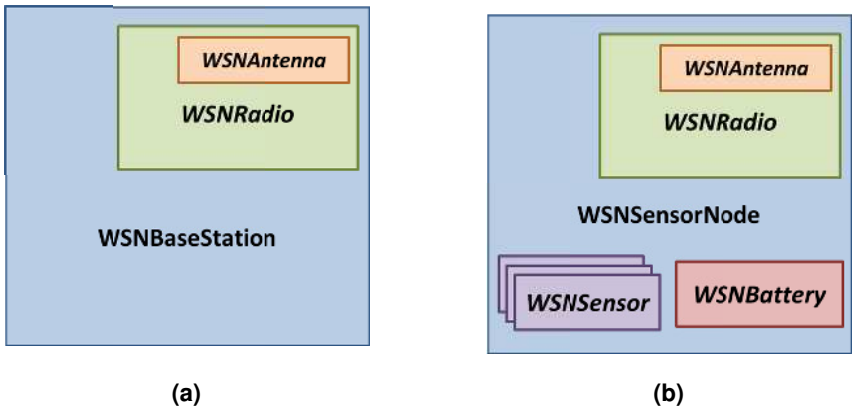


Figura 5.3 – Composição dos nós da rede. (a) Estação rádio base (WSNBaseStation); (b) Nó sensor (WSNSensorNode).

Um objeto de uma subclasse de *WSNNetworkNode* também é um objeto *WSNThread*, que por sua vez também é um objeto *BaSSThread*. Este objeto, portanto, é sincronizável com os outros que também tenham *BaSSThread* como superclasse.

Cada objeto que representa um nó da rede deve ser interpretado como se fosse um microcontrolador, isto é, um dispositivo eletrônico que executa tarefas de acordo com a sua programação. Os rádios e os sensores associados aos nós também podem ser dispositivos que apresentem processamento independente e, para entrarem em sincronização com toda a simulação, devem ser subclasses de *WSNThread*, além de serem retornados no método `getThreads()` a ser implementado pelas subclasses de *WSNNetworkNode*. As baterias, por sua vez, também podem ser um objeto *WSNThread*, pois podem ter suas cargas variáveis ao

longo do tempo, com momentos de recuperação de carga, de acordo com o padrão de consumo.

A comunicação interna entre os objetos nós da rede (*WSNNetworkNode*) e seus objetos agregados (*WSNRadio* e *WSNSensor*) deve ser implementada de acordo com a necessidade de detalhamento do projeto. Comumente, essas comunicações entre elementos de hardware são realizadas através de protocolos de comunicação do tipo SPI (*Serial Peripheral Interface*) ou I^2C (*Inter-Integrated Circuit*). Pode-se simular a execução desses protocolos de comunicação utilizando o método `wait()` para temporizar o tempo de transmissão e, de forma mais detalhada, utilizar métodos de interrupção para se realizar a comunicação entre os dispositivos. Ambas as funcionalidades estão disponíveis na arquitetura BaSS.

Com relação ao consumo de energia de um nó sensor, a soma da corrente elétrica utilizada por todos os seus componentes deve ser somada e fornecida em Ampères pelo método `getActualConsumptionInAmperes()`, o qual deve ser implementado nas subclasses de *WSNSensorNode*. Este método é utilizado de forma automática pelo simulador para decrementar a carga da bateria utilizada pelo nó sensor.

A Figura 5.4 apresenta um exemplo do método `main()` de um nó sensor (método sincronizado herdado de *WSNThread*) que realiza as operações de obter a temperatura ambiente e transmiti-la (nome do nó : identificador da mensagem : temperatura) a cada intervalo de 10 a 15 minutos (tempo definido aleatoriamente). Para cada operação ou conjunto de operações executado pelo nó sensor, há um tempo associado para sua realização, o qual deve ser indicado pela chamada do método `wait()` antes da sua execução.

A mudança de estado do nó sensor e dos seus dispositivos também deve ser realizada de forma cuidadosa. O método `getActualConsumptionInAmperes()`, a ser definido na classe do nó sensor (*WSNSensorNode*), deve levar em consideração essa mudança de estado e retornar o valor correto de consumo energético do nó associado ao seu estado atual. O tempo de permanência em determinado estado está, portanto, diretamente relacionado ao tempo de vida do nó.

5.3.2 – Transmissão de dados entre os nós da rede

Na arquitetura do WSN-BaSS, a transmissão de dados entre os nós da rede é coordenada pelo objeto da classe *WSNSimulator*, ou seja, o simulador propriamente dito. Este objeto, que contém todos os nós da rede pertencentes à simulação, possui métodos que trabalham em conjunto com os objetos que implementam a interface *WSNRadio*, pertencentes aos nós da rede, para determinar quais nós estão envolvidos em uma determinada transmissão.

```
01  public void main() {
02      while ( true ) {
03          //Nó requisita ao rádio que entre em estado WOR. Tempo utilizado: 1 ms
04          super.wait( new BaSSStandardTime( "00:00:00.001" ) );
05          this.radio.setState( Radio.RadioState.WOR );
06
07          //Nó entra em estado de POWER_SAVE. Tempo utilizado: 2 ms
08          super.wait( new BaSSStandardTime( "00:00:00.002" ) );
09          this.setState( SensorNodeState.POWER_SAVE );
10
11          //Nó aguarda de 10 a 15 minutos
12          super.wait( super.getSimulator().getRandom().nextBaSSTime(
                                            new BaSSStandardTime( "00:10:00.000" ),
                                            new BaSSStandardTime( "00:15:00.000" ) ) );
13
14          //Nó entra em estado de ACTIVE. Tempo utilizado: 1 ms
15          super.wait( new BaSSStandardTime( "00:00:00.001" ) );
16          this.setState( SensorNodeState.ACTIVE );
17
18          //Nó pede ao sensor que faça a leitura da temperatura. Tempo utilizado: 2 ms
19          super.wait( new BaSSStandardTime( "00:00:00.002" ) );
20          this.sensor.read();
21
22          //Nó aguarda 1 segundo para poder ler a temperatura
23          super.wait( new BaSSStandardTime( "00:00:01.000" ) );
24
25          //Nó lê a temperatura do sensor. Tempo utilizado: 1 ms
26          super.wait( new BaSSStandardTime( "00:00:00.001" ) );
27          temp = this.sensor.getValue();
28
29          //Nó verifica se o rádio já está transmitindo. Se estiver, aguarda.
30          while ( this.radio.getState() == Radio.RadioState.RX )
31              this.wait( new BaSSStandardTime( "00:00:00.001" ) );
32
33          //Nó coloca uma mensagem no buffer do rádio. Tempo utilizado: 5 ms
34          super.wait( new BaSSStandardTime( "00:00:00.005" ) );
35          this.radio.putBuffer( super.getName() + ":" + Integer.toString( ++id ) +
                                  ":" + Double.toString( temp ) );
36
37          //Nó requisita ao rádio que entre em estado TX. Tempo utilizado: 1 ms
38          super.wait( new BaSSStandardTime( "00:00:00.001" ) );
39          this.radio.setState( Radio.RadioState.TX );
40
41          //Nó gera uma interrupção no rádio, requisitando que ele transmita a mensagem
42          //colocada no buffer. Tempo utilizado: 1 ms
43          super.wait( new BaSSStandardTime( "00:00:00.001" ) );
44          this.radio.callInterruption( "transmitMessage" );
45
46      }
47  }
```

Figura 5.4 – Exemplo de implementação do método *main()* para um nó sensor.

O simulador possui um mapeamento de todas as conexões possíveis de serem estabelecidas entre os nós da rede. Em outras palavras, dado um conjunto de nós e suas posições no espaço, o simulador possui uma lista com todos os nós vizinhos entre si, ou seja, que podem se comunicar. Uma das condições utilizadas pelo WSN-BaSS para determinar essa possibilidade de comunicação é estabelecida pelo modelo de propagação de ondas no meio, que deve ser programada em uma classe que implemente a interface *WSNPropagationModel*.

A Figura 5.5 apresenta a definição da classe *FriisPropagationModel* que implementa a Equação de Friis (Equação 2.1 e Equação 2.2) como o modelo de propagação de ondas no

meio. Para cada par de rádios (`radioTx` e `radioRx`) e a distância entre ele (`distanceInMeters`), o método `getRxPowerInMilliWatt()`, a ser implementado para a interface *WSNPropagationModel*, retorna a potência do sinal que chega no rádio receptor.

```
01  public class FriisPropagationModel implements WSNPropagationModel {
02
03    public double getRxPowerInMilliWatt( WSNSimulator simulator, WSNRadio radioTx,
                                           WSNRadio radioRx, double distanceInMeters ) {
04     double f = radioTx.getFrequencyInHz();
05     double spaceLoss = Math.pow( (( 299792458/f )/( 4*Math.PI*distanceInMeters )), 2 );
06     //[dBm]->[mW]
07     double P_Tx_mW = Math.pow( 10, ( radioTx.getTxPowerIndBm() / 10 ) );
08     //[dBi]->[mW]
09     double gain_Tx_mW = Math.pow( 10, radioTx.getAntenna().getGainIndBi()/10 );
10     //[dBi]->[mW]
11     double gain_Rx_mW = Math.pow( 10, radioRx.getAntenna().getGainIndBi()/10 );
12     //[mW]
13     return ( gain_Tx_mW * gain_Rx_mW * spaceLoss * P_Tx_mW );
14    }
15  }
```

Figura 5.5 – Implementação da Equação de Friis como um modelo de propagação de ondas magnéticas.

A potência e a frequência de transmissão do rádio transmissor devem ser estabelecidas na implementação da classe *WSNRadio* através dos métodos `getTxPowerIndBm()` e `getFrequencyInHz()`, respectivamente.

Além desses parâmetros do rádio, na arquitetura do WSN-BaSS todo objeto implementado a partir da interface *WSNRadio* deve retornar um objeto implementado da interface *WSNAntenna* através do método `getAntenna()`. O ganho da antena, parâmetro comumente utilizado nos modelos de propagação de ondas, é determinado pelo método `getGainIndBi()` na implementação da classe *WSNAntenna*. O valor deve ser fornecido em dBi, que é o ganho em decibel em relação a uma antena isotrópica, isto é, que irradia ondas eletromagnéticas igualmente em todas as direções.

Contudo, não é só o modelo de propagação de ondas que define se um nó está no alcance de outro. O tipo de antena utilizado na transmissão também é um fator determinante. Na arquitetura WSN-BaSS, dois tipos de antenas podem ser definidos: omnidirecional ou direcional. Classes que implementam antenas no simulador (*WSNAntenna*), devem possuir os métodos `getDirection()` e `getBeamwidth()`, os quais determinam a direção e a abertura (largura) do feixe da antena, respectivamente.

A Figura 5.6 ilustra os dois tipos de antena. Uma antena omnidirecional deve ser definida com a abertura do feixe (β) igual a 360.0 (360°), não importando o valor atribuído para a direção da antena (α). Valores de β diferentes de 360.0 caracterizam uma antena direcional e o valor de α deve ser definido de acordo com a origem e orientação anti-horária apresentadas na Figura 5.6c.

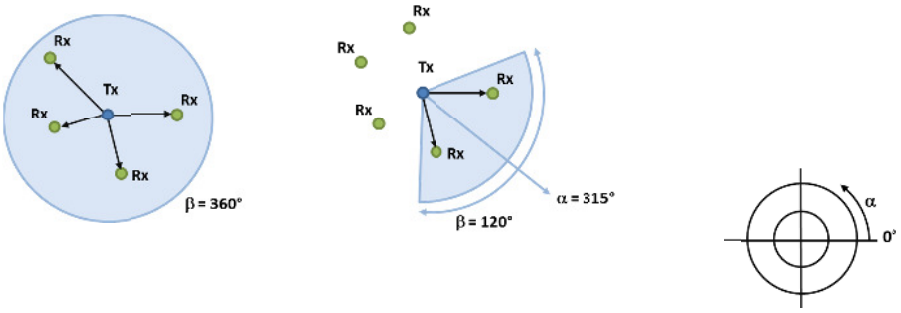


Figura 5.6 – Tipos de direcionamento de antena permitidos no WSN-BaSS. O ângulo α define a direção da antena e o ângulo β define a abertura do feixe. (a) Antena omnidirecional. (b) Antena direcional, com opção de seleção da direção e ângulo de abertura. (c) Origem e orientação anti-horária crescente do ângulo da direção da antena.

Além do modelo de propagação de ondas e do tipo da antena, há um terceiro fator a ser considerado para haver comunicação entre dois nós: a presença de obstáculos entre o nó transmissor e o nó receptor. Um obstáculo pode ser introduzido no simulador como um objeto da classe *WSNObstacle*, definido geometricamente como uma semirreta determinada por dois pontos e possuindo características atenuadoras de sinal. O construtor da classe *WSNObstacle* recebe como parâmetro a posição dos dois pontos e o valor de atenuação do sinal em dBm (decibel-miliwatt).

A Figura 5.7 apresenta dois casos de atenuação do sinal: no primeiro (Figura 5.7a), a atenuação do sinal é baixa, permitindo a comunicação; no segundo (Figura 5.7b), a atenuação do sinal é alta, não permitindo a comunicação.

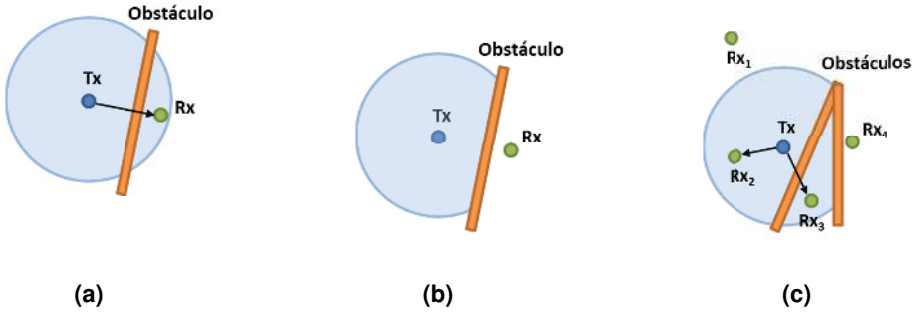


Figura 5.7 – Obstáculos atenuadores de sinal. (a) Obstáculo com baixa atenuação de sinal: a potência do sinal que atinge o receptor ainda é capaz de sensibilizá-lo. (b) Obstáculo com alta atenuação de sinal: a potência do sinal que atinge o receptor não é capaz de sensibilizá-lo. (c) Cenário com dois obstáculos, um nó transmissor (Tx) e quatro nós receptores (Rx_1, Rx_2, Rx_3 e Rx_4). Devido a atenuação do sinal provocado pelo modelo de propagação do sinal no meio, Rx_1 não recebe o sinal de Tx. Rx_2 está mais próximo de Tx, sendo o seu rádio sensibilizado pelo sinal exclusivamente atenuado pelo modelo de propagação do sinal no meio. Rx_3 encontra-se no mesmo caso de (a). Entre Tx e Rx_4 há dois obstáculos, cuja soma da atenuação, acrescida da atenuação do modelo de propagação, não permite a recepção do sinal.

A potência de recebimento do sinal em um rádio receptor leva em consideração, portanto, além da atenuação do sinal estabelecida no modelo de propagação de ondas, a atenuação do sinal provocada pelos obstáculos existentes entre o transmissor e o receptor. A Equação 5.1 apresenta a expressão utilizada pelo WSN-BaSS para o cálculo da potência recebida pelo rádio receptor.

A Equação 5.2 apresenta a conversão do valor de potência de dBm para mW para poder ser utilizada na Equação 5.1, uma vez que os valores de atenuação de sinal dos obstáculos comumente são encontrados na unidade dBm (Tabela 2.6), assim como este valor deve ser definido nesta unidade na classe *WSNObstacle*.

$$P_R = P_{MP} - \sum_{i=0}^{N} P_{O_i} \tag{5.1}$$

onde:
P_R = potência recebida [mW]
P_{MP} = potência calculada pelo modelo de propagação de ondas [mW]
N = quantidade de obstáculos entre o transmissor e o receptor
P_{Oi} = potência do sinal atenuada no obstáculo i [mW]

$$P_{mW} = 10^{(P_{dBm}/10)} \tag{5.2}$$

onde:
P_{mW} = potência em miliwatt [mW]
P_{dBm} = potência em decibel-milliwatt [dBm]

Portanto, além do direcionamento e abertura da antena, o simulador utiliza o modelo de propagação de ondas e a atenuação do sinal provocada por barreiras para verificar se a potência calculada na posição do nó receptor é suficiente para sensibilizar seu rádio, ou seja, se o receptor está no alcance do transmissor. Caso isso ocorra, o simulador considera que pode existir uma comunicação entre o nó transmissor e o nó receptor. Caso contrário, o nó receptor não está no alcance do nó transmissor e, portanto, não receberá mensagens enviadas pelo nó transmissor.

A sensibilidade do rádio receptor deve ser fornecida para o simulador em dBm através do método `getSensitivityIndBm()`, método de implementação obrigatória da interface *WSNRadio*.

Uma vez conhecidas as possibilidades de comunicação entre os rádios dos nós da rede, a transmissão de uma mensagem deve seguir uma sequência de etapas que envolvem o rádio transmissor (*WSNRadio*), o simulador (*WSNSimulator*) e nenhum ou um conjunto de

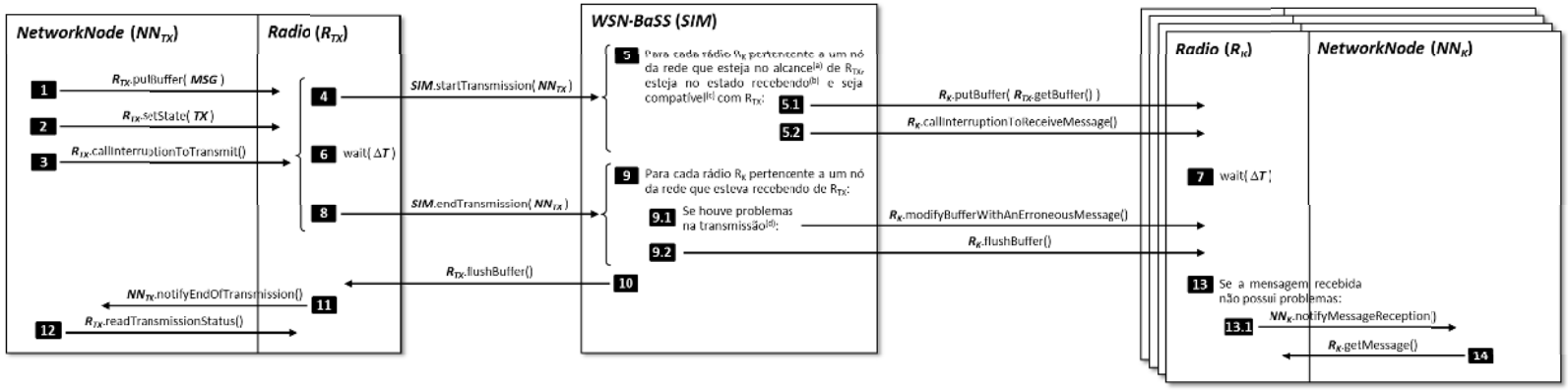
rádios receptores (*WSNRadio*). A Figura 5.8 apresenta essa sequência de etapas, considerando que os nós da rede e os rádios utilizados na simulação são objetos distintos e possuem capacidade de processamento independente, isto é, ambos são *WSNBaSS*: os nós da rede por serem subclasse de *WSNNetworkNode* e os rádios por serem subclasse de *WSNThread* (além de implementarem *WSNRadio*). Outras configurações, mais simples ou mais complexas, podem existir.

O nó da rede que deseja transmitir uma mensagem deve inicialmente verificar se o seu rádio não está transmitindo ou recebendo nenhuma outra mensagem. Caso isso não esteja ocorrendo, o nó da rede deve inserir a mensagem a ser transmitida no *buffer* do rádio através do método `putBuffer()`. A mensagem pode ser qualquer tipo de objeto (*Object*), como uma *String*, um array de bytes, etc. Em seguida, o nó da rede deve requisitar ao rádio para que ele mude seu estado para o de transmissão (Tx), fazendo com que o seu método `isTransmitting()` (interface WSNRadio) retorne *true*. Essa mudança de estado também implicará uma alteração no consumo de energia do rádio. A partir deste momento, o nó ficará livre para realizar outras tarefas e deixar a cargo do seu rádio a transmissão da mensagem. Isso pode ser feito através da chamada de uma interrupção do rádio, que cuidará da transmissão da mensagem.

Para iniciar a transmissão da mensagem, o rádio transmissor deve utilizar o método `startTransmission()` do simulador (*WSNSimulator*), passando a si próprio com argumento. O simulador identifica os nós que estão no alcance do rádio transmissor e cujos rádios estão no estado de recebimento (verificação feita através do método `isReceiving()` da interface *WSNRadio*). Para cada rádio que esteja recebendo, a mensagem do *buffer* do rádio de transmissão é copiada no *buffer* dos rádios que estão recebendo. Isto é feito através dos métodos `getBuffer()` e `putBuffer()`, respectivamente.

A partir deste momento, o rádio transmissor e o os rádios receptores (se houver algum) devem aguardar o tempo de simulação necessário para a transmissão da mensagem. Isso pode ser feito de duas maneiras distintas: (1) o rádio transmissor utiliza o método `wait()` passando como parâmetro o tempo necessário para a transmissão da mensagem e o método `waitSignal()` é chamado pelos rádios receptores, que posteriormente serão sinalizados; (2) tanto o rádio transmissor, como os rádios receptores chamam o método `wait()` passando como parâmetro o tempo necessário para a transmissão/recepção.

Neste segundo caso, os rádios receptores só terão o conhecimento do tempo necessário para o recebimento da mensagem se este vier junto com a mensagem inserida no *buffer* no início da transmissão ou se as transmissões forem de tamanho fixo, todas possuindo, portanto, o mesmo tempo de transmissão/recepção já conhecido.



***NetworkNode* (NN_{TX})**: Nó da rede que faz a requisição a seu rádio para transmitir a mensagem

***Radio* (R_{TX})**: Rádio do nó da rede NN_{TX} (rádio que transmite a mensagem)

***WSN-BaSS* (*SIM*)**: Simulador que controla a transmissão das mensagens

***NetworkNode* (NN_K)**: Um dos nós sensores que recebe a mensagem transmitida por NN_{TX}

***Radio* (R_K)**: Rádio do nó da rede NN_K (rádio que recebe a mensagem)

MSG: Mensagem transmitida (inserida no buffer do rádio R_{TX} e buffer do rádio R_K)

Δt: tempo necessário para a transmissão da mensagem MSG

(a): Um rádio receptor está no alcance de um rádio transmissor quando a potência recebida pelo receptor (potência de transmissão menos atenuação do sinal no meio) é suficiente para sensibilizá-lo. Na arquitetura do WSN-BaSS, o modelo de propagação do meio é definido em uma implementação da classe WSNPropagationModel do pacote br.eng.rsalustiano.wsn.

(b): Um rádio no estado recebendo significa que ele está apto a receber mensagem, geralmente identificado pelos estados RX ou WOR (wake up on radio) nos dispositivos. Na arquitetura WSN-BaSS este estado é verificado pelo método isReceiving() que deve ser implementado pelas classes da interface Radio (pacote br.eng.rsalustiano.wsn).

(c): A compatibilidade entre rádios pode levar em consideração diversos fatores, como a frequência, a taxa e o canal de transmissão, por exemplo. Na arquitetura WSN-BaSS essa verificação é realizada pelo método isCompatibleWith(WSNRadio radio) que deve ser implementado pelas classes da interface Radio (pacote br.eng.rsalustiano.wsn).

(d): A arquitetura WSN-BaSS prevê quatro tipos de problemas na transmissão: (1) colisão, (2) colisão por terminal escondido, (3) perda de conexão e (4) ruído do meio. Quando um desses eventos ocorrer durante a transmissão, a mensagem recebida deve ser substituída por uma que apresente erros. O método que faz isso é o modifyBufferWithAnErroneousMessage(), que deve ser implementado pelas classes da interface Radio (pacote br.eng.rsalustiano.wsn).

SEQUÊNCIA DE TRANSMISSÃO-RECEPÇÃO DE UMA MENSAGEM

(Considerando os nós sensores e os rádios com processamentos independentes, isto é, ambos são objetos da classe BaSSThread e se comunicam por interrupção)

A transmissão inicia através de uma requisição do NN_{TX} (nó da rede que quer transmitir a mensagem) para seu rádio (R_{TX}). Primeiramente **[1]** o nó NNTX insere no buffer do rádio R_{TX} a mensagem MSG que deseja transmitir. Em seguida, NN_{TX} **[2]** requisita ao seu rádio que entre no modo de transmissão e **[3]** chama uma interrupção que fará com que a mensagem seja transmitida. O rádio R_{TX} **[4]** informa ao simulador (SIM) que está dando início a uma transmissão. O simulador **[5]** localiza todos os nós da rede que estejam ao alcance de R_{TX}, no modo de recebimento e são compatíveis com R_{TX} e **[5.1]** coloca a mensagem nos seus *bufferes* . O método callInterruptionToReceiveMessage() de cada rádio receptor **[5.2]** é chamado de forma a disparar a rotina de recebimento da mensagem. Enquanto o rádio R_{TX} fica **[6]** bloqueado para transmitir durante o tempo Δt, os rádios receptores também **[7]** ficam bloqueados por esse mesmo tempo. Ao final da transmissão, o rádio R_{TX} **[8]** avisa ao simulador que terminou de enviar a mensagem. O simulador verifica **[9]** todos os rádios que estavam recebendo a mensagem e se **[9.1]** houve algum problema na transmissão, requisita que haja uma alteração na mensagem recebida. Em seguida requisita **[9.2]** o descarregamento do *buffer* nos rádios que receberam a mensagem e do **[10]** *buffer* rádio transmissor (R_{TX}). O rádio R_{TX} **[11]** notifica o nó NN_{TX} que a transmissão finalizou e o nó NN_{TX} **[12]** verifica o status da transmissão. Concomitantemente, os rádios que receberam a mensagem **[13]** verificam se ela está sem problemas e se sim, **[13.1]** notificam seus nós do recebimento de uma mensagem e o nó **[14]** faz a leitura da mensagem recebida.

Figura 5.8 – Sequência transmissão-recepção de uma mensagem na arquitetura WSN-BaSS.

Passado o tempo de transmissão da mensagem, o nó transmissor deve informar ao simulador o fim da transmissão através do método `endTransmission()`, passando como argumento a si próprio. Esse método faz com o que o simulador verifique quais nós estavam recebendo a mensagem (se algum) e para cada um deles é verificado se houve algum problema na transmissão[1] (colisão, perda de conexão ou ruído) e em caso afirmativo, o simulador pede para o nó em questão substituir o conteúdo do seu *buffer* por uma mensagem com conteúdo adulterado (errado).

Ainda para cada rádio receptor, o simulador chama o método `flushBuffer()`, sinalizando a finalização da transmissão. No caso dos rádios receptores estarem esperando um sinal, o método `flushBuffer()` deverá cuidar dessa sinalização através da chamada do método `signal()`. O último evento a ser executado pelo simulador é chamar o método `flushBuffer()` do rádio transmissor, sinalizando o final da transmissão.

No lado do nó transmissor, o rádio deve gerar uma interrupção no seu nó, indicando o final da transmissão. O nó, por sua vez, deve pedir ao rádio transmissor o *status* da transmissão. Cada implementação de rádio é diferente uma da outra, portanto o status pode ser o número de mensagens enviadas no caso do rádio só transmitir mensagens de tamanho fixo e a mensagem original for maior que este tamanho. Esta leitura de *status* pode nem existir em alguns casos.

No lado dos receptores, o rádio geralmente faz uma verificação da mensagem recebida (verificação do *CRC – Cyclic Redundancy Check* – da mensagem, por exemplo) e se ela está sem problemas, chama uma interrupção para o nó receptor, notificando o recebimento de uma mensagem. Em seguida, o nó lê essa mensagem do rádio.

Para evitar problemas de sincronização, o rádio transmissor sempre deverá estar com prioridade superior em relação às prioridades dos rádios receptores, particularmente se a espera pelo fim da transmissão/recepção da mensagem é realizado pelo método (2). Como a duração da transmissão inicia simultaneamente e tem a mesma duração, tanto para o transmissor como para o receptor, eles serão desbloqueados no mesmo tempo de simulação. Se o nó receptor estiver com prioridade maior que a do nó transmissor, aquele será desbloqueado primeiro e já verificará se a mensagem recebida não possui problemas (etapa 13 na Figura 5.8). Só depois, e no mesmo tempo de simulação, é que o rádio transmissor será desbloqueado e informará ao simulador o término da transmissão (etapa 8 na Figura 5.8), momento em que é feita a verificação dos problemas de transmissão.

A Figura 5.9 e a Figura 5.10 mostram os métodos que implementam essa sequência

[1] Os problemas de transmissão serão abordados separadamente no final deste item.

de etapas apresentadas na Figura 5.8 nos objetos da classe *WSNRadio*. O método utilizado para a espera durante o tempo de transmissão da mensagem é o (1), isto é, o que utiliza sinalização.

```
01  //Método a ser executado no início da simulação
02  public void init() {
03      try {
04          //Registra a interrupção para recebimento de mensagem
05          super.addInterruptionEvent( new BaSSInterruption( "receiveMessage",
                                                              "rx", 1, true ) );
06          //Registra a interrupção para transmissão de mensagem
07          super.addInterruptionEvent( new BaSSInterruption( "transmitMessage",
                                                              "tx", 0, true ) );
08      } catch ( Exception e ) { }
09  }
10
11  //Método da interface WSNRadio responsável por chamar a interrupção para
12  //recebimento de mensagem
13  public void callInterruptionToReceiveMessage() {
14      super.callInterruption( "receiveMessage" );
15  }
16
17  //Método da interrupção para transmissão de mensagens
18  public void tx() {
19      //Rádio avisa ao simulador para iniciar a transmissão da mensagem
20      super.getSimulator().startTransmission( this );
21      //Rádio aguarda o tempo necessário para a transmissão: 20 ms
22      this.wait( new BaSSStandardTime( "00:00:00.020" ) );
23      //Rádio avisa ao simulador para finalizar a transmissão da mensagem
24      super.getSimulator().endTransmission( this );
25      //Rádio entra no estado de WOR. Tempo utilizado: 1 ms
26      this.wait( new BaSSStandardTime( "00:00:00.001" ) );
27      this.setState( RadioState.WOR );
28  }
29
30  //Método da interrupção para recepção de mensagens
31  public void rx() {
32      //Rádio entra no estado de RX. Tempo utilizado: 1 ms
33      this.wait( new BaSSStandardTime( "00:00:00.001" ) );
34      this.setState( RadioState.RX );
35      //Rádio aguarda o sinal de final de transmissão
36      this.waitSignal();
37      //Se há alguma mensagem recebida
38      if ( !this.msgQueue.isEmpty() ) {
39          //Rádio gera uma interrupção do nó, avisando que há uma nova mensagem
40          //para ser recebida. Tempo utilizado: 2 ms
41          this.wait( new BaSSStandardTime( "00:00:00.002" ) );
42          this.networkNode.callInterruption( "receiveMessage" );
43      }
44      //Rádio entra no estado de WOR. Tempo utilizado: 1 ms
45      this.wait( new BaSSStandardTime( "00:00:00.001" ) );
46      this.setState( RadioState.WOR );
47  }
48  //Método chamado pelo simulador ao término do envio e recebimento de mensagens
49  //para descarregar o buffer com a mensagem transmitida ou recebida.
50  public void flushBuffer() {
51      //Se o rádio está no estado RX e a mensagem não tem erro de recebimento
52      //Tempo utilizado: 5 ms
53      this.wait( new BaSSStandardTime( "00:00:00.005" ) );
54      if ( ( this.state == Radio.RadioState.RX ) &&
             ( !this.buffer.toString().equals( Radio.errorMessage ) ) )
56          //Enfila a mensagem recebida.
57          this.msgQueue.add( new String( this.buffer.toString() ) );
58      }
59
```

Figura 5.9 – Implementação de métodos da interface WSNRadio necessários para a transmissão/recepção de mensagens (parte 1/2).

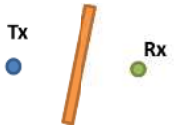
```
60      //Limpa o buffer do Rádio. Tempo utilizado: 1 ms
61      this.wait( new BaSSStandardTime( "00:00:00.001" ) );
62      this.buffer = null;
63      //Se o estado do radio é RX (o rádio está recebendo), então sinaliza para
64      //finalizar o recebimento da mensagem.
65      if ( this.state == RadioState.RX ) {
            super.signal();
66      }
67  }
```

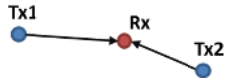
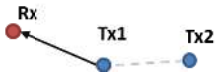
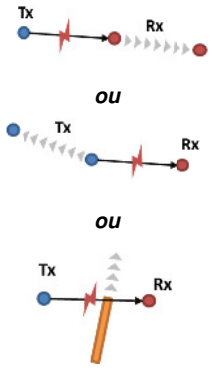
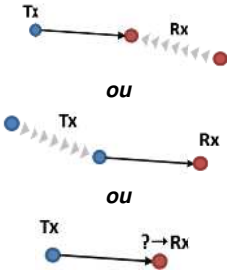
Figura 5.10 – Implementação de métodos da interface WSNRadio necessários para a transmissão/recepção de mensagens (parte 2/2).

A responsabilidade do simulador, no que se refere à transmissão dos dados, não somente está relacionada à determinação dos nós que receberão as mensagens e ao processo de inicialização e finalização da transmissão, mas também verificar se houve algum problema durante a transmissão.

A Tabela 5.2 apresenta todas as situações previstas pelo modelo do WSN-BaSS em relação à comunicação entre os rádios dos nós. Com relação aos erros de transmissão, o simulador realiza, a cada avanço do tempo de simulação, uma verificação nas condições de transmissão e recepção de todos os nós da rede, registrando os eventos que possam resultar em erros de transmissão. Ao final de cada transmissão (etapa 9.1 da Figura 5.8), esses registros são verificados e caso tenha havido algum problema na transmissão, o simulador requisita ao nó receptor que a alteração do conteúdo do seu *buffer* por uma mensagem com erros.

Tabela 5.2 – Situações que podem ocorrer durante a transmissão de mensagens

Situação	Descrição	Resultado
	*ATENUAÇÃO DO SINAL* A potência do sinal transmitido pelo rádio do nó Tx não é suficiente para sensibilizar o rádio do nó Rx devido à distância entre eles.	O nó Rx não recebe a mensagem transmitida.
	*RECEPÇÃO DO SINAL* A potência do sinal transmitido pelo rádio do nó Tx é suficiente para sensibilizar o rádio do nó Rx.	O nó Rx recebe a mensagem transmitida.
	*BARREIRA ATENUADORA DE SINAL* Apesar da potência de transmissão do rádio Tx ser suficiente para sensibilizar o rádio do nó Rx, há uma barreira atenuadora do sinal entre os nós que impede a recepção.	O nó Rx não recebe a mensagem transmitida.

	*COLISÃO* O rádio do nó Rx recebe o sinal transmitido pelos rádios dos nós Tx1 e Tx2. Há interferência entre os sinais transmitidos.	O nó Rx recebe uma mensagem com erro.
	*COLISÃO POR TERMINAL ESCONDIDO* O rádio do nó Rx recebe o sinal transmitido pelo rádio do nó Tx1 com interferência do sinal transmitido pelo rádio do nó Tx2. A interferência do sinal de Tx2 no sinal de Tx1 pode ocorrer durante toda a transmissão de Tx1 ou somente em parte dela.	O nó Rx recebe uma mensagem com erro.
	*PERDA DE CONEXÃO* *Caso 1*: O rádio do nó Rx estava recebendo o sinal do rádio do nó Tx, quando aquele se moveu e saiu do alcance deste. *Caso 2*: O rádio do nó Rx estava recebendo o sinal do rádio do nó Tx, quando este se moveu e saiu do alcance daquele. *Caso 3*: O rádio do nó Rx estava recebendo o sinal do rádio do nó Tx, quando uma barreira entra entre os nós e impende a comunicação. *Caso 4*: Qualquer combinação dos casos anteriores.	O nó Rx recebe uma mensagem com erro.
	*ESTABELECIMENTO DE CONEXÃO TARDIO* *Caso 1*: O rádio do nó Rx entra na área de cobertura do sinal transmitido pelo rádio do nó Tx depois que este já havia iniciado a transmissão. *Caso 2:* O rádio transmissor do nó Tx entra na área de recepção do rádio do nó Rx depois que aquele já havia iniciado a transmissão. *Caso 3: O rádio do nó Rx iniciou a recepção do sinal transmitido pelo rádio do nó Tx, isto é, mudou o estado do rádio para permitir a recepção, depois que este já havia iniciado a transmissão.*	O nó Rx não recebe a mensagem, pois seu rádio não detecta o preâmbulo da mensagem transmitida.
	*RUÍDO* O sinal transmitido pelo rádio do nó Tx é recebido pelo rádio do nó Rx, mas foi corrompido durante a transmissão pela ação de um sinal de interferência do meio de transmissão.	O nó Rx recebe uma mensagem com erro.

Com relação à situação de ruído do meio que possa interferir no conteúdo da mensagem durante a transmissão, este é determinado por uma variável de ambiente (item 5.3.3) cujos valores devem ser probabilísticos ([0,1]). Para cada transmissão, o simulador gera um número aleatório entre 0 e 1 utilizando a classe *BaSSRandom* (Capítulo 4) e verifica se

esse número gerado é menor ou igual ao valor da probabilidade de ocorrer a interferência na onda transmitida tanto na posição do transmissor (ou na área dele) como na posição do receptor (ou na área dele). Essa verificação é realizada para todas as variáveis do tipo ruído. Se pelo menos uma das verificações resultar em verdadeira, o ruído é aplicado à transmissão e a mensagem recebida deve ser alterada para uma mensagem com erros (etapa 9.1 da Figura 5.8).

Alguns rádios possuem registradores que armazenam o RSSI (*Received Signal Strength Indication* – Indicador da "força" do sinal recebido) quando recebem uma mensagem. Este valor pode ser obtido pelo método `getRSSI()` da classe WSNSimulator.

5.3.3 – Variáveis do ambiente e sensores

Os nós sensores são posicionados em um espaço bidimensional que possui um conjunto de variáveis ambientais e cujos sensores são responsáveis por mensurar. Na arquitetura WSN-BaSS, as variáveis ambientais são classes que implementam a interface *WSNVariable* e os sensores como classes que implementam a interface *WSNSensor* , ambas do pacote `br.eng.rsalustiano.wsn`.

Uma classe que descreva o comportamento de uma variável no ambiente, isto é, uma implementação de *WSNVariable*, deve necessariamente conter o método `getValue( WSNSimulator simulator, WSNPosition position )`, no qual deve ser retornado um valor do tipo `double` que indique o valor da variável na posição `position`, ou seja, na coordenada (`position.x, position.y`). Este método, portanto, é responsável por indicar o valor da variável na posição em que o nó se encontra.

O sensor (implementação de *WSNSensor*) deve utilizar o método `getVariableValue( Class<?> variableClass, WSNSensor sensor )` do simulador para obter o valor de uma determinada variável na posição do seu nó.

Dessa forma, supondo que a classe *VariableTemperature* tenha sido desenvolvida como uma implementação de *WSNVariable* e possua as temperaturas do ambiente. Para que um sensor obtenha a temperatura na posição do seu nó, ele deve utilizar o método `simulator.getVariableValue( VariableTemperature.class, this )`. O simulador retornará o valor da temperatura na posição do nó que contém o sensor.

Conforme visto no item 5.3.1, os nós sensores podem possuir um ou mais sensores distintos. Alguns sensores são componentes eletrônicos analógicos e outros já possuem uma eletrônica digital embutida. O gasto energético dos primeiros está relacionado à transdução da grandeza física e a conversão A/D (analógico-digital). Nos segundos, além da transdução e

conversão A/D, há ainda o gasto com a comunicação digital entre o sensor e o nó, este representando um microcontrolador.

Seja qual for o tipo de sensor, a contabilidade do gasto energético sempre é realizada pelo nó sensor (método `getActualConsumptionInAmperes()`). Os sensores devem, portanto, fornecer ao nó sensor a corrente elétrica que estão consumindo em cada estado que se encontrem (fazendo a transdução, deligados, em comunicação, etc.).

Nos sensores eletrônicos o processamento dos dados e o controle da comunicação com o nó dependem de sincronização. Portanto, esses tipos de sensores devem ser desenvolvidos como uma subclasse de *WSNThread*, além de implementarem *WSNSensor*. O detalhamento na comunicação entre o sensor e o nó deve ser programado de acordo com a necessidade de cada projeto.

Com relação às variáveis ambientais, elas devem ser dinâmicas para que suas representações sejam mais realísticas. Os valores determinados para cada posição do ambiente, portanto, devem se alterar ao longo do tempo. Isso é possível de ser realizado fazendo com que a variável seja desenvolvida como uma subclasse de *WSNThread* e as mudanças nas condições da variável sejam realizadas no método sincronizado `main()` (Capítulo 3).

As variáveis ambientais também podem representar ruídos que geram interferência nas transmissões (item 5.3.2). Neste caso, a implementação do método `isNoise()` da interface *WSNVariable* deve retornar `true` e os valores retornados pelo método `getValue()` devem representar uma probabilidade de ocorrer o ruído, ou seja, devem estar no intervalo [0,1].

5.3.4 – *Loggers* e atualizadores

Os *loggers* e os atualizadores (*updaters*) da arquitetura do WSN-BaSS apresentam a mesma estrutura e funcionamento que os encontrados na arquitetura do BaSS (item 4.6.2). As interfaces *WSNLogger* e *WSNUpdater* devem ser implementadas, respectivamente, no desenvolvimento de novos *loggers* e atuadores para simulações no WSN-BaSS.

Todos os eventos ocorridos no simulador relacionados à transmissão de mensagens (início e fim de transmissão, início e fim de recepção) e controle da simulação (início e fim de simulação, pausa e retorno na simulação) são reportados pelo simulador nos *loggers*.

5.3.5 – O controle da simulação

Conforme visto, a classe *WSNSimulator* é responsável pela coordenação da simulação. Ela é uma extensão da classe *BaSSControl*, herdando todos os mecanismos de

controle do BaSS necessários à sincronização dos objetos. Além disso, a classe *WSNSimulator* implementa *BaSSUpdater*, executando um conjunto de ações toda vez que o relógio da simulação é incrementado, isto é, trata-se de um atualizador do tipo SIMULATION_TIME_ UPDATER.

Esse conjunto de ações é que de fato compõem o mecanismo específico do simulador de Redes de Sensores Sem Fio. A Figura 5.11 apresenta o pseudocódigo referente a essas ações.

```
 0   last_update_time ← 00:00:00.000;
     void update( BaSSControl control ) {
 1       delta_time ← ( control.getSimulationTime() – last_upate_time );
 2       Para cada nó sensor (WSNSensorNode) sn da rede:
2.1          Se !sn.getBattery().isEmpty(), então
2.1.1            consumption ← sn.getActualConsumaptionInmA() * delta_time;
2.1.2            sn.getBattery().use( consumption )
2.2          Se sn.getBattery().isEmpty(), então
2.2.1            Finaliza sn;
2.2.2            Remove sn da rede;
2.3          Atualiza as conexões da rede;
 3       Se não existe mais nenhum nó sensor (WSNSensorNode) na rede, então:
3.1          Para cada estação radio base (WSNBaseStation) bs da rede:
3.1.1            Finaliza bs;
3.1.2            Remove bs da rede;
3.2          Finaliza a simulação;
 4       Verifica a situação das mensagens em transmissão;
 5       last_update_time ← actual_simulation_time;
     }
```

Figura 5.11 – Pseudocódigo com as ações executadas a cada avanço do tempo de simulação na arquitetura do WSN-BaSS.

No decorrer da simulação, um atributo (`last_update_time`) é utilizado para registrar o momento (tempo de simulação) em que a última atualização foi realizada. Inicialmente esse valor é zero. Toda vez que o método `update()` é chamado, calcula-se o intervalo de tempo desde a última execução do `update()` até a chamada atual (`delta_time`). Para cada nó sensor da rede é verificado se ainda existe carga na sua bateria (`!sn.getBattery().isEmpty()`) e caso ainda exista é utilizada uma quantidade de energia da bateria de acordo com o `delta_time` e o consumo atual do nó (`sn.getActualConsumptionInAmperes()`). Caso não haja mais carga na bateria, o nó sensor é retirado da rede. Após a retirada dos nós que "morreram", se não existir mais nenhum nó sensor na rede, então a execução de todas as estações rádio base da rede é finalizada e a simulação chega ao fim.

Portanto, o tempo de vida da Rede de Sensores Sem Fio considerado no simulador WSN-BaSS foi estabelecido de acordo com a definição (3) do item 2.5, ou seja, uma rede chega ao fim a partir do momento em que todos os nós sensores estejam sem carga. Caso seja importante a verificação de outra condição para declarar o final da operação da rede, o programador poderá implementar uma classe *WSNUpdater* e fazer a verificação nela. Quando o critério for atingido, pode-se utilizar o método `stop()` para finalizar a simulação ou simplesmente registrar o evento em algum logger.

Além da verificação das condições de energia dos nós, a cada execução do método `update()` o simulador também verifica a situação das mensagens em transmissão, conforme apresentado no item 5.3.2.

5.4 – Criação de redes, configuração e execução das simulações

Uma vez definidas as classes necessárias à simulação e programado o funcionamento dos nós e demais objetos sincronizáveis, deve-se definir uma rede e os parâmetros de simulação. Em seguida, a simulação pode ser executada.

Há duas maneiras de se criar uma rede e definir os parâmetros de uma simulação: utilizando a programação Java (modo direto) ou através de um documento XML (modo indireto) que é processado e executado de acordo com seu conteúdo.

5.4.1 – Criação de redes utilizando a programação Java

A criação de uma rede utilizando a programação Java deve ser feita seguindo o exemplo apresentado na Figura 5.12. Um objeto da classe WSNSimulator deve ser criado passando como parâmetro a classe do modelo de propagação de ondas. Em seguida, deve-se acrescentar no simulador as classes dos nós da rede permitidas na simulação.

A partir desses comandos, pode-se definir os demais parâmetros de controle da simulação, como gerador de números aleatórios (`setRandom()`), limite de tempo de simulação (`setSimulationTimeLimit()`) e outros parâmetros da arquitetura do BaSS.

Os nós da rede devem ser inseridos na simulação através do método `addNetworkNode()`, passando como parâmetros o nó (*WSNBaseStation* ou *WSNSensorNode*) e a sua posição (*WSNPosition*). Os obstáculos são inseridos utilizando o método `addObstacle()`. As variáveis de ambiente são inseridas na simulação através do método `addVariable()`.

```
01  WSNSimulator sim = new WSNSimulator( FriisPropagationModel.class );
02
03   sim.addAllowedNetworkNodeClass( BaseStation.class );
04   sim.addAllowedNetworkNodeClass( SensorNode.class );
05
06   sim.setRandom( new BaSSRandom( 1 ) );
07
08   sim.addNetworkNode( new BaseStation( "BS" ), new WSNPosition( 0.00, 0.00 ) );
09   sim.addNetworkNode( new SensorNode( "SN01" ), new WSNPosition( 40.00, 0.00 ) );
10   sim.addNetworkNode( new SensorNode( "SN02" ), new WSNPosition( 65.00, 20.00 ) );
11   sim.addNetworkNode( new SensorNode( "SN03" ), new WSNPosition( 65.00, -20.00 ) );
12   sim.addNetworkNode( new SensorNode( "SN04" ), new WSNPosition( 95.00, 20.00 ) );
13   sim.addNetworkNode( new SensorNode( "SN05" ), new WSNPosition( 120.00, 45.00 ) );
14   sim.addNetworkNode( new SensorNode( "SN06" ), new WSNPosition( 120.00, -5.00 ) );
15   sim.addNetworkNode( new SensorNode( "SN07" ), new WSNPosition( 120.00, -40.00 ) );
16   sim.addNetworkNode( new SensorNode( "SN08" ), new WSNPosition( 140.00, -60.00 ) );
17   sim.addNetworkNode( new SensorNode( "SN09" ), new WSNPosition( 160.00, -40.00 ) );
18   sim.addNetworkNode( new SensorNode( "SN10" ), new WSNPosition( 160.00, -10.00 ) );
19
20   sim.addObstacle( new WSNObstacle( 60.00, 0.00, 90.00, 0.00, -6.0 ) );
21   sim.addObstacle( new WSNObstacle( 140.00, 45.00, 140.00, -40.00, -6.0 ) );
22
23   VariableTemperatureDinamic ctd = new VariableTemperatureDinamic( "Temperature" );
24   sim.addVariable( ctd );
25   sim.addThread( ctd );
26
27   sim.addLogger( new FileLogger( "log001.log" ) );
28
29   sim.start();
```

Figura 5.12 – Exemplo de criação de uma rede utilizando a programação Java.

Os *loggers* e atualizadores são inseridos pelos métodos `addLogger()` e `addUpdater()`, como na arquitetura do BaSS, assim como as *WSNThreads* são inseridas pelo método `addThread()`.

Contudo, se um objeto exercer mais de uma função na simulação, como é o caso do objeto `ctd` da classe *VariableTemperatureDinamic*, que é uma subclasse de *WSNThread* e implementação de *WSNVariable*, ele deve ser inserido na simulação utilizando todos os métodos de inserção cabíveis. No caso o objeto `ctd`, foi inserido como uma thread e como uma variável.

Assim como na arquitetura BaSS, a simulação tem início com a chamada do método `start()` e a condição de parada pode ser estabelecida por algum critério de limite de tempo ou pela finalização da operação da rede conforme estabelecido no algoritmo da Figura 5.11.

5.4.2 – Criação de redes utilizando documentos XML

Um arquivo escrito na linguagem XML (*eXtensible Markup Language* – Linguagem de Marcação Extensível) [DEI00] pode ser utilizado para configurar uma simulação de uma Rede de Sensores Sem Fio no WSN-BaSS. A Figura 5.13 apresenta o DTD (*Document Type*

Definition – Definição de Tipo de Documento) com as regras que definem as *tags*[2] e os atributos permitidos na descrição da simulação.

A principal *tag* de um arquivo XML utilizado na descrição de uma simulação na arquitetura WSN-BaSS é a <WSNSIMULATOR>. Ela é utilizada para definir os parâmetros da simulação e é a *tag* de maior hierarquia que compõem o arquivo XML. Cada *tag* interna à <WSNSIMULATOR> define um tipo de objeto que deve ser criado pelo simulador e inserido na simulação. A Tabela 5.3 apresenta as *tags* XML do WSN-BaSS e uma breve descrição dos seus atributos obrigatórios para a criação dos objetos e utilização dos mesmos na simulação. As classes dos WSN-BaSS relacionadas às *tags* também são apresentadas nessa tabela.

```
01  <?xml version="1.0" encoding="UTF-8"?>
02
03  <!ELEMENT WSNSIMULATOR ( NETWORKNODE*, OBSTACLE*, LOGGER*, UPDATER*, VARIABLE*,
                             ATTRIBUTE*, THREAD*, GENERIC* )*>
04  <!ATTLIST WSNSIMULATOR propagation_model_class CDATA #REQUIRED>
05  <!ATTLIST WSNSIMULATOR simulation_time_limit CDATA #IMPLIED>
06  <!ATTLIST WSNSIMULATOR wallclock_time_limit CDATA #IMPLIED>
07  <!ATTLIST WSNSIMULATOR random_seed CDATA #IMPLIED>
08  <!ATTLIST WSNSIMULATOR delay_in_millis CDATA #IMPLIED>
09  <!ATTLIST WSNSIMULATOR delay_type ( EVENT_DELAY | SIMULATION_TIME_DELAY ) #IMPLIED>
10  <!ATTLIST WSNSIMULATOR allowed_network_node_classes CDATA #REQUIRED>
11
12  <!ELEMENT NETWORKNODE ( ATTRIBUTE* )>
13  <!ATTLIST NETWORKNODE class CDATA #REQUIRED>
14  <!ATTLIST NETWORKNODE name ID #REQUIRED>
15  <!ATTLIST NETWORKNODE x CDATA #REQUIRED>
16  <!ATTLIST NETWORKNODE y CDATA #REQUIRED>
17
18  <!ELEMENT OBSTACLE ( ATTRIBUTE* )>
19  <!ATTLIST OBSTACLE x1 CDATA #REQUIRED>
20  <!ATTLIST OBSTACLE y1 CDATA #REQUIRED>
21  <!ATTLIST OBSTACLE x2 CDATA #REQUIRED>
22  <!ATTLIST OBSTACLE y2 CDATA #REQUIRED>
23  <!ATTLIST OBSTACLE attenuation_in_dBm CDATA #REQUIRED>
24
25  <!ELEMENT LOGGER ( ATTRIBUTE* )>
26  <!ATTLIST LOGGER class CDATA #REQUIRED>
27
28  <!ELEMENT UPDATER ( ATTRIBUTE* )>
29  <!ATTLIST UPDATER class CDATA #REQUIRED>
30
31  <!ELEMENT VARIABLE ( ATTRIBUTE* )>
32  <!ATTLIST VARIABLE class CDATA #REQUIRED>
33
34  <!ELEMENT ATTRIBUTE EMPTY>
35  <!ATTLIST ATTRIBUTE name CDATA #REQUIRED>
36  <!ATTLIST ATTRIBUTE value CDATA #REQUIRED>
37
38  <!ELEMENT THREAD ( ATTRIBUTE* )>
39  <!ATTLIST THREAD class CDATA #REQUIRED>
40  <!ATTLIST THREAD name ID #REQUIRED>
41
42  <!ELEMENT GENERIC ( ATTRIBUTE* )>
43  <!ATTLIST GENERIC class CDATA #REQUIRED>
```

Figura 5.13 – DTD com as regras que definem os elementos e os atributos permitidos para a descrição de uma simulação na arquitetura WSN-BaSS.

[2] *Tag* (etiqueta) é uma palavra-chave que identifica um dado como sendo de um determinado tipo. Tal associação permite a classificação do dado.

Tabela 5.3 – *Tags* do documento XML utilizadas para definir uma simulação

Tag	Descrição e atributos obrigatórios	Classe relacionada do WSN-BaSS
<WSNSIMULATOR>	Define uma simulação e determina suas propriedades: `propagation_model_class` (a classe que define o modelo de propagação de ondas eletromagnéticas) e `allowed_network_node_classes` (lista com as classes que definem os nós de rede que podem fazer parte da simulação)	**WSNSimulator**
<NETWORKNODE>	Insere na simulação um nó da rede, objeto de uma subclasse de *WSNNetworkNode*. O atributo `class` identifica a classe do nó, `name` o identificador do nó para a simulação, `x` e `y` as coordenadas do nó.	Qualquer subclasse de **WSNNetworkNode**
<OBSTACLE>	Insere na simulação um obstáculo, objeto da classe *WSNObstacle*. Os atributos `x1` e `y1` são da primeira coordenada do obstáculo e os atributos `x2` e `y2` são da segunda coordenada. A atenuação do sinal é definida em dBm e indicada no parâmetro `attenuation_in_dBm`.	**WSNObstacle**
<LOGGER>	Insere na simulação um objeto que implemente *WSNLogger*, isto é, um *logger*. O atributo `class` determina a classe desse objeto.	Qualquer classe que implemente **WSNLogger**
<UPDATER>	Insere na simulação um objeto que implemente *WSNUpdater*, isto é, um atualizador. O atributo `class` determina a classe desse objeto.	Qualquer classe que implemente **WSNUpdater**
<VARIABLE>	Insere na simulação um objeto que implemente *WSNVariable*. O atributo `class` determina a classe desse objeto.	Qualquer classe que implemente **WSNVariable**
<ATTRIBUTE>	*Tag* que define um par atributo-valor para ser atribuído a qualquer outro elemento (objeto) da simulação que pertença a uma classe que implemente *WSNConfigurable*. Os parâmetros `name` e `value` são utilizados para definir o nome do atributo e seu valor, respectivamente.	Qualquer classe que implemente **WSNConfigurable**
<THREAD>	Insere na simulação um objeto da classe *WSNThread*. Quando a simulação iniciar, este objeto é executado em sincronização com os outros objetos *WSNThread* e *BaSSThread*. Se o objeto implementar outras interfaces (*WSNLogger*, *WSNUpdater* e/ou *WSNVariable*), ele será automaticamente inserido como essas funcionalidades também.	Qualquer subclasse de **WSNThread**
<GENERIC>	Esta *tag* é utilizado para inserir um objeto na simulação que possua mais de uma funcionalidade. Ele deve ser utilizado no caso de uma classe implementar mais de uma das seguintes uma interfaces: *WSNLogger*, *WSNUpdater* e/ou *WSNVariable* e não ser um objeto *WSNThread*. O atributo `class` especifica a classe desse objeto.	Qualquer classe que implemente **WSNLogger** e/ou **WSNUpdater** e/ou **WSNVariable**

No caso de uma classe implementar mais de uma interface, *WSNLogger* e *WSNUpdater*, por exemplo, se um objeto dessa classe for definido através da *tag* `<LOGGER>`, ele será inserido na simulação apenas como um *logger*; se for definido pela *tag* `<UPDATER>`, ele se comportará apenas como um atualizador. Se essa classe for utilizada na definição de um objeto através da *tag* `<LOGGER>` e no mesmo documento XML for utilizada na definição de um objeto através da *tag* `<UPDATER>`, dois objetos independentes serão criados: um com a funcionalidade de *logger* e outro com a funcionalidade de atualizador. Para que um mesmo objeto dessa classe possua as duas funcionalidades concomitantemente, ele deve ser definido no documento XML com a *tag* `<GENERIC>`.

No caso da definição de um objeto que seja uma *WSNThread* e implemente outras interfaces, quando utilizada a *tag* `<THREAD>`, o objeto já é inserido na simulação com as outras funcionalidades referentes às interfaces.

A Figura 5.14 apresenta um exemplo de documento XML que cria uma simulação idêntica à criada pela programação Java da Figura 5.12.

```
01    <?xml version="1.0" encoding="UTF-8"?>
02
03    <!DOCTYPE WSNSIMULATOR SYSTEM "wsnsimulator.dtd">
04
05    <WSNSIMULATOR propagation_model_class="wsntese.FriisPropagationModel" random_seed="1"
                  allowed_network_node_classes="wsntese.SensorNode,wsntese.BaseStation">
06
07        <NETWORKNODE class="wsntese.BaseStation" name="BS" x="0.00" y="0.00"/>
08        <NETWORKNODE class="wsntese.SensorNode" name="SN01" x="40.00" y="0.00"/>
09        <NETWORKNODE class="wsntese.SensorNode" name="SN02" x="65.00" y="20.00"/>
10        <NETWORKNODE class="wsntese.SensorNode" name="SN03" x="65.00" y="-20.00"/>
11        <NETWORKNODE class="wsntese.SensorNode" name="SN04" x="95.00" y="20.00"/>
12        <NETWORKNODE class="wsntese.SensorNode" name="SN05" x="120.00" y="45.00"/>
13        <NETWORKNODE class="wsntese.SensorNode" name="SN06" x="120.00" y="-5.00"/>
14        <NETWORKNODE class="wsntese.SensorNode" name="SN07" x="120.00" y="-40.00"/>
15        <NETWORKNODE class="wsntese.SensorNode" name="SN08" x="140.00" y="-60.00"/>
16        <NETWORKNODE class="wsntese.SensorNode" name="SN09" x="160.00" y="-40.00"/>
17        <NETWORKNODE class="wsntese.SensorNode" name="SN10" x="160.00" y="-10.00"/>
18
19        <OBSTACLE x1="60.00" y1="0.00" x2="90.00" y2="0.00" attenuation_in_dBm="-6.0"/>
20        <OBSTACLE x1="140.00" y1="45.00" x2="140.00" y2="-40.00" attenuation_in_dBm="-6.0"/>
21
22        <LOGGER class="wsntese.FileLogger">
23            <ATTRIBUTE name="fileName" value="log001.log"/>
24        </LOGGER>
25
26        <THREAD class="wsntese.VariableTemperatureDinamic" name="Temperature"/>
27
28    </WSNSIMULATOR>
```

Figura 5.14 – Exemplo de um documento XML que define os parâmetros de uma simulação e os elementos da rede.

A operação de leitura de um arquivo XML e a transformação do mesmo em uma simulação é feita pela classe *WSNXMLParser*, pertencente ao pacote `br.rsalustiano.wsn.parser`. Esta classe possui, dentro outros métodos de manipulação de

documentos XML relacionados ao WSN-BaSS, o método estático `main()`, que pode ser utilizado para execução da simulação em um terminal. Deve-se, para isso, executar a classe *WSNXMLParser* passando como argumento um arquivo contendo um documento XML descritor da rede. O método faz a sua leitura e validação do documento XML, transforma das *tags* em objetos e executa a simulação.

5.5 – Grafos

Os grafos são estruturas de dados fundamentais para o estudo de redes, pois eles permitem a realização de uma série de verificações na estrutura da rede que auxiliam na apuração da eficiência dos protocolos de comunicação utilizados. A possibilidade de conexão entre os nós, a existência de partições na rede e a determinação de caminhos mínimos entre nós utilizando diferentes critérios são apenas algumas das análises que podem ser realizadas com o auxílio de grafos.

As quatro classes do pacote `br.eng.rsalustiano.wsn.graph` apresentadas na Figura 5.2, *WSNGraph*, *WSNGraphEdge*, *WSNGraphVertice* e *WSNGraphPath*, são utilizadas na arquitetura do WSN-BaSS para a criação de grafos a partir da estrutura de rede formada pelos nós inseridos na simulação.

Cada vértice do grafo é um objeto da classe *WSNGraphVertice*, o qual possui um objeto do tipo *WSNNetworkNode*, um objeto do tipo *WSNPosition* e um valor numérico associado ao vértice. As arestas são definidas como um conjunto de dois objetos da classe *WSNGraphVertice* (um vértice de entrada e outro de saída, no caso de grafos dirigidos) e um valor numérico associado. Os valores numéricos associados aos vértices e às arestas variam de acordo com a análise que se deseja realizar, podendo ser distância entre os nós da rede nas arestas e disponibilidade energética nos nós, por exemplo.

Um objeto da classe *WSNGraph* é o que contém toda a estrutura de um grafo, isto é, um conjunto de vértices (conjuntos de objetos da classe *WSNGraphVertice*) e um conjunto de arestas (conjunto de objetos da classe *WSNGraphEdge*).

Apesar de ser possível a criação de um grafo inserindo os nós e os vértices em um objeto da classe *WSNGraph* de acordo com a estrutura da rede, esta classe possui o método `createWSNConnectionGraph( WSNSimulator simulator )` que cria um grafo dirigido com os nós da rede como vértices e a possibilidade de comunicação entre os nós como arestas. O que define a possibilidade de comunicação entre os nós é o modelo de propagação de ondas adotado na simulação.

A classe *WSNGraph* também possui o método `findMinimumPaths()` que retorna

os caminhos mínimos entre um nó e outros nós quaisquer da rede, podendo ser considerados os valores atribuídos nos vértices ou nas arestas como os valores utilizados na contabilidade do custo do caminho. Este método, implementado pelo algoritmo de Dijkstra [COR12], é útil na averiguação do menor caminho entre um nó qualquer e uma das estações rádio base da rede, podendo o custo ser contabilizado pelo menor número de nós envolvidos na comunicação, pela menor distância, pelo caminho que utiliza menos energia ou outro critério. Este método retorna um vetor com objetos da classe *WSNGraphPath* com o(s) caminho(s) de menor custo, de acordo com o critério utilizado.

Como os grafos são mais bem entendidos de forma visual, exemplos da sua utilização na análise da rede serão abordados no item 5.6, que apresenta a interface gráfica do WSN-BaSS.

5.6 – A interface gráfica do WSN-BaSS

O estudo da estrutura e do comportamento das Redes de Sensores Sem Fio não é uma tarefa trivial de ser realizada utilizando apenas a programação e a análise dos relatórios produzidos pelas simulações. Uma ferramenta computacional gráfica que auxilie na criação da rede, visualização da posição e relação entre os nós, acompanhamento visual da simulação durante a sua execução, além da possibilidade de realizar análises visuais e produzir relatórios gráficos, promove uma maior facilidade e agilidade no estudo das WSN.

O pacote `br.eng.rsalustiano.wsn.gui` possui dez classes que criam uma interface gráfica para o usuário trabalhar no desenvolvimento e análise das redes desenvolvidas na arquitetura do WSN-BaSS. A Figura 5.15 apresenta o diagrama de classes deste pacote, enquanto a Tabela 5.4 descreve de maneira sucinta qual a função de cada uma das classes na interface gráfica do WSN-BaSS.

A maior parte das classes do pacote gráfico está relacionada com a estrutura de janelas e controle de eventos relacionados à utilização da interface visual. Foge do objetivo deste trabalho a descrição da sua programação e do seu funcionamento, uma vez que são baseadas na API (*Application Programming Interface* – Interface de Programação de Aplicativos) *Swing* do Java [DEI99][SWI13]. A exceção são as interfaces *WSNGUINetworkNode* e *WSNGUIBoardPainter* e a classe abstrata *WSNGUIFrame*, que devem ser conhecidas pelo programador que deseja utilizar interface gráfica do WSN-BaSS.

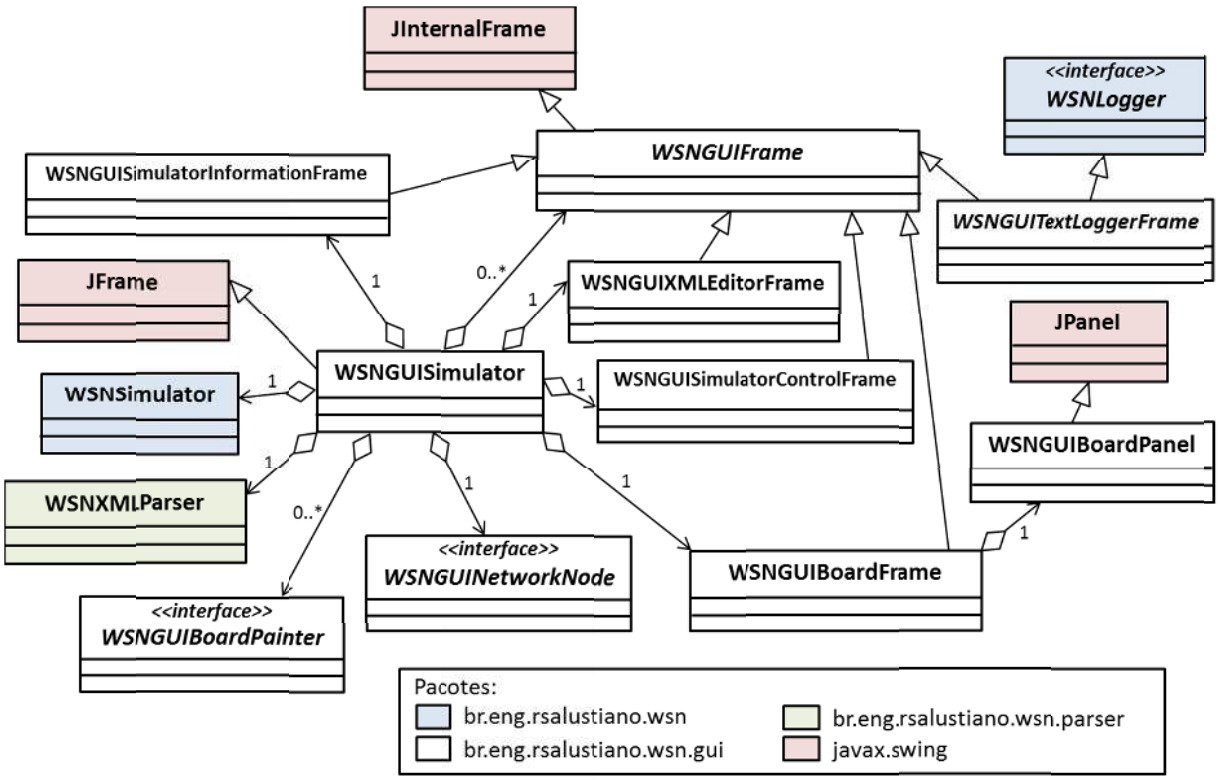


Figura 5.15 – Digrama de classes do pacote *br.eng.rsalustiano.wsn.gui.* Neste diagrama também são exibidas as classes dos pacotes *br.eng.rsalustiano.wsn*, *br.eng.rsalustiano.wsn.parser* e *javax.swing*, cujas classes do pacote *br.eng.rsalustiano.wsn.gui* apresentam dependência direta.

Tabela 5.4 – Classes do pacote br.eng.rsalustiano.wsn.gui

Classe	Descrição
WSNGUIBoardFrame	Subclasse de *WSNGUIFrame* que cria uma janela interna no programa gráfico contendo um painel da classe *WSNGUIBoardPanel* no qual são exibidas informações gráficas da rede.
WSNGUIBoardPainter	Interface que permite que qualquer classe da simulação que a implementa "desenhe" no painel *WSNGUIBoardPanel*, componente interno da janela definida pela classe *WSNGUIBoardFrame*.
WSNGUIBoardPanel	Painel que realiza o "desenho" da rede. Trata-se de um CAD (*Computer Aided Design*) bem simples que permite a manipulação dos nós da rede e dos obstáculos, movendo-os no espaço bidimensional e/ou selecionando-os para visualizar e editar seus dados. Também é possível inserir e remover os nós sensores e os obstáculos da simulação através desse painel. As classes da simulação que implementam a interface *WSNGUIBoardPainter* podem "desenhar" neste painel.
WSNGUIFrame	Os objetos criados a partir de subclasses da classe abstrata *WSNGUIFrame* são inseridos como uma janela interna na janela principal da interface gráfica do simulador WSN-BaSS (*WSNSimulator*).

WSNGUINetworkNode	Interface que deve ser implementada por todo objeto *WSNNetworkNode* (*WSNSensorNode* e *WSNBaseStation*) para que ele possa ser exibido no painel *WSNGUIBoardPanel*. Devem ser definidos o formato representativo (círculo, quadrado, losango, etc.) e a cor dos nós da rede. Informações sobre os nós da rede também podem ser implementadas nessa interface para serem exibidas no painel.
WSNGUISimulator	Classe principal da interface gráfica do WSN-BaSS. Trata-se de uma classe executável que produz a janela principal do programa de simulação.
WSNGUISimulatorControlFrame	Subclasse de *WSNGUIFrame* que cria uma janela interna no programa gráfico contendo os botões de controle da simulação (início, parar, pausa) e as informações do estado da simulação e os tempos de simulação e *wallclock*.
WSNGUISimulatorInformationFrame	Subclasse de *WSNGUIFrame* que cria uma janela interna no programa gráfico contendo as informações gerais da simulação, tais como: número e a que classe pertencem os nós da rede (*WSNNetworkNode*), os obstáculos (*WSNObstacle*), os registros (*WSNLogger*) e as variáveis (*WSNVariable*).
WSNGUITextLoggerFrame	Subclasse abstrata de *WSNGUIFrame* que implementa *WSNLogger*. É uma janela interna do programa gráfico do WSN-BaSS que recebe e exibe cadeia de caracteres como forma de registro (*log*) dos eventos da simulação. Na implementação de uma subclasse desta classe, o programador deve decidir qual a cadeia de caracteres que deve ser registrada para cada evento de registro. Um botão presente na janela interna criada por essa classe permite que o usuário do simulador salve o conteúdo registrado em um arquivo do tipo texto.
WSNGUIXMLEditorFrame	Subclasse de *WSNGUIFrame*. É uma janela interna do programa gráfico do WSN-BaSS que permite a visualização e edição do arquivo XML que define os parâmetros da simulação e os nós da rede.

A inicialização da interface gráfica do WSN-BaSS é feita pela execução da classe *WSNGUISimulator* pertencente ao pacote `br.eng.rsalustiano.wsn.gui`. Esta classe possui o método estático `main()` que carrega o programa e gera os componentes visuais da interface gráfica. A aparência da tela principal do programa pode ser analisada na Figura 5.16, na qual também há uma descrição das principais funcionalidades do programa.

Para se carregar uma simulação no programa, deve-se utilizar um arquivo com um documento XML que descreva a rede, como apresentado no item 5.4.2. Alterações realizadas na rede com as ferramentas de edição da interface gráfica podem ser atualizadas no arquivo XML aberto ou pode-se criar um novo arquivo (“Salvar como...”) com as modificações.

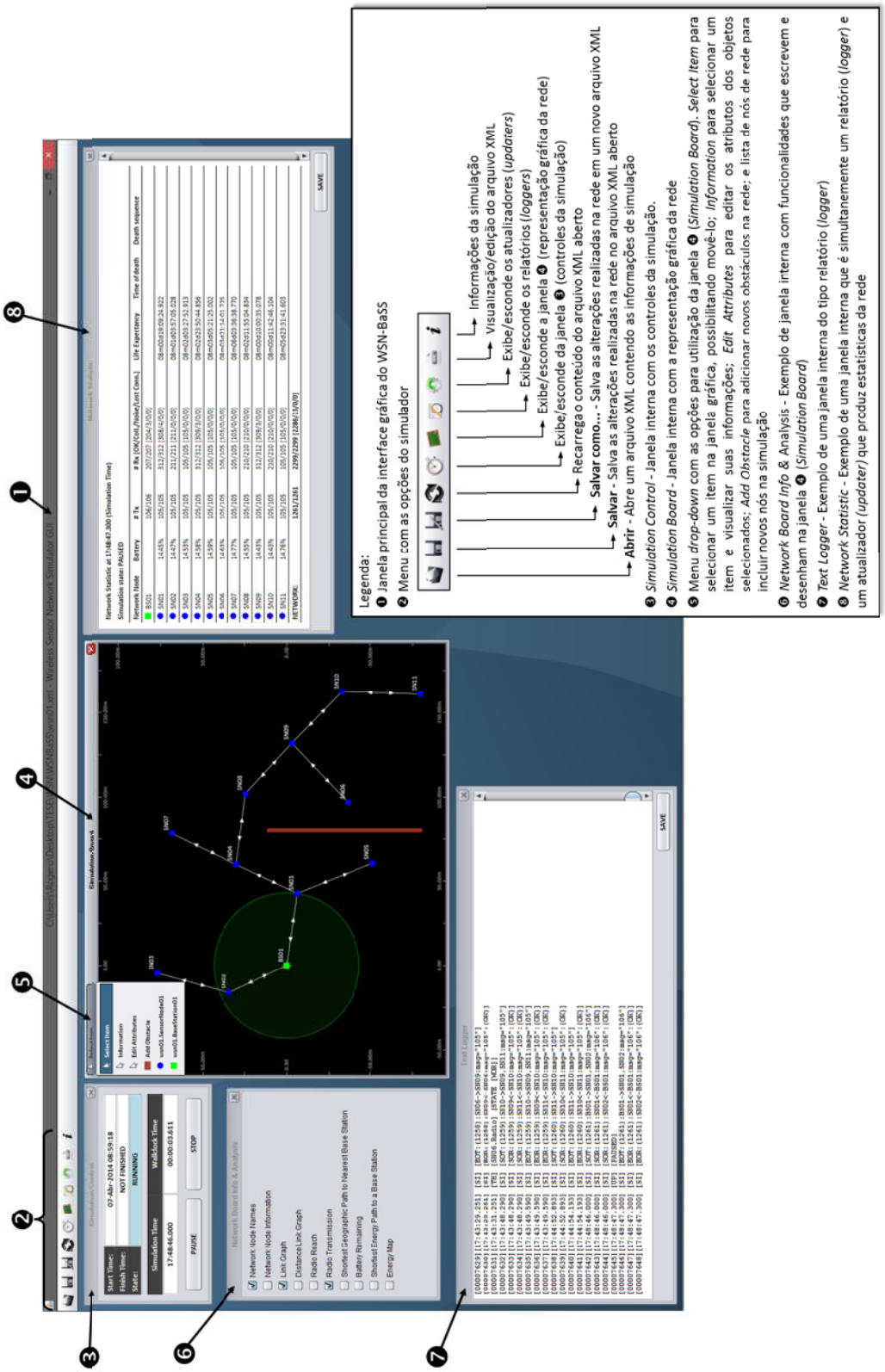


Figura 5.16 – Interface gráfica do WSN-BaSS.

Do ponto de vista da simulação, a definição de um nó de rede deve ser feita através de uma subclasse de *WSNSensorNode* ou de *WSNBaseStation*. Contudo, a interface gráfica necessita de uma forma visual para poder representar esse nó, o que deve ser feito pela implementação da interface *WSNGUINetworkNode*. Os métodos a serem implementados são: `getShape()`, `getColor()`, `getInformation()` e `getReport()`.

O método `getShape()` deve retornar a forma geométrica representativa do nó, cujas possibilidades são definidas na classe do tipo enumeração *WSNGUIShape*, definida internamente à *WSNGUINetworkNode*: CIRCLE (círculo), SQUARE (quadrado), TRIANGLE_UP (triângulo com um dos vértices para cima), TRIANGLE_DOWN (triângulo com um dos vértices para baixo) ou DIAMOND (quadrado com 45° de rotação). O método `getColor()` deve retornar a cor que a forma geométrica deve assumir.

Os métodos `getInformation()` e `getReport()` podem ser utilizados para indicar algumas informações relacionadas ao nó que podem ser úteis para o usuário que esteja trabalhando com o simulador. O primeiro método deve conter uma informação mais reduzida (possivelmente para ser visualizada junto com a representação geométrica do nó) e o segundo, um conjunto maior de informações.

Estes quatro métodos da interface *WSNGUINetworkNode* são chamados toda vez que a interface gráfica do simulador necessitar representar o nó da rede. Pode-se, portanto, variar os seus valores de retorno ao longo da execução da simulação. Por exemplo, pode-se mudar a cor do nó de acordo com o seu estado no decorrer da simulação.

A interface *WSNGUIBoardPainter* deve ser implementada pelas classes destinadas a fazer algum tipo de desenho na janela gráfica (*Simulation Board*), sendo isso realizado através do método `boardPaint( WSNGUISimulator guiSimulator, WSNGUIBoardPanel board, Graphics2D g2d )`. Este método é chamado em todos os objetos adicionados à simulação que implementam *WSNGUIBoardPainter* toda vez que a janela gráfica é atualizada (repintada).

Os únicos elementos “desenhados” automaticamente no *Simulation Board* são os nós sensores (com as formas e cores definidas pelo programador pela implementação da interface *WSNGUINetworkNode*), os obstáculos, as coordenadas e a origem, isto é, a posição (0,0). Qualquer outro tipo de traço ou informação adicional presente na janela gráfica é resultado da execução do método `boardPaint()` de algum objeto que implementa a interface *WSNGUIBoardPainter*.

A Figura 5.17 apresenta um exemplo de implementação do método `boardPaint()`, no qual um grafo de distância entre os nós que podem se comunicar é criado. Os nomes dos

nós são atribuídos aos vértices e a distância entre eles são colocadas nas arestas. A Figura 5.18 ilustra o *Simulation Board* resultante da execução do método `boardPaint()` apresentado na Figura 5.17.

```
01  public void boardPaint( WSNGUISimulator guiSimulator, WSNGUIBoardPanel board,
                            Graphics2D g2d ) {
02      WSNGraph graph = new WSNGraph();
03      BasicStroke bStroke = new BasicStroke( (float) ( 0.5 / board.getScale() ) );
04      Font gFont = new Font( "Calibri", Font.PLAIN, 11 );
05
06      graph.createWSNConnectionGraph( guiSimulator.getSimulator() );
07
08      for ( WSNGraphVertice vertice: graph.getVertices() )
09          vertice.setInformation( vertice.getNetworkNode().getName() );
10
11      for ( WSNGraphEdge edge: graph.getEdges() ) {
12          edge.setValue( edge.getInputVertice().getPosition().distance(
                                                  edge.getOutputVertice().getPosition() ) );
13          edge.setInformation( String.format( Locale.US, "%.2fm", edge.getValue() ) );
14      }
15
16      board.drawGraphEdges( graph, Color.WHITE, bStroke, false );
17      board.drawGraphEdgeInformation( graph, Color.WHITE, gFont );
18      board.drawGraphVerticeInformation( graph, Color.WHITE, gFont );
19  }
```

Figura 5.17 – Exemplo da implementação do método *boardPaint()* da interface *WSNGUIBoardPainter*.

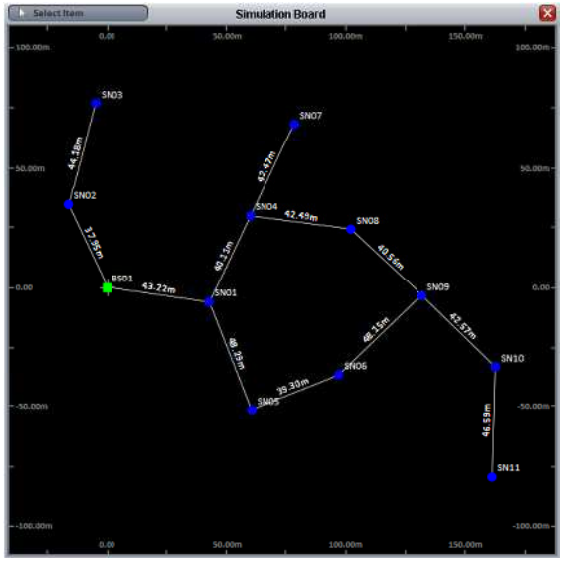


Figura 5.18 – Resultado da execução do método *boardPaint()* descrito na Figura 5.17.

A Figura 5.19 apresenta outros seis resultados de implementações do método `boardPaint()`. As implementações que necessitam a seleção de um ou mais nós devem utilizar o método `board.getSelectedNetworkNodes()` para identificar quais nós foram selecionados pelo usuário na interface gráfica do *Simulation Board*.

A classe abstrata *WSNGUIFrame* (subclasse da *JInternalFrame* pertencente ao pacote `javax.swing` [SWI13]) é utilizada como referência para qualquer janela interna que necessite ser inserida no simulador. Toda classe que estender *WSNGUIFrame* tem acesso ao

objeto *WSNGUISimulator* através do método `getGUISimulator()` e necessita implementar o método abstrato `updateFrame()`, o qual é chamado toda vez que o tempo de simulação avançar.

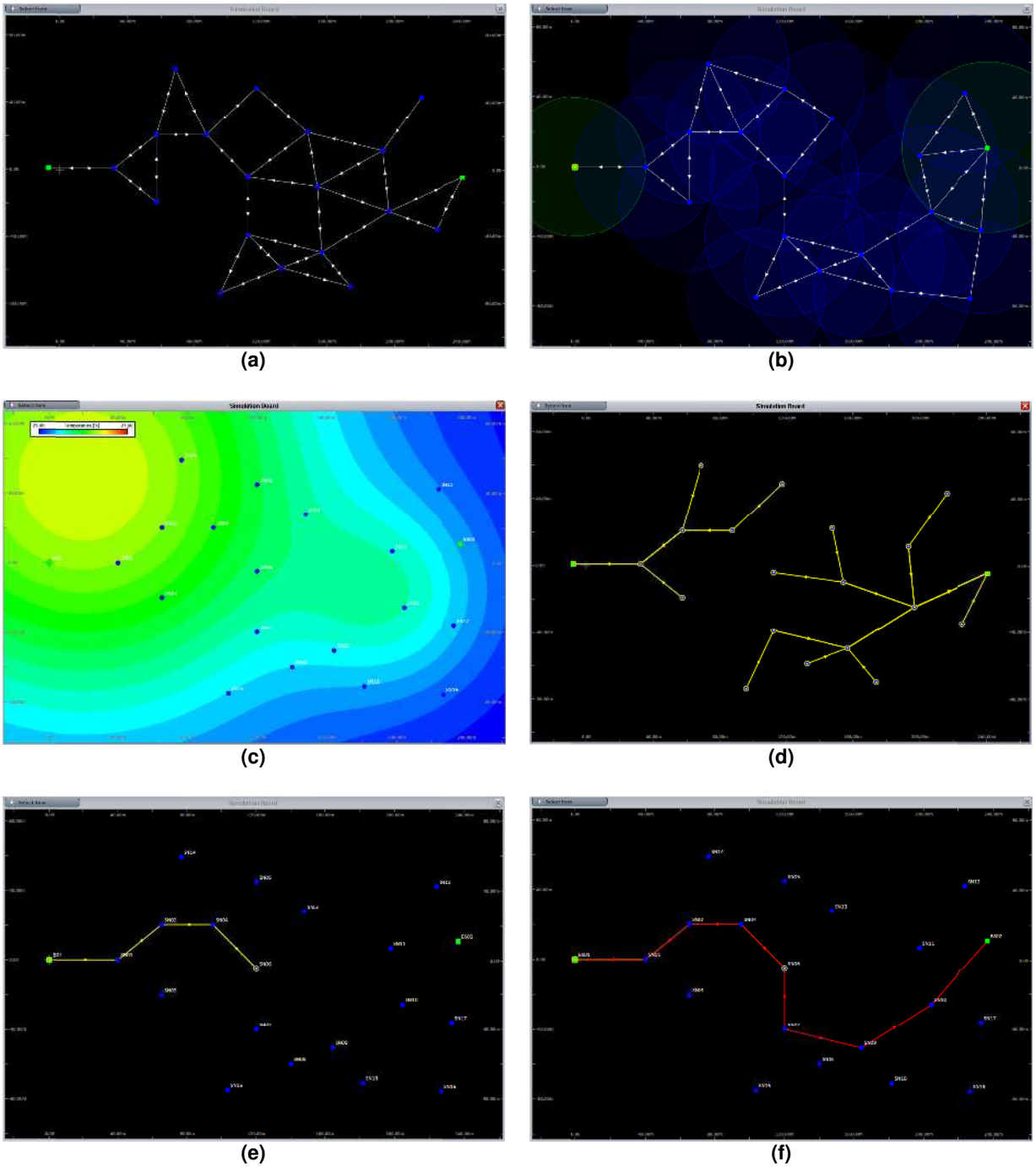


Figura 5.19 – Exemplos de resultados obtidos a partir de implementações do método *boardPaint()*. Os círculos azuis são nós sensores e os quadrados verdes são estações rádio base. (a) Grafo de conexão entre os nós da rede. (b) Alcance dos rádios. (c) Mapa de distribuição da variável temperatura pelo ambiente. (c) Menor distância geográfica de todos os nós sensores até a estação rádio base mais próxima (todos os nós foram selecionados). (e) Menor distância geográfica do nó sensor SN06 (nó selecionado) até a estação rádio base mais próxima. (f) Caminho de consumo mínimo de energia para transmissão do nó SN6 (selecionado) até uma estação rádio base (no caso, dois caminhos são possíveis).

Essas janelas internas são uteis para o desenvolvimento de relatórios (*loggers*) e atualizadores (*updaters*) que apresentem elementos visuais, como tabelas e gráficos, além de controles (exibe/esconde) para a exibição de informações na janela *Network Board*.

A Figura 5.16 apresenta três janelas internas adicionadas ao simulador: *Network Board Info & Analysis* (Informações e Análises Gráficas da Rede), *Text Logger* (Relatório Textual) e *Network Statistic* (Estatística da Rede). Essas janelas foram desenvolvidas em classes que estendem *WSNGUIFrame*, mas além disso, a primeira implementa *WSNUpdater* e *WSNGUIBoardPainter*, a segunda implementa *WSNLogger*, e finalmente a terceira implementa *WSNLogger* e *WSNUpdater*. Cada objeto dessas classes foi adicionado à simulação através do documento XML como um `<UPDATER>`, `<LOGGER>` ou `<GENERIC>`, de acordo com a sua(s) funcionalidade(s). Contudo, como todas estendem *WSNGUIFrame*, automaticamente suas janelas internas foram inseridas dentro da janela principal da interface gráfica do WSN-BaSS.

Em suma, a criação de ferramentas de análise visual para o acompanhamento das simulações de Redes de Sensores Sem Fio pode utilizar a classe abstrata *WSNGUIFrame* para o desenvolvimento de novas janelas com objetos visuais próprios ou implementar a interface *WSNGUIBoardPainter*, utilizando o *Network Board* como “pano de fundo”. O tipo de relação que a classe desenvolvida possui com os eventos da simulação determina se a classe de análise desenvolvida deve implementar *WSNUpdater*, *WSNLogger*, *WSNThread* ou uma combinação dessas interfaces.

5.7 – Testes e aplicação do WSN-BaSS

O desenvolvimento da estrutura do WSN-BaSS foi acompanhado de diversos testes, nos quais cada funcionalidade implementada foi testada isoladamente e em conjunto com o todo do simulador.

Os principais testes estão relacionados ao controle da simulação e às situações que podem ocorrer durante a transmissão das mensagens (Tabela 5.2). Primeiramente, todas as situações foram programadas e verificadas de forma isolada em cenários com poucos nós. Num segundo momento, redes com quantidades maiores de nós (de dez a vinte nós) distribuídos aleatoriamente foram submetidas à simulação (transmissão aleatória de mensagens) durante um minuto de tempo de simulação e os *loggers* das transmissões foram validados manualmente.

Considerando a complexidade e extensão dos testes, não cabe aqui o registro dos mesmos, uma vez que a melhor maneira de testar um software é aplicá-lo a um determinado problema, comprovando sua praticidade de uso, desempenho e coerência nos resultados.

Contudo, um teste de desempenho do WSN-BaSS será relatado com um cenário de distribuição aleatória dos nós da rede no ambiente e com seus parâmetros de funcionamento extraídos das folhas de dados de componentes eletrônicos.

5.7.1 – Avaliação de desempenho do WSN-BaSS

Com o intuito de avaliar o desempenho do WSN-BaSS e ao mesmo tempo obter algumas informações sobre as transmissões realizadas pelas redes simuladas, foram criados dois cenários (Simulação A e Simulação B) com distribuição aleatória de nós sensores, conforme apresentado na Figura 5.20 e na Figura 5.21 (captura de tela do *Simulation Board* da simulação). As características dos nós sensores, estações rádio base e variáveis de ambiente são as mesmas para os dois cenários de simulação.

O ambiente possui uma variável que representa a temperatura, cujo valor varia de 25 a 27°C de acordo com posição do ambiente e o instante da simulação. Trata-se, portanto de uma variável dinâmica. Os nós da rede são dispositivos com processamento baseado no microcontrolador ATMega64L [ATM13] e possuem rádio com as características do transceptor CC2500 [CC_09]. Os nós sensores possuem um sensor de temperatura digital baseado nas características do SHT11 [SHT11] e uma bateria CR2032 [CR_14]. A Tabela 5.5 apresenta as características desses componentes utilizadas na modelagem da simulação.

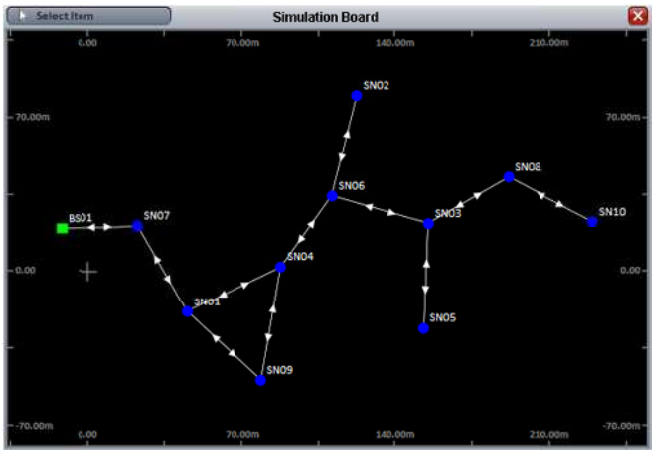


Figura 5.20 – Simulação A. Distribuição espacial dos dez nós sensores e da estação rádio base.

Os nós sensores foram programados para executarem a leitura da temperatura de 15 em 15 minutos e caso a temperatura for menor que 26,0°C, uma mensagem deve ser enviada contendo o identificador do nó, um número identificador da mensagem (número com contagem crescente) e a temperatura. Ao receber uma mensagem, um nó sensor deverá retransmitir a mensagem caso ainda não a tenha recebido. Se a mensagem já foi recebida, nada deverá ser feito. Cada nó armazena apenas o identificador do nó e o identificador da

última mensagem recebida de cada nó. Assim, se o identificador da mensagem recebida for maior que o último identificador da mensagem registrada para o nó que enviou a mensagem, ela deve ser retransmitida. A estação rádio base deve apenas receber a mensagem e registrá-la.

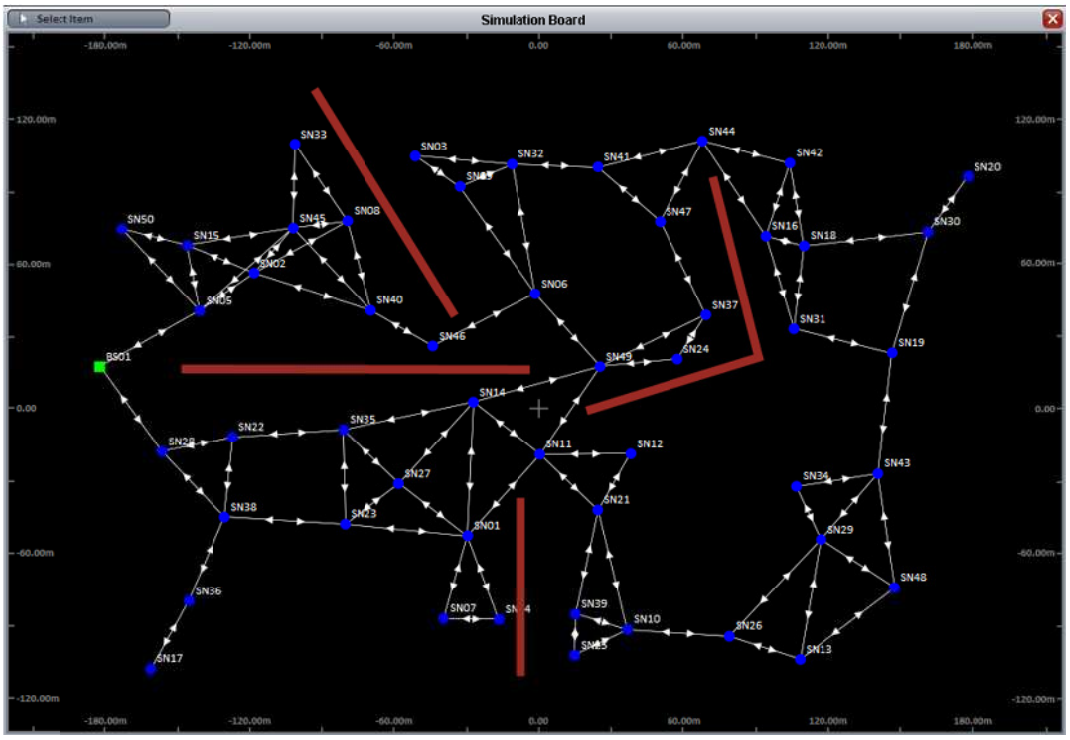


Figura 5.21 – Simulação B. Distribuição espacial dos 50 nós sensores e da estação rádio base.

As mensagens enviadas foram consideradas de tamanho fixo de 32 Bytes, que somando a 8 Bytes de preâmbulo, 4 Bytes de bits de sincronização e 4 Bytes de CRC, resultam em 384 bits. A uma taxa de 250 kbps, o tempo de transmissão de uma mensagem é de 0,001536 segundos.

O Apêndice D apresenta a programação das classes envolvidas nesse teste de simulação e os resultados em termos de quantidade de mensagens enviadas, recebidas e que sofreram algum tipo de interferência para cada nó. O tempo operacional de cada nó sensor e sequência de "morte" dos nós também foram registrados. Nesse apêndice também são mostradas figuras com as conexões de rede em determinados momentos da Simulação B.

Cada simulação foi executada duas vezes, uma utilizando a interface gráfica e a outra sem. O computador utilizado foi um *notebook* com processador Intel i7-3520M com dois núcleos de processamento capazes que executar 4 *thread* simultaneamente pelo mecanismo de *Hyper-Threading* (2,9GHz, *cache* L1 128kB x2, *cache* L2 256kB x2 e *cache* L3 4MB) e 6GB de memória RAM. Os tempos necessários para execução da simulação (*wallclock time*) e o tempo de vida da rede são apresentados na Tabela 5.6.

Tabela 5.5 – Características dos componentes utilizados como referência nas simulações

Componente	Descrição	Característica	Valor utilizado
ATMega64L [ATM13]	Microcontrolador	Frequência de operação (0 a 8 MHz) Tensão de alimentação (2,7 a 5,5 V) Temperatura de referência (-55 a 125 °C) Consumo: modo ACTIVE (5,0 µA @ 4 MHz) modo IDLE (2,5 µA @ 4 MHz)	4 MHz 3,0 V 25°C 5,0 µA 2,5 µA
CC2500 [CC_09]	Rádio transceptor	Frequência de operação (26 ou 27 MHz) Frequência de transmissão (2,4 GHz) Taxa de transmissão (1,2 a 500 kBaud)[3] Tensão de alimentação (1,8 a 3,6 V) Potência de transmissão (+1 a -55 dBm) Sensibilidade (-83 dBm @ 500kBaud) Consumo: modo TX (11,1 mA @ -12 dBm) modo RX (13,3 a 16,6 mA) modo WOR (900nA)	26 MHz 2,4 GHz 250 kbps 3,0V -12 dBm -83 dBm 11,1 mA 15,0 mA 900 nA
SHT11 [SHT11]	Sensor digital de temperatura e umidade	Tempo para leitura da temperatura (320 ms) Tensão de alimentação (2,4 a 5,5 V) Consumo: modo SLEEP (1,5 µA) modo MEASURING (1,0 mA)	320 ms 3,0 V 1,5 µA 1,0 mA
CR2032 [CR_14]	Bateria	Tensão (3,0 V) Capacidade (240 mAh) Auto descarregamento (<2% ao ano)	3,0 V 240 mAh 1,5% ao ano

Tabela 5.6 – Resultados temporais das simulações

Simulação	Descrição	Tempo de vida da rede	Tempo de execução da simulação	
			Com Interface Gráfica	Sem Interface Gráfica
Simulação A (Figura 5.20)	1 estação rádio base 10 nós sensores	8 anos, 05 meses, 06 dias, 18 h, 51 min e 41.636 s	03:21:16.894	01:40:10.984
Simulação B (Figura 5.21)	1 estação rádio base 50 nós sensores	9 anos, 07 meses, 17 dias, 20 h, 59 min e 43.783 s	07:42:32.328	06:54:34.745

O tempo de execução das simulações está coerente com o número total de *BaSSThreads* envolvidas na simulação, sendo menor na Simulação A, que possui 33 *BaSSThreads,* se comparado com a Simulação B, que possui 153.

A utilização da interface gráfica teve grande impacto na Simulação A, uma vez que

[3] A taxa em kbps é considerada a metade da taxa em kBaud [CC_09]

a presença dela fez com que o tempo de simulação fosse 2,01 vezes mais lento que a execução sem a utilização da interface. Já na simulação B, a utilização da interface gráfica teve um impacto menor, pois a sua presença fez com que a simulação fosse 1,12 vezes mais lenta que na ausência de interface.

O tempo de vida da rede em ambas as simulações (falência do último nó) está coerente com a atividade que os nós desempenham. Na maior parte do tempo os nós permanecem no estado de baixo consumo (nó sensor no estado IDLE, rádio no estado WOR e sensor no estado SLEEP), consumindo 0,0025019 A. De 15 em 15 minutos ocorrem eventos de transmissão que duram 0,001536 segundos cada, sendo a quantidade de transmissões variável para cada uma das simulações e o tempo de vida de cada nó de forma particular.

Com uma bateria de 240 mAh, se os nós permanecessem com este consumo o tempo inteiro eles durariam 10,95 anos. O tempo de vida do último nó da rede com aproximadamente 8,42 anos para a Simulação A e 9,58 anos para a Simulação B estão coerentes.

Muitas outras conclusões podem ser obtidas pela análise dos resultados da Simulação A e da Simulação B, como o impacto gerado no consumo energético nos nós sensores a partir da sequência de falência operacional dos nós da rede ou em que momento nenhum dado mais chega à estação rádio base. Mas o objetivo dessas simulações, no contexto dessa tese, é verificar o desempenho do simulador, isto é, contabilizar o tempo necessário para a execução das simulações e a validade dos resultados obtidos em termos de tempo de vida da rede.

5.7.2 – Aplicação do WSN-BaSS ao problema de alocação de baterias

A principal aplicação do WSN-BaSS foi realizada no trabalho de mestrado de Felipe Antonio Moura Miranda, intitulado “Proposta de alocação de baterias em redes de sensores sem fio orientada à maximização do tempo de vida” [MIR11].

O simulador de Redes de Sensores Sem Fio, ainda em desenvolvimento, sem o nome de WSN-BaSS e com uma estrutura de classes ainda pouco definida, foi utilizado como ferramenta auxiliar no trabalho de pesquisa supracitado. A motivação para a utilização do simulador em desenvolvimento, em detrimento de outros já conhecidos, foi justificada pela *“[...] liberdade para implementação e troca de novos componentes; possibilidade de implementação de protocolos e observação direta do consumo de energia de cada mote de uma RSSF. Além disso, a possibilidade de modificar diretamente seu código-fonte, elaborado na linguagem de programação Java, foi fundamental para a implementação de novas classes e métodos úteis para o trabalho, tais como: protocolos próprios; motes e componentes diversos; observação de*

diversos parâmetros durante as simulações etc" [MIR11].

Grande parte dos problemas de programação e estruturação do núcleo de simulação e do simulador de Redes de Sensores Sem Fio foi levantado no projeto de alocação de baterias para os nós da rede. O resultado da utilização das versões preliminares do simulador nesse projeto foi satisfatório, motivando a sua formalização conceitual, a ampliação das suas funcionalidades e a sua implementação de forma mais estruturada.

Resumo

Este capítulo descreveu a arquitetura funcional e a implementação do WSN-BaSS, simulador destinado ao estudo de Redes de Sensores Sem Fio.

O esquema de troca de mensagens entre os nós da rede foi explicado de forma detalhada e os nós, rádios, sensores e variáveis de ambiente foram descritos em termos estruturais e funcionais.

Uma vez que o objetivo do capítulo não é ser um manual de utilização do WSN-BaSS, não foi apresentado um detalhamento de utilização de todas as classes do simulador de forma específica, mas um panorama geral das suas funções dentro do simulador foi apresentado.

Com a estrutura de programação suficientemente flexível para a implementação das camadas de rede e dos protocolos de comunicação, o WSN-BaSS permite que os protocolos apresentados no capítulo 2 sejam facilmente incorporados à sua estrutura. A programação orientada a objetos permite a elaboração de classes dedicadas para esses protocolos, as quais podem ser reutilizadas em diferentes projetos.

Da mesma maneira, componentes de hardware podem ser modelados de acordo com a necessidade do projeto. Microcontroladores, sensores, baterias, rádios, antenas e outros componentes eletrônicos podem ser descritos em termos de funcionamento e consumo de energia.

Um exemplo de rede implementada (programação no Apêndice D) e simulada em dois cenários distintos foi apresentado, avaliando, assim, o desempenho do WSN-BaSS.

6
Conclusões

"Eu não chuto. Como cientista eu chego a conclusões baseadas em observações e experimentação".
Sheldon Cooper

O conhecimento produzido nesta tese abrange tanto a teoria como a prática. O aperfeiçoamento dos mecanismos de sincronização por barreiras e a modelagem de dados referente à representação de uma rede em ambiente computacional são as principais contribuições teóricas. A implementação na linguagem de programação Java de mecanismos para o controle e sincronização de *threads*, a produção de um *framework* para o desenvolvimento de simuladores (BaSS – *Barrier Synchronization Simulator*) e de um simulador para Redes de Sensores Sem Fio (WSN-BaSS) são as contribuições práticas do trabalho.

A contribuição mais relevante deste trabalho é a incorporação das interrupções e dos temporizadores no mecanismo de sincronização por barreiras, permitindo, assim, uma melhor descrição dos elementos simulados. A sincronização por barreiras clássica não permitiria uma avaliação mais detalhada do comportamento de componentes de hardware e software ainda em desenvolvimento.

A tese apresentada é resultado de um conjunto de diversos aspectos da pesquisa científica. (1) A revisão bibliográfica criou um panorama das definições relacionadas às Redes de Sensores Sem Fio e levantou os principais problemas e desafios encontrados no estudo e

desenvolvimento dessas redes. (2) O estudo dos mecanismos de sincronização serviu de base para a proposição de um modelo de sincronização por barreiras que atendesse às características de modelamento desejáveis para a criação de um *framework* destinado ao desenvolvimento de simuladores. (3) O entendimento da arquitetura da linguagem de programação Java deu suporte ao mecanismo de bloqueio de *threads* e, em conjunto com a Engenharia de Software, permitiu o modelamento da estrutura de classes que compõem as arquiteturas de software desenvolvidas. (4) O estudo dos grafos e dos algoritmos que os manipulam deram suporte para a implementação de ferramentas para manipulação e análise da rede. (5) A compreensão das classes de manipulação gráfica da linguagem Java permitiu o desenvolvimento do ambiente de computação gráfica que permitir a análise da rede de forma visual. (6) A compreensão do funcionamento de dispositivos eletrônicos (microcontroladores, rádios, sensores e baterias) e a utilização dos mesmos em pequenos projetos durante o desenvolvimento da tese foi de extrema importância na modelagem funcional do simulador e como base para os parâmetros utilizados na simulação.

O *framework* BaSS mostrou-se bastante versátil para o desenvolvimento de simuladores determinísticos que necessitam de um controle apurado da evolução temporal e ao mesmo tempo permita a sincronização de objetos independentes e a interação entre eles.

O WSN-BaSS apresentou resultados satisfatórios nas simulações executadas. As funcionalidades providas pelo BaSS, principalmente os mecanismos de interrupção, permitiram uma flexibilidade na modelagem funcional dos nós da rede, conforme mostrado no capítulo 5. O controle centralizado da troca de mensagens entre os nós mostrou-se eficiente na previsão das possíveis interferências que podem ocorrer durante as transmissões.

O ambiente gráfico desenvolvido para o WSN-BaSS foi de fundamental importância para um melhor entendimento dos eventos que acontecem na rede e a relação existe entre eles. Somente a estrutura organizacional e funcional do simulador não seria suficiente tanto para a programação adequada do simulador, como para a interpretação de ocorrências da rede durante as simulações. Acredita-se que a interface gráfica será uma ferramenta de grande valia para os desenvolvedores que utilizarão o simulador, principalmente pelo fato da simulação poder ser acompanhada visualmente enquanto está sendo executada (um dos principais diferenciais do WSN-BaSS em relação a outros simuladores de Rede de Sensores Sem Fio).

Apesar de não ser uma regra geral, a interface gráfica é útil para a visualização de alguns eventos que ocorrem na rede e para a compreensão do funcionamento de alguns protocolos de comunicação, inclusive para a verificação se os mesmos estão programados corretamente. No estudo do tempo de vida da rede, entretanto, pode não ser necessário (ou

mesmo não fazer sentido) o acompanhamento da simulação de forma gráfica, uma vez que ela pode levar horas para finalizar.

A arquitetura do simulador foi projetada para a execução de simulações determinísticas e que envolvem um conjunto de interações entre os objetos que não podem ser equacionadas, isto é, que não podem ser previstas por leis matemáticas. Os objetos simulados são autônomos, agindo e reagindo conforme a sua programação e a interação com outros objetos. Essas características não permitem um aumento na velocidade de execução das simulações, sendo o detalhamento descritivo dos objetos e a frequência de execução do computador utilizado na simulação os fatores limitantes.

No caso das simulações executadas no WSN-BaSS, deve-se considerar, também, que o tempo que está sendo simulado é da ordem de meses e anos (a depender da bateria utilizada e do consumo energético dos nós) e o detalhamento operacional dos nós tem impacto direto no tempo necessário à simulação.

Há muitas extensões e melhorias possíveis e já previstas para essa tese. Segue uma relação das principais delas.

Desenvolvimento de novos simuladores: A utilização do *framework* BaSS como núcleo de processamento em outros simuladores é prevista. Toda a modelagem e implementação do *framework* foi realizada tendo em mente a sua possível aplicação em outros simuladores. Por isso, sua arquitetura organizacional é flexível e não há restrições para o seu uso em outros projetos de simulação.

Inclusão de novos protocolos de comunicação: O WSN-BaSS ainda necessita de muitas extensões para poder ser utilizado de forma adequada e prática. Apenas o cerne do simulador foi desenvolvido e há a necessidade de se implementar os diversos protocolos apresentados no capítulo 2 para aumentar a sua usabilidade.

Inclusão de novos dispositivos: Dispositivos eletrônicos (processadores, sensores e baterias) devem ser mapeados na arquitetura do WSN-BaSS com o intuito de ampliar as possibilidades composicionais dos nós sensores e estações rádio base. As classes que implementarão esses dispositivos devem apresentar as características de funcionamento e consumo energético dos dispositivos reais.

Confronto com o mundo real: Um simulador é útil na realização de estimativas do comportamento de sistemas do mundo real e a sua calibração é importante e deve ser feita tendo como base o sistema real que ele representa. Por se tratar de um simulador novo, o WSN-BaSS ainda precisa de muito aperfeiçoamento e este pode

ser realizado através de uma análise comparativa com os resultados produzidos em redes experimentais reais.

Estudo de novos protocolos de comunicação: O principal objetivo do WSN-BaSS é auxiliar no estudo das Redes de Sensores Sem Fio. Por isso, ele deve ser utilizado como ferramenta auxiliar no desenvolvimento de novos protocolos de comunicação e na análise do desempenho deles.

Estudo sistemático das WSN: A comparação da eficiência de uma Rede de Sensor Sem Fio com outras nem sempre é realizada de forma adequada, utilizando parâmetros muitas vezes equivocados. A estrutura do WSN-BaSS pode auxiliar na criação de uma ferramenta métrica de avaliação das Redes de Sensores Sem Fio, sistematizando o procedimento. Tal ferramenta pode ser distribuída e incorporada ao simulador, facilitando o processo comparativo entre diferentes redes implementadas.

Estudo de distribuição dos nós: Topologias de rede padronizadas e com características específicas podem ser utilizadas como padrão para a execução de testes e avaliação de protocolos de comunicação. A interface gráfica do BaSS pode ser utilizada como geradora desses padrões e o documento XML produzido pelo WSN-BaSS (capitulo 5) pode ser utilizado como forma de registro dessas distribuições dos nós.

Ferramenta didática para o estudo de sincronização de processos: O *framework* BaSS pode ser utilizado como ferramenta didática no estudo da sincronização de processos, uma vez que já se encontra desenvolvido, testado e possui uma razoável documentação. Tanto a sua teoria como a sua utilização no ensino de sincronização de processos, para os níveis de graduação e pós-graduação, pode ser feita como auxílio didático e eventualmente utilizada em projetos de disciplinas aplicadas.

Extensão para outros tipos de redes: A estrutura do WSN-BaSS pode ser utilizada como parte de um projeto de simulação mais abrange, no qual seria possível a inclusão de outros tipos de redes no simulador, realizar estudos de interligação de redes e incluir objetos em rede, prevendo o comportamento e vislumbrando possibilidades para a Internet das Coisas.

Apesar do WSN-BaSS ainda necessitar de mais testes e precisar ser aplicado em outros projetos de pesquisa para poder ser validado de forma mais apurada, a sua avaliação inicial como ferramenta de simulação destinada ao estudo das Redes de Sensores Sem Fio foi satisfatória em termos de flexibilidade estrutural e eficiência de execução das simulações.

A utilidade de um simulador para Redes de Sensores Sem Fio é inquestionável diante as dificuldades de execução de experimentos com essas redes no mundo real. Este trabalho de doutorado, apesar da sua singeleza, conseguiu reunir as principais características desejáveis em um simulador dessa natureza, propôs um mecanismo de simulação sincronizada e apresentou as implementações do mecanismo de sincronização e do simulador na linguagem de programação Java.

Referências Bibliográficas

[ALC10] ALCARAZ, P. et al.. **Wireless Sensor Networks and the Internet of Things: Do We Need a Complete Integration?**. In Proceedings of the 1st International Workshop on the Security of The Internet of Things (SecIoT 2010). Tokyo, Japão, 2010.

[ALK04] AL-KARAKI, J. N.; KAMAL, A. E.. A Taxonomy of Routing Techniques in Wireless Sensor Networks. In: MAHGOUB, I; MOHAMMAD, I.. **Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems**. London: CRC Press, 2005. Cap. 6, pp. 6.1-6.22.

[ALM04] ALMEIDA, V. C.. **Sistema Operacional YATOS para Redes de Sensores sem Fio**. 2004. Disponível em: <http://www.dcc.ufmg.br/~mmvieira/publications/04wso-yatos.pdf>. Acesso em: 15 fev. 2007.

[AKA05] AKAN, Ö. B.; FELLOW, I. F.. Event-to-Sink Reliable Transport in Wireless Sensor Networks. **IEEE/ACM Transactions on Networking**. out. 2005. pp. 1003-1016.

[AMM03] AMMAR, M. et al.. **Report of the National Science Foundantion Workshop on Fundamental Research in Networking**. 2004. Disponível em: <http://www.cs.virginia.edu/~jorg/workshop1>. Acesso em: 23 abr. 2007.

[ARE89] ARENSTORF, N. S.; JORDAN, H. F.. Comparing barrier algorithms. **Parallel Computing**. Vol. 12. 1989. pp. 157-170.

[ASA98] ASADA, G. et al.. **Wireless Integrated Network Sensors: Low power systems on a chip**. In Proceedings of the 24th European Solid-State Circuits Conference (ESSCIRC). Hague, Netherlands, 1998. pp. 9-16.

[ATM13] ATMEL: AVR 8-bit and 32-bit Microcontroller. Disponível em: <http://www.atmel.com>. Acesso em: 13 mar. 2013.

[AXE86] AXELROD, T. S.. Effects of synchronization barriers on multiprocessor performance. **Parallel Computing**. Vol. 3. 1986. pp. 129-140.

[BAL03] BALL, C.; BULL, M.. **Barrier Synchronization in Java**. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.3867&rep=rep1&type=pdf>. Acesso em: 25 mar. 2011.

[BAL05] BALDWIN, P. et al.. **VisualSense: Visual Modeling for Wireless and Sensor Network Systems**. Technical Memorandum ICBIERL, University of California, Berkeley, 2005.

[BAN05] BANKA, T.; TANDON, G.; JAYASUMANA, A. P. **Zonal Rumor Routing for Wireless Sensor Networks**. In Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC). Las Vegas, USA, 2005. Vol. 2, pp. 562-567.

[BHA01] BHARDWAJ, M; GARNETT, T. CHANDRAKASAN, A. **Upper Bounds on the Lifetime of Sensor Networks**. In Proceedings of the IEEE International Conference on Communications (IEEE ICC). Helsinki, Finland, 2001. Vol. 3, pp. 785-790.

[BHA02] BHARDWAJ, M. CHANDRAKASAN, A. **Bounding the Lifetime of Sensor Networks Via Optimal Role Assignments**. In Proceedings of the 21st IEEE Conference on Computer Communications (IEEE INFOCOM). New York, USA, 2002. vol 3, pp. 1587-1596.

[BHA05] BHATTI, S. et al.. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. **ACM Kluwer Mobile Networks & Applications (MONET) Journal**: **Special Issue on Wireless Sensor Networks**. pp. 563-579, Ago. 2005.

[BHA06] BHARATHIDASAN, A.; PONDURU, V. A. S.. **Sensor networks: an overview**. Disponível em: <wwwcsif.cs.ucdavis.edu/~bharathi/sensor/survey.pdf>. Acesso em: 20 mai 2007.

[BRA02] BRAGINSKY, D.. **Rumor routing algorthim for sensor networks**. In Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA). Atlanta, USA, 2002. pp. 22-31.

[BOK04] BOKAREVA, T.; BULUSU, N.; JHA, S.. **A Performance Comparison of Data Dissemination Protocols for Wireless Sensor Networks**. In Proceedings of the Global Telecommunications Conference Worklshops (Globecom). 2004. pp. 85-89.

[BIL03] BILSTRUP, U. et al.. **Capacity Limitations in Wireless Sensor Networks**. In Proceedings of Emerging Technologies and Factory Automation (ETFA). 2003. Vol. 2, pp. 529-536.

[BLO02] BLOUGH, D. M.; SANTI, P. **Investigating Upper Bounds on Network Lifetime Extension for Cell-based Energy Conservation Techniques in Stationary Ad Hoc Networks**. In Proceedings of the 8th ACM International Conference on Mobile Computing and Networking (MobiCom), Atlanta, USA, 2002. pp. 183-192.

[BOR04] BORDIM, J. L. NAKANO, K.. Fundamental Protocols to Gather Information in Wireless Sensor Networks. In: MAHGOUB, I; MOHAMMAD, I.. **Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems**. London: CRC Press, 2005. Cap. 23, pp. 23.1-23.15.

[BOU04] BOULIS, A.. Models for Programmability in Sensor Networks. In: MAHGOUB, I; MOHAMMAD, I.. **Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems**. London: CRC Press, 2005. Cap. 4, pp. 4.1-4.13.

[BOU07] BOULIS, A.. **Castalia: Revealing pitfall in designing distributed algorithms in WSN**. In Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys). Sydney, Australia, 2007. pp. 407-408.

[BOU09] BOUKERCHE, Azzedine. **Algorithms and Protocols for Wireless Sensor Network**. Canada: John Wiley & Sons Incorporated, 2009. 544p.

[BRO86] BROOKS, E. D.. The Butterfly Barrier. **International Journal of Parallel Programming**. Vol. 15, nº 4. Ago. 1986. pp. 295-307.

[BRO06] BROWNFIELD, M. I.. **Energy-efficient Wireless Sensor Network MAC Protocol**. Tese (Doutorado) - Faculty of Virginia Polytechnic Institute and State University, Blacksburg, 2006.

[BRY77] BRYANT, R. E.. **Simulation of Packet Communication Architecture Computer Systems**. Technical Report TR-188, MIT Laboratory of Computer Science, Massachusetts, 1977.

[BUL01] BULL, J. M. et al.. **Benchmarking Java against C and Fortran for Scientific Applications**. In Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande. Palo Alto, USA, 2001. pp. 97-105.

[BUR10] BURNS, A. et al.. SHIMMER – A Wireless Sensor Platform for Noninvasive Biomedical Research. **IEEE Sensor Journal**. Vol. 10, nº 9. Sep. 2010. pp. 1527-1534.

[BTN07] BTnodes: A Distributed Environment for Prototyping Ad Hoc Networks. Disponível em: <http://www.btnode.ethz.ch/>. Acesso em: 22 set. 2008.

[CHA79] CHANDY, K. M.; MISRA, J.. **Distributed Simulation: A Case Study in Design and Verification of Distributed Programs**. In Proceedings of IEEE Transactions on Software Engineering. Los Alamitos, USA, 1979. Vol. 5, nº 5. pp. 440-452.

[CHI02] CHIASSERINI, C. F. et al.. **Energy Efficient Design of Wireless Ad Hoc Networks**. In Proceedings of 2nd IFIP Networking. Pisa, Italy, 2002. pp. 376-386.

[CAL03] CALLAWAY Jr., E. H.. **Wireless Sensor Networks – Architectures and Protocols**. USA: CRC Press, 2004. 342p.

[CAR05] CARLEY, T. W. **Sidh: A Wireless Sensor Network Simulator**. ISR Technical Report, University of Maryland, USA, 2005. Disponível em: <http://drum.lib.umd.edu/bitstream/1903/6565/1/TR_2005-88.pdf>. Acesso em: 15 ago. 2012.

[CAR05b] CARDEI, M. et al.. **Energy-Efficient Target Coverage in Wireless Sensor Networks**. In Proceedings of the 24th IEEE Conference on Computer Communications (INFOCOM). Miami, USA, 2005. Vol. 3. pp. 1976-1984.

[CAR06] CARBUNAR, B. et al.. Redundancy and coverage detection in sensor networks. **ACM Journal of Transactions on Sensor Networks** (TOSN). Vol. 2, nº 1. Fev. 2006. pp. 94-128.

[CAV02] CAVIN, D.; SASSON, Y.; SCHIPER, A. **On the accuracy of MANET simulators**. In Proceedings of the 2nd ACM International Workshop on Principles of Mobile Computing. Toulouse, France, 2002. pp. 38-43.

[CAV07] CAVALCANTE, R. A.; SILVA, T.R. **Estudo do sinal de rádio frequência em redes sem fio do âmbito da UCG/Área III**. Trabalho de Conclusão de Curso (Ciência da Computação). Universidade Católica de Goiás, Goiás, 2008.

[COD04] Projeto CodeBlue (Wireless Sensors for Medical Care). Disponível em: <http://fiji.eecs.harvard.edu/CodeBlue>. Acesso em 28 abr. 2013.

[COL03] COLERI, S. et al.. **Power Efficient System for Sensor Networks**. In Proceedings of the 8th IEEE International Symposium on Computer and Communication (ISCC). Antalya, Turquia, 2003. pp. 837-843.

[COR12] CORMEN, T. H. et al.. **Algoritmos: Teoria e Prática**. Rio de Janeiro: Campus. 3ª Edição. 2012. 944p.

[CHA08] CHAMAM, A. ; PIERRE, S.. **Power-Efficient Clustering in Wireless Sensor Networks under Coverage Constraint**. In Proceedings of the International Conference on Wireless and Mobile Computing, Networking and Communications (WIMOB). Avignon, França, 2008. pp. 460-465.

[CHE04] CHEN, G. et al.. SENSE : A wireless sensor network simulator. In: SZYMANSKI, B. ; YENER, B.. **Advanced in Pervasive Computing and Networking**. New York: Springer, 2004. pp. 249-267.

[CHE05] CHEN, Y. ; ZHAO, Q.. On the Lifetime of Wireless Sensor Networks. **IEEE Communications Letters**. Vol. 9, nº 11. Nov 2005. pp. 976-978. Disponível em: <http://interface.cipic.ucdavis.edu/~qzhao/ChenZhao05COML.pdf> Acesso em : 06 nov. 2012.

[CHI01] CHIASSERINI, C. F. ; RAO, R. R.. Improving Battery Performance by Using Traffic Shaping Techniques. **IEEE Journal on Selected Areas in Communications**. Vol. 19, nº 7. Jul 2001. pp. 1385-1394.

[CHI12] CHIU, J. et al.. A Watershed-Based Debris Flow Early Warning System Using Sensor Web Enabling Techniques in Heterogeneous Environments. **IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing**. Vol. 5, nº 6. Dez 2012. pp. 1729-1739.

[CIA06] CIANCIO, A. et al.. **Energy-Efficient Data Representation and Routing for Wireless Sensor Networks Based on a Distributed Wavelet Compression Algorithm**. In Proceedings of the 5th International Symposium on Information Processing in Sensor Networks (IPSN). Nashville, USA, 2006. pp. 309-316. Disponível em : <http://sipi.usc.edu/~ortega/Papers/IPSN06-Ciancio.pdf>. Acesso em : 29 jan. 2013.

[COO13] Cookies WSN. Disponível em: <http://cookieswsn.wordpress.com/>. Acesso em: 30 abr. 2013.

[CR_14] Energizer CR2032: Lithium Coin Battery. Disponível em: <http://data.energizer.com/PDFs/cr2032.pdf>. Acesso em: 21 jan. 2014.

[DAM06] DAM, T. V.; LANGENDOEN, K.. **An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Network**. In Proceedings of the 1st International Conference of on Embedded Networked Sensor Systems (SenSys). Los Angeles, USA, 2003. pp. 171-180.

[DAV98] DAVIS, G. ; CASADY, W. ; MASSEY, R.. Precision Agriculture: An Introduction. **Wather quality**. University of Missouri Extension, USA, 1998. pp. 1-7. Disponível em : <http://extension.missouri.edu/explorepdf/envqual/wq0450.pdf>. Acesso em : 04 mar. 2013.

[DAI05] DAI, S. et al.. **Research and Analysis on Routing Protocols for Wireless Sensor Network**. In Proceedings of the IEEE International Conference on Communications, Circuits and Systems (ICCCAS). Hong Kong, China, 2005. pp. 407-411.

[DAV12] DAVIS, A; CHANG, H. **Airport protection using sensor networks**. In Proceedings of the IEEE Conference on Technologies for Homeland Security (HST). Waltham, USA, 2012. pp. 36-42.

[DEI99] DEITEL, H. M.; DEITEL, P. J. **Java: How to Program**. New Jersey: Prentice Hall. 3rd Edition. 1999. 1355p.

[DEI00] DEITEL, H. M.; et al.. **XML: How to Program**. New Jersey: Prentice Hall. 2000. 934p.

[DEM06] DEMIRKOL, I. et al.. Wireless Sensor Networks for Intrusion Detection: Packet Traffic Modeling. **IEEE Communications Letters**. Vol. 10, nº 1. Jan 2006. pp. 22-24.

[DIE09] DIETRICH, I.; DRESSLER, F. On the Lifetime of Wireless Sensor Networks. **ACM Journal of Transactions on Sensor Networks (TOSN).** Vol. 5, nº 1. Fev 2009. pp. 1-38. Disponível em: <http://www.ccs-labs.org/bib/dietrich2009lifetime/dietrich2009lifetime.pdf>. Acesso em: 14 dez 2012.

[DON05] DONG, Q.. **Maximizing System Lifetime in Wireless Sensor Networks**. In Proceedings of the 4th International Conference on Information Processing in Sensor Networks (IPSN). Los Angeles, USA, 2005. pp. 13-19.

[DUL02] DULMAN, S.; HAVINGA, P.. **Operating System Fundamentals for the EYES Distributed Sensor Network**. In Proceedings of Progress 2002. Utrech, Netherlands, 2002. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.11.5073&rep=rep1&type=pdf>. Acesso em: 09 set. 2011.

[DUN04] DUNKELS, A. et al.. **Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors**. In Proceedings of the 29th Annuall IEEE International Conference on Local Computer Networks. Tampa, USA, 2004. pp. 455-462. Disponível em: <http://www.sics.se/~adam/ dunkels04contiki.pdf> Acesso em: 23 fev. 2007.

[DUS02] Projeto Smart Dust. Disponível em: <http://robotics.eecs.berkeley.edu/~pister/SmartDust/>. Acesso em: 28 abr 2013.

[EGE05] EGEA-LÓPEZ, E. et al.. **Simulation Tools for Wireless Sensor Networks**. In Proceedings of the Summer Simulation Multi-conference (SPECTS). Philadelphia, USA, 2005. pp. 2-9. Disponível em: <http://www.ait.upct.es/~eegea/pub/spects05.pdf>. Acesso em: 12 nov. 2011.

[ELS04] ELSON, J.; ESTRIN, D.. Sensor Networks: A Bridge to the Physical World. In: RAGHAVENDRA, C. S.. **Wireless Sensor Networks**. New York: Springer Science+Business Media Inc., 2004. Cap. 1, pp. 3-20.

[IEE08] IEEE 754-2008: IEEE Standard for Floating-Point Arithmetic. Aug. 2008. p. 1-58. Disponível em: <http://ieeexplore.ieee.org/>. Acesso em: 17 mar. 2008.

[FAG86] FAGAN, M. E.. Advances in software inspections. **IEEE Transactions on Software Engineering**. Vol. SE-12, nº 7. Jul 1986. pp. 744-751.

[RFM99] TR1000 - 916.50 MHz Hybrid Transceiver. RF Monolithics, Inc. Disponível em: <http://www.rfm.com/products/data/tr1000.pdf>. Acesso em: 27 mai 2007.

[CC_09] CC2500 - Low-Cost Low-Power 2.4 GHz RF Transceiver. Chipcom. Disponível em: <http://www.ti.com/lit/ds/symlink/cc2500.pdf>. Acesso em: 25 set 2012.

[FAN05] FANG, X. et al.. Fast synchronization on shared-memory multiprocessors: An architectural approach. **Journal of Parallel and Distributed Computing**. Vol. 65. 2005. pp. 1158-1170.

[FUJ90] FUJIMOTO, R. M.. Paralell Discrete Event Simulation. **Communications of the ACM Magazine: Special Issue on Simulation**. Vol. 33, nº 10. Out. 1990. pp. 30-53.

[FUJ00] FUJIMOTO, R. M.. **Parallel and Distributed Simulation Systems**. New York: John Wiley & Sons Inc. 2000. 300p.

[FUJ01] FUJIMOTO, R. M.. **Parallel and Distributed Simulation Systems**. In Proceedings of the ACM Winter Simulation Conferece. New York, USA, 2001. pp. 147-157.

[GAN13] GANGWAR, D. S.. **Biomedical Sensor Network for Cardiovascular Fitness and Activity Monitoring**. In Proceedings of the 2013 IEEE Point-of-Care Heathcare Technologies (PHT). Bangalore, India, 2013. pp. 279-282.

[GAY03] GAY, D. et al.. **The nesC Language: A Holistic Approach to Networked Embedded Systems**. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). San Diego, USA, 2003. pp. 1-11.

[GEO11] GEORGOULAS, D.; BLOW, K.. In-Motes EYE: A Real Time Application for Automobiles in Wireless Sensor Networks. **Wireless Sensor Network**. Vol 3, nº 5, 2011. pp. 158-166.

[GIL02] GILBERT, C.; SZYMANSKI, B. K.. **COST: a component-oriented discrete event simulator**. In Proceedings of the Winter Simulation Conference. San Diego, USA, 2002. Vol. 1. pp. 776-782.

[GIR07] LEWIS, G. et al.. Emstar: A Software Environment for Developing and Deploying Heterogeneous Sensor-Actuator Networks. **ACM Transactions on Sensor Networks**. Vol. 3, nº 3. Fev 2007. pp. 1-34.

[GIU09] GIUSTO, D. et al.. **The Internet of Things**. In the Proceedings of the 20th Tyrrhenian Workshop on Digital Communications. Pula, Italia, 2009.

[GOE05] GÓES, L. F. W. et al.. **JSDESLib: A library for the Development of Discrete-Event Simulation Tools fot Parallel Systems**. In Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium. Denver, USA: 2005.

[GOM95] GOMES, F. et al.. **SimKit: A High Performance Logical Process Simulation Class Library in C++**. In Proceedings of the 1995 Winter Simulation Conference. Arlington, USA, 1995. pp. 706-713.

[GRO02] GRÖTKER, T. et al.. **System Design with SystemC**. USA: Kluwer Academic Publishers. 2002. 326p.

[HSU98] HSU, V. S.; KAHN, J. M.; PISTER, K. S. J.. **Wireless Communications for Smart Dust**. University of California - Electronics Research Laboratory Memorandum Number M98/2. 1998. Disponível em: <http://robotics.eecs.berkeley.edu/~pister/publications/1998/smartdust_comm_memo%5B1%5D.pdf>. Acesso em: 11 abr. 2012.

[JAI11] JAIN, R.. **A Survey of Wireless Sensor Network Simulation Tools**. Disponível em: <http://www.cse.wustl.edu/~jain/cse567-11/ftp/sensor/index.html>. Acesso em: 20 fev. 2012.

[JAV12] Java Home Page. Oracle Systems. Disponível em: <http://www.java.com>. Acesso em 21 mar. 2012.

[JUN07] JUNG, D.. **MATSNL: a MATLABTM Wireless Sensor Node Platform Lifetime Prediction & Simulation Package**. Disponível em: <http://www.osti.gov/eprints/topicpages/documents/record/975/0108846.html>. Acesso em: 15 out. 2008.

[KAL11] KALAIVANI, T. et al.. **A survey on Zigbee based wireless sensor networks in agriculture**. In Proceedings of the 3rd International Conference on Trendz in Information Sciences and Computing (TISC). Chennai, Índia, 2011. p. 85-89.

[KHA11] KHAN, Z. M. et al.. **Limitations of Simulation Tools for Large-Scale Wireless Sensor Networks**. In Proceedings of the 2011 Workshop of International Conference on Advanced Information Networking and Applications (WAINA). Singapura, 2011. pp. 820-825.

[KNE93] KNEPELL, P. L.; ARANGNO, D. C.. **Simulation Validation: A Confidence Assessment Methodology (Systems)**. Wiley-IEEE Computer Society Press. 1st Edition, 1993. 168 p.

[KO_10] KO, J. et al.. **Egs: A Cortex M3-Based Mote Platform**. In Proceedings of the 7th Annual IEEE Communications Society Conference on Sensor Mesh and Ad Hoc Communications and Networks. Boston, USA, 2010. pp. 1-3.

[KOK00] KOKAR, M. BACLAWSKI, K.. **Modeling combined time- and event-driven dynamic systems**. In Proceedings of the 9th OOPSLA Workshop on Behavioral Semantics. Minneapolis, USA, 2000. pp. 122-129.

[KÖP08] KÖPKE, A. et al.. **Simulating wireless and mobile networks in OMNet++: the MiXiM vision**. In Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops. Marseille, France, 2008.

[KAN05] KANG, I.; POOVERNDRAN, R. Maximizing Network Lifetime of Broadcasting over Wireless Stationary Ad Hoc Networks. **Mobile Networks and Apllications (MONET): Special Issue on Energy Constraints and Lifetime Performance in Wireless Sensor Networks**. Vol. 10, nº 6. Dec 2005. pp. 879-896.

[KOR09] KORKALAINEN, M. et al.. **Survey of Wireless Sensor Networks Simulation Tools for Demanding Applications**. In Proceedings of the 5th International Conference on Networking and Services (ICNS). Valência, Itália, 2009. pp. 102-106.

[KOT04] KOTZ, D. et al.. **Experimental Evaluation of Wireless Simulation Assumptions**. In Proceedings of the 7th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM). Veneza, Itália, 2004. pp. 78-82.

[KRE97] KREYTZER, W.; HOPKINS, J.; MIERLO, M.. **SimJAVA – A framework for modeling queueing networks in JAVA**. In Proceedings of the 1997 Winter Simulation Conference. Atlanta, USA, 1997. Pp. 483-488.

[KRO05] KROLLER, A. et al.. **Shawn: a new approach to simulating wireless sensor networks**. In Proceedings of Design, Analysis and Simulation of Distributed Systems (DASD). San Diego, USA, 2005. pp. 117-124. Disponível em: <http://arxiv.org/pdf/cs/0502003.pdf>. Acessso em: 20 nov. 2009.

[KRO08] KRONEWITTER, F. D.. **Dynamic Huffman Addressing in Wireless Sensor Netoworks Based on the Energy Map**. In Proceedings of the 2008 Military Communications Conference (MILCOM). San Diego, USA, 2008. pp. 1-6.

[KUM05] KUMAR, S.; ARORA, A.; LAI, T. H.. **On the Lifetime Analysis of Always-On Wireless Sensor Network Applications**. In Proceedings of the 2nd IEEE International Conference of Mobile and Ad Hoc Sensor Systems (MASS). Washington, USA, 2005. pp. 188-190.

[KUR05] KURKOWSKI, S.; CAMP, T.; COLAGROSSO, M.. MANET Simulation Studies: The Incredibles. **ACM Mobile Computing and Communications Review: Special Issue on Medium Access and Call Admission Control Algorithms for Next Generation Wireless Networks**. Vol. 9, nº 4. Oct. 2005. pp. 50-61.

[KUM12] KUMAR, P.; SINGH, M. P.; TRIAR, U. S.. A Review of Routing Protocols in Wireless Sensor Networks. **International Journal of Engineering Research & Technology**. Vol. 1, nº 4. Jun. 2012. pp. 1-4.

[LAZ05] LAZOS, L.; POOVENDRAN, R.. SeRLoc: Robust Localization for Wireless Sensor Networks. **ACM Transactions on Sensor Networks**. Vol. 1, nº 1. Ago. 2005. pp. 73-100.

[LI_01] LI, S.. **Low Power Operating Systems for Heterogeneous Wireless Communication Systems**. In Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques. Barcelona, Espanha, 2003. pp. 1-16.

[LI_04] LI, X.; WANG, Y.. Wireless Sensor Networks and Computational Geometry. In: MAHGOUB, I; MOHAMMAD, I.. **Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems**. London: CRC Press, 2005. Cap. 39, pp. 39.1-39.49.

[LEV03] LEVIS, P. et al.. **TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications**. In Proceedings of the 1st ACM Conference on Embedded Networked Sensor Systems. Los Angeles, 2003. pp. 126-137.

[LHA12] LAHMAR, K.; CHÉOUR, R.; ABID, M.. **Wireless Sensor Networks: Trends, Power Consumption and Simulators**. In Proceedings of the 6th Asia Modelling Synposium (AMS). Bali, Indonésia, 2012. pp. 200-204.

[LIN02] LINDSEY, S.; RAGHAVENDRA, C. S.. **PEGASIS: Power-efficient gathering in sensor information systems**. In Proceedings of the 2002 IEEE Aerospace Conference. Big Sky, USA, 2002. Vol. 3. pp. 1125-1130.

[LIN13] LIN, T. et al.. An Ultra-Low Power Asynchronous-Logic In-Situ Self-Adaptative V_{DD} System for Wireless Sensor Networks. **IEEE Journal of Solid-State Circuits**. Vol. 48, nº 2. 2013. pp. 573-586.

[LIU05] LIU, H. et al.. **Maximal Lifetime Scheduling in Sensor Surveillance Networks**. In the Proceedings of the 24th IEEE Conference on Computer Communications (INFOCOM). Miami, USA, 2005. Vol. 4. pp. 2482-2491.

[LOH11] LOHIER, S. et al.. **Wireless Sensor Network Simulators Relevance compared to a real IEEE 802.15.4 Testbed**. In Proceedings of the 7th International Wireless Communications and Mobile Computing Conference. Istambul, Turquia, 2011. pp. 1347-1352.

[LOM90] LOMOW, G.; BAEZNER, D.. **A Tutorial Introduction to Object-oriented Simulation and Sim++**. In Proceedings of the 1990 Winter Simulation Conference. New Orleans, USA, 1990. pp. 149-153.

[LOW99] LOW, Y. et al.. Survey of Languages and Runtime Libraries for Parallel Discrete-Event Simulation. **Simulation**. Vol. 72, nº 3. 1999. pp. 170-186.

[LUC03] LUCIO, G. F. et al.. **OPNET Modeler and Ns-2: Comparing the Accuracy Of Networked Simulator for Packet-Level Analysis using a Network Testbed**. In Proceedings of the 3rd WEAS International Conference on Simulation, Modelling and Optimization (ICMSAO). Sharjah, Emirados Árabes Unidos, 2003.

[LUK05] LUKE, S. et al.. MASON: A Multi-Agent Simulation Environment. **SIMULATION: Transactions of the society for Modeling and Simulation International**. Vol. 81, nº 7. Jul. 2005. pp. 517-527.

[KUL99] KULIK, J.; RABINER, W.; BALAKRISHNAN, H.. **Adaptive Protocol for Information Dissemination in Wireless Sensor Networks**. In Proceedings of the 5th annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom). Seattle, USA, 1999. pp. 174-185.

[HAE04] HAENGGI, M.. Opportunities and Challenges in Wireless Sensor Networks. In: MAHGOUB, I; MOHAMMAD, I.. **Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems**. London: CRC Press, 2005. Cap. 1, p. 1.1-1.14.

[HAN03] HANDZISKI, V. et al.. **A common wireless sensor network architecture?** Technical report TKN-03-012 on the Telecommunications Networks Group, Technische Unisersität Berlin, Berlin, 2003. pp. 10-17.

[HAN05] HAN, C. et al.. **A Dynamic Operating System for Sensor Nodes**. In Proceedings of the 3rd International Conference on Mobile Systems, Aplications and Services (ModiSys). Seattle, USA, 2005. pp. 163-176.

[HAY08] HAYKIN, S.; MOHER, M.. **Sistemas Modernos de Comunicação Wireless**. Porto Alegre: Bookman, 2008. 580p.

[HE_03] HE, T. et al.. **SPEED: A Stateless Protocol for Real-Time Communication in Sensor Networks**. In Proceedings of the 23rd International Conference on Distributed Computing Systems. Providence, USA, 2003. pp. 46–55.

[HE_03] HE, T. et al.. **Range-Free Localization Schemes for Large Scale Sensor Networks**. In Proceedings of the 9th Annual International Conference on Mobile Computing and Networking (MobiCom). San Diego, USA, 2003. pp. 81-95.

[HED88] HEDETNIEMI, D.; LIESTMAN, A.. A Survey of Gossiping and Broadcasting in Communication. **Networks: An International Journal**. Vol. 18, nº 4. 1988. pp. 319-349.

[HEN88] HENGSEN, D.; FINKEL, F; MANBER, U.. Two Algorithms for Barrier Synchronization. **International Journal of Parallel Programming**. Vol. 17, nº 1. Fev. 1988. pp. 1-17.

[HEI99] HEINZELMAN, W. R.; KULIK, J.; BALAKRISHNAN, H.. **Adaptive Protocols for Information Dissemination in Wireless Sensor Networks**. In Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking. Seattle, USA. pp. 174-185.

[HEI00] HEINZELMAN, W.; CHANDRAKASAN, A.; BALAKRISHNAN, H.. **Energy-efficient communication protocol for wireless microsensor networks**. In Proceedings of the 33rd Annual Hawaii International Converence on System Sciences. Hawaii, USA, 2000. Vol. 8. pp. 8020.

[HIL98] HILL, J. M. D.; SKILLICORN, D. B.. **Practical Barrier Synchronization**. In Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing. Madrid, Espanha, 1998. pp. 438-444.

[HUA07] HUANG, T. et al.. **LA-TinyOS: A Locality-Aware Operationg System for Wireless Sensors Networks**. In Proceedings of the 2007 ACM Symposium on Applied Computing (SAC). Seoul, Korea, 2007. pp. 1151-1158.

[IEE11] 802.15.4-2011 - IEEE Standard for Local and metropolitan area networks--Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs). Disponível em: <http://standards.ieee.org/findstds/standard/802.15.4-2011.html>. Acesso em: 22 julho 2014.

[IEE12] 802.15.4e-2012 - IEEE Standard for Local and metropolitan area networks--Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC sublayer. Disponível em: <http://standards.ieee.org/findstds/standard/802.15.4e-2012.html>. Acesso em: 22 julho 2014.

[IMO08] Projeto IMote (Intel Mote**)**. Disponível em: <http://www.intel.com/research/exploratory/motes.htm>. Acesso em: 10 nov. 2008.

[IMR10] IMRAN, M.; SAID, A. M.; HASBULLAH, H.. **A Survey on Simulators, Emulators and Testbesd for Wireless Sensor Networks**. In Proceedings of the 2010 International Symposium in Information Technology (ITSim). Kuala Lampur, Malásia, 2010. Vol. 2. pp. 897-902.

[INT00] INTANAGONWIWAT, C.. **Direct Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks**. In Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom). Boston, USA, 2000. pp. 56-67.

[MAI02] MAINWARING, A. et al.. **Wireless Sensor Network for Habitat Monitoring**. In Proceedings of the Wireless Sensor Networks and Applications (WSNA). Atlanta, USA, 2002. pp. 88-97.

[MAK09] MAK, N. H.; SEAH, W. K. G.. **How long is the lifetime of a wireless sensor network?** In Proceedings of the International Conference on Advanced Information Networking and Applications. Bradford, Inglaterra, 2009. pp. 763-770.

[MAN01a] MANJESHWAR, E.; AGRAWAL, D. P.. **TEEN: A Routing Protocol for Enhanced Efficiency in Wireless Sensor Networks**. In Proceedings of the 15th International Parallel and Distributed Processing Symposium. San Franscisco, USA, 2001. pp. 2009-2015.

[MAN01b] MANJESHWAR, E.; AGRAWAL, D. P.. **APTEEN: A Hybrid Protocol for Efficient Routing and Comprehensive Information Retrieval in Wireless Sensor Networks**. In Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS). Ft. Lauderdale, USA, 2001. pp. 195-202.

[MAN03] Projeto MANTIS (*Multimodal Network of In-situ Sensors*). Disponível em: <http://mantis.cs.colorado.edu/index.php/tiki-index.php>. Acesso em: 29 mai. 2007.

[MAT04] MATEUS, G. R.; LOUVEIRO, A. A. F.. **Introdução à Computação Móvel**. 2ª Edição. 115p. Disponível em: <www.dcc.ufmg.br/~loureiro/cm/docs/cm_livro_2e.pdf>. Acesso em: 17-mar-2010. Versão Preliminar.

[MAY90] MAYRHAUSER, A.. **Software Engineering: Methods and Management**. London: Academic Press Limited, 1990. 864p.

[MIC02] Projeto Mica Motes. Disponível em: <http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA.pdf>. Acesso em: 29 mai. 2007.

[MIC02b] Projeto MicroAmps - Micro-Adaptive Multi-domain Power aware Sensors. Disponível em: <http://mtlweb.mit.edu/researchgroups/icsystems/uamps/>. Acesso em: 28 abr. 2013.

[MIC13a] Mica mote. Disponível em: <http://www.sensorsmag.com>. Acesso em: 28 abr. 2013.

[MIC13b] MICA2DOT Hardware. Disponível em: <http://www.tinyos.net/scoop/special/hardware#mica2dot>. Acesso em: 28 abr. 2013.

[MIC13c] MICA-Z Hardware. Disponível em: <http://www.openautomation.net/uploadsproductos/micaz_datasheet.pdf>. Acesso em: 28 abr. 2013.

[MIR11] Miranda, F. A. M.. **Proposta de Alocação de Baterias em Redes de Sensores Sem Fio Orientada à Maximização do Tempo de Vida**. Dissertação (Mestrado). Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e Computação. Campinas, 2011.

[MUL97] MULLER, P.. **Instant UML**. Paris: Wrox Press Ltd., 1997. 345p.

[NAC08] NACHMAN, L. et al.. **IMOTE2: Serious Computation at the Edge**. In Proceedings of the International Wireless Communications and Mobile Computing Conference. Ilha de Creta, 2008. pp. 1118-1123.

[NAD7] NADALIN, E. Z.. **Determinação da força peso, a partir dos impactos de pisada, utilizando um sensor piezoelétrico**. Dissertação (Mestrado). Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e Computação. 2007. 122p.

[NIC95] NICOL, D. M.. Noncommittal Barrier Synchronization. **Parallel Computing**. Vol. 21, nº 4. Abr. 1995.00: 529-549.

[NS_97] **T**he Network Simulato: ns-2. Disponível em: <http://www.isi.edu/nsnam/ns/>. Acesso em: 05 jan. 2013.

[NS_05] The Network Simulator: ns-3. Disponível em: <http://www.nsnam.org/>. Acesso em: 05 jan. 2013.

[ODE08] ODEN, C. P. et al.. **Wireless Sensor Network in Geophysics**. In Proceedings of the 21st EEGS Symposium on the Application of Geophysics to Engineering and Environmental Problems (SAGEEP). Philadelphia, USA, 2008. Vol. 21, nº 1. pp. 963-971.

[OUL05] OULD-AHMED-VALL, E. et al.. **Simulation of large-scale sensor netwokrs using GTSNetS**. In Proceedings of the 13th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. Atlanta, USA, 2005. pp. 211-218.

[OPN12] OPNET: Network Simulator. Disponível em: <http://www.opnet.com/solutions/network_rd/modeler.html>. Acesso em: 13 dez. 2012.

[PAP04] PAPAVASSILIOU, S; ZHU, J.. Architecture and Modeling of Dynamic Wireless Sensor Networks. In: MAHGOUB, I; MOHAMMAD, I.. **Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems**. London: CRC Press, 2005. Cap. 15, p. 15.1-15.14.

[PAR00] PARK, S.; SAVVIDES, A.; SRIVASTAVA, M. B.. **SensorSim: a simulation framework for sensor networks**. In Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (WSWIN). Milwaukee, UA, 2000. pp. 104-111.

[PAR06] PARK, S. et al.. **A Nano Operating System for Wireless Sensor Networks**. In Proceedings of the 8th International Conference in Advanced Communication Technology (ICACT). Phoenix Park, Irlanda, 2006. pp. 348-352.

[PER08] PERLA, E. et al.. **PowerTOSSIM Z: Realistic Energy Modeling for Wireless Sensor Network Enrironments**. In Proceedings of the 3rd ACM Workshop on Performance Monitoring and Measurement of Heretogeneous Wireless and Wired Networks (PM2HW2N). Vancouver, Canadá, 2008. pp. 35-42.

[PIC03] Projeto PicoRadio. Disponível em: <http://bwrc.eecs.berkeley.edu/Research/Pico_Radio/Default.htm>. Acesso em: 27 abr. 2013.

[POL04] POLLEY, J. et al.. **ATEMU: A Fine-grained Sensor Network Simulator**. In Proceedings of the IEEE International Conference on Sensor and Ad Hoc Communication Networks. Santa Clara, USA, 2004. pp. 145-152.

[POL04b] POLASTRE, J. et al.. **The mote revolution: Low power wireless sensor network devices**. In Proceedings of the HotChip 16: A Symposium on High Performance Chips. Stanford, USA, 2004.

[PUL07] W1030: Wireless External Antenna for 2.4GHz Applications. Pulse: A Technitrol Company. Disponível em: <http://www.pulseelectronics.com/download/2962/w1030pdf>. Acesso em: 24 ago. 2012.

[QUA01] QualNet Simulator User's Manual. Disponível em: <http://www.eurecom.fr/~chenj/SimulatorManual-3.1.pdf>. Acesso em: 12 dez. 2012.

[RAB00] RABAEY, J. M. et al.. PicoRadio Supports Ad Hoc Ultra-Low Power Wireless Networking. **Computer**. Jul. 2000. Vol. 33, nº 7. pp. 42-48.

[RAI05] RAICU, I. et al.. Local Load Balancing for Globally Efficient Routing in Wireless Sensor Networks. **International Journal of Distributed Sensor Networks**. 2005. Vol. 1, nº 2. pp. 163-185.

[RAC10] RACHEDI, A. et al.. **Wireless Network Simulators Relevance Compared to a Real Testbed in Outdoor and Indoor Environments**. In Proceedings of the 6th International Wireless Communications and Mobile Computing Conference (IWCMC). Istambul, Turquia, 2010. pp. 346-350.

[RAJ03a] RAJARAVIVARMA, V.; YANG, Y.; YANG, T.. **An Overview of Wireless Sensor Network and Applications**. In Proceedings of the 35th Southeastern Symposium on System Theory. Morgantown, USA, 2003. pp. 432-436.

[RAJ03b] RAJENDRAN, V. et al.. **Energy-Efficient Collision-Free Medium Access Control for Wireless Sensor Networks**. In Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys). Los Angeles, USA, 2003. pp. 181-192.

[RIB12] Ribeiro, D. H. et al.. **JSensor: A parallel simulator for wireless sensor network and distributed systems**. In Proceedings of the 41st International Conference on Paralell Processing Workshops (ICPPW). Pittsburg, USA, 2012. pp. 604-605.

[RIL03] RILEY, G. F.. **The Georgia Tech Network Simulator**. In Proceedings of the 2003 Workshop on Models, Methods and Tools for Reproducible Network Research (MoMeTools). Karlsruhe, Alemanha, 2003. pp. 5-12.

[RUI03] RUIZ, L. B.. **MANÁ: Uma Arquitetura para Gerenciamento de Redes de Sensores Sem Fio**. Tese (Doutorado). Universidade Federal de Minas Gerais. Belo Horizonte, 2003. Disponível em: <http://homepages.dcc.ufmg.br/~linnyer/TeseMANNA.pdf>. Acesso em: 07 jun. 2008.

[RUI04a] RUIZ, L. B. et al.. **Arquiteturas para Redes de Sensores Sem Fio**. In Proceedings of the 22º Simpósio Brasileiro de Redes de Computadores. Gramado, Brasil, 2004. pp. 167-218.

[RUI04b] RUIZ, L. B.; NOGUEIRA, J. M.; LOUVEIRO, A. A. F.. Sensor Network Management. In: MAHGOUB, I; MOHAMMAD, I.. **Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems**. London: CRC Press, 2005. Cap. 3, pp. 3.1-3.28.

[SAL06] SALUSTIANO, R. E.. **Aplicação de Técnicas de Fusão de Sensores no Monitoramento de Ambientes**. Dissertação (Mestrado). Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e Computação. 2006. Disponível em: <http://www.lpm.fee.unicamp.br/~rsalusti/mestrado/SALUSTIANO-MESTRADO.pdf>. Acesso em: 20 jul. 2013.

[SAL08] SALUSTIANO, R. E.; DOS REIS, C. A.. **Barrier Synchronization Simulatior for Wireless Sensor Networks**. In Proceedings of the 7th IEEE International Caribbean Conference on Devices, Circuits and Systems. Cancun, México, 2008. pp. 1-5.

[SAL09a] SALUSTIANO, R. E.; DOS REIS, C. A.. **A Java Package for Synchronous Simulation Processing**. In Proceedings of the 8th International Conference and Workshop on Ambient Intelligence and Embedded Systems. Madeira, Portugal, 2009.

[SAM04] SAMMER, S.; KIM, W. Y.; A. GUL. **SENS: A Sensor, Environment and Network Simulator**. In Preceedings of the 37th Annual Simulation Symposium (ANSS), Arlington, USA, 2004. pp. 221-228.

[SAN03] SANKARASUBRAMANIAM, Y.; AKAN, O. B.; AKYILDIZ, I. F.. **ESRT: Event-to-sink Reliable Transport in Wireless Sensor Networks**. In Proceedings of the 4th ACM International Symposium on Mobile Ad Hoc Networking & Computing (MobiHOc). Annapolis, USA, 2003. pp. 177-188.

[SAN05] SANTI, P.. **Topology Control in Wireless Ad Hoc and Sensor Networks**. England: John Wiley & Sons, Ltd. 2005. 282p.

[SAH07] SAHA, S; MATSUMOTO, M.. **A Framework for Disaster Management System and WSN Protocol for Rescue Operation**. In Proceedings of the TENCON 2007 IEEE Region 10 Conference. Taipei, China, 2007. pp. 1-4.

[SAR10] SARGENT, R. G.. **Verification and Validation of Simulation Models**. In: Proceedings of the 2010 Winter Simulation Conference (WSC). Baltimore, USA, 2010. pp. 166-183.

[SEH13] SEHGAL, A.. **Using the Contiki Cooja Simulator**. Disponível em: <http://cnds.eecs.jacobs-university.de/courses/iotlab-2013/cooja.pdf>. Acesso em: 06 jul 2014.

[SHA05] SHAH, R. C; PETROVIC, D.; RABAEY, J. M.. Energy-Aware Routing and Data Funneling in Sensor Networks. In: MAHGOUB, I; MOHAMMAD, I.. **Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems**. London: CRC Press, 2005. Cap. 30, pp. 30.1-30.14.

[SHE04] SHEN, C.; JAIKAEO, C.; SRISATHAPORNPHAT, C.. Sensor Network Architecture and Applications In: MAHGOUB, I; MOHAMMAD, I.. **Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems**. London: CRC Press, 2005. Cap. 8, pp. 8.1-8.13.

[SHI06] SHI, Y. T. H.; EFRAT, A.. **Algorithm Design for Base Station Placement Ploblems in Sensor Networks**. In Proceedings of the 3rd International Conference on Quality of Service in Heterogeneous Wired/Wireless Networks. Waterloo, Canadá, 2006. pp. 21-38.

[SHN04] SHNAYDER, V. et al.. **PowerTOSSIM: Efficient Power Simulation for TinyOS Applications**. In Proceedings of the ACM International Conference on Embedded Networked Sensor Systems (SenSys). Baltimore, USA, 2004. Disponível em: <http://www.eecs.harvard.edu/~shnayder/papers/sensys04ptossim.pdf>. Acesso em: 19 mar. 2013.

[SHT11] SHT11 – Humidity and Temperature Sensor IC. Sensirion. Disponível em: <http://www.sensirion.com/fileadmin/user_upload/customers/sensirion/Dokumente/Humidity/Sensirion_Humidity_SHT1x_Datasheet_V5.pdf>. Acesso em: 10 mar 2012.

[SIC05] SICHITIU, M. L.; DUTTA, R.. **Benefits of multiple battery levels for the lifetime of large wireless sensor networks**. In Proceedings of the 4th International IFIP-TC6 Conference on Networking Technologies, Services and Protocols. Waterloo, Canadá, 2005. pp. 1440-1444.

[SIL00] SILVA, J. L., Jr. et al.. **Wireless Protocols Design: Challenges and Opportunities**. In Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES). San Diego, USA, 2000. pp. 147-151.

[SIL04] SILVA, F. A. et al.. Tecnologia de Nós Sensores Sem Fio. **Revista Controle e Instrumentação.** 2004. Disponível em: <http://homepages.dcc.ufmg.br/~linnyer/ufmgnossensores.pdf>. Acesso em: 05 out 2006.

[SLI04] SLIJEPCEVIC, S.; WONG, J. L.; POTKONJAK, M.. Security and Privacy Protection in Wireless Sensor Network. In: MAHGOUB, I; MOHAMMAD, I.. **Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems**. London: CRC Press, 2005. Cap. 31, pp. 31.1-31.15.

[SOB06] SOBEIH, A. H. et al.. J-Sim: a simulation and emulation environmental for wireless sensor networks. **IEEE Wireless Comunications**. Ago. 2006. Vol. 13, nº 4. pp. 104-119.

[SOH00] SOHRABI, K.; GAO, J.; AILAWADHI, V.; POTTIE, G.. Protocols for self-organization of wireless sensor networks. **IEEE Personal Communications**. Out. 2000. Vol. 7, nº 5. pp. 16-27.

[SOL08] SOLTAN, M.; HWANG, I.; PEDRAM, M.. **Modulation-Aware Energy Balancing in Hierarchical Wireless Sensor Networks**. In Proceedings of the 3rd International Symposium on Wireless Pervasive Computing (ISWPC). Santorini, Grécia, 2008. pp. 355-359.

[SOM01] SOMMERVILLE, I.. **Software Engineering**. England: Pearson Education Limited. 6th Edition. 2001. 693p.

[SOR05] SORO, S.; HEINZELMAN, W. B.. **Prolonging the Lifetime of Wireless Sensor Networks via Unequal Clustering**. In Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS). Denver, USA, 2005.

[SWI13] Swing – Package javax.swing API reference. Java Platform, Standard Edition 7. Oracle Systems. Disponível em: <http://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>. Acesso em: 18 nov 2013.

[TAN97] TANENBAUM, A. S.. **Operation Systems: Design and Implementation**. New Jersey: Prentice-Hall, Inc.. 2nd Edition. 1997. 939p.

[TAN03] TANENBAUM, A. S.. **Redes de Computadores.** Rio de Janeiro: Elsevier Editora Ltda. 3ª Edição. 2003. 923p.

[TEL04] Telos: Ultra low power IEEE 802.15.4 compliant wireless sensor module. Disponível em: <http://moss.csc.ncsu.edu/~mueller/rt/rt11/readings/projects/g4/datasheet.pdf>. Acesso em: 6 fev. 2013.

[TIA02] TIAN, D.; GEORGANAS, N. D.. **A coverage-preserving node scheduling scheme for large wireless sensor networks**. In Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA). Atlanta, USA, 2002. pp. 32-41.

[TIN06] TinyOS Project. Disponível em: <http://www.tinyos.net/>. Acesso em: 04 abr. 2013.

[TIT05] TITZER, B. L.; LEE, D. K.; JENS, P.. **Avrora: Scalable Sensor Network Simulation with Precise Timing**. In Proceedings of the 4th International Symposium of Information Processing in Sensor Networks (IPSN). Los Angeles, USA, 2005. pp. 477-482.

[THO98] THONDUGULAN, N.. **Unsynchronized Parallel Discrete Event Simulation**. Dissertação (Mestrado). University of Cincinnati. Department of Electrical and Computer Engineering and Computer Science. USA, 1998. 77p. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.24.2405&rep=rep1&type=pdf>. Acesso em 20 fev. 2011.

[TMO13] Projeto Tmote Sky. Disponível em: <http://tmote-sky.blogspot.com.br/2012/04/tmote-sky-lower-power-wireless-network.html>. Acesso em: 27 abr. 2013.

[TUB03] TUBAISHAT, M.; MADRIA, S.. Sensor Networks: An Overview. **IEEE Potentials**. Vol. 20, nº 2. 2003. pp. 20-23.

[TYA02] TYAN, H.. **Design, realization and evaluation of a componente-based compositional software architecture for network simulation**. Tese (Doutorado). Ohio State University. Departamento de Engenharia Elétrica. USA, 2002. Disponível em: <http://j-sim.cs.uiuc.edu/whitepapers/tyan_thesis.pdf>. Acesso em 13 abr. 2009.

[TZE05] TZENG, N.; KASULA, B.; WU, H.. **Efficient Barrier Synchronization on Wireless Computing Systems**. In Proceedings of the 11th International Conference on Parallel and Distributed Systems. Fukuoka, Japão, 2005. Vol. 1. pp. 782-788.

[UAM13] Projeto µ-Amps. Disponível em: <http://www-mtl.mit.edu/researchgroups/icsystems/uamps/>. Acesso em 13 abr. 2013.

[VER05] VERDONE, R.; BURATTI, C.. **Modelling for Wireless Sensor Network Protocol Design**. In Proceedings of the International Workshop on Wireless Ad-hoc Networks (IWWAN). Londres, Inglaterra, 2005.

[VIE04] VIEIRA, M. A. M.. **BEAN: Uma Plataforma Computacional para Rede de Sensores Sem Fio**. Dissertação (Mestrado). Universidade Federal de Minas Gerais (UFMG). Departamento de Ciência da Computação. Belo Horizonte, 2004. Disponível em: <http://www.dcc.ufmg.br/~mmvieira/publications/bean.pdf>. Acesso em: 03 ago 2006.

[VU_09] VU, C. T.; LI, Y.. **Delaunay-triangulation based complete coverage in wireless sensor networks**. In Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom). Galveston, USA, 2009. pp. 1-5.

[YE_06] YE, W.; HEIDEMANN, J.; ESTRIN, D.. **An Energy-Efficient MAC Protocol for Wireless Sensor Networks**. In Proceedings of the 3rd IEEE Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON). Reston, USA, 2006. Vol. 3. pp. 1567-1576.

[YOU07] YOUSSEF, M.; EL-SHEIMY, N.. **Wireless Sensor Network: Research vs. Reality Design and Deployment Issues**. In Proceedings of the 5th Annual Conference on Communication Networks and Services Research (CNSR). Frederlcton, Canadá, 2007. pp. 8-9.

[YU_01] YU, Y.; GOVINDAN, D. E. **Geographical and Energy Aware Routing: a recursive data dissemination protocol for wireless sensor networks**. Technical Report UCLA-CSD TR-01-0023, UCLA Computer Science Department. Los Angeles, 2001.

[YU_06] YU, Y.; PRASANNA, V. K.; KRISHNAMACHARI, B.. **Information Processing and Routing in Wireless Sensor Networks**. USA: World Scientic Publising Company, 2006. 204 p.

[YUA04] YUAN, L.; QU, G.. Energy-Efficient Design of Distributed Sensor Networks. In: MAHGOUB, I; MOHAMMAD, I.. **Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems**. London: CRC Press, 2005. Cap. 38, pp. 38.1-38.18.

[WIL08] WILSON, W.; ATKINSON, G.. Wireless Sensing Opportunities for Aerospace Applications. **Sensor & Transducers Journal**. Vol. 97, nº 7. Jul. 2008. pp. 83-90.

[WAN02] WAN, C. Y.; CAMPBELL, A. T.; KRISHNAMURTHY, L.. **PSQF: A Reliable Transport Protocol for Wireless Sensor Networks and Applications**. In Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA). Atlanta, USA, 2002. pp. 1-11.

[WAN04] WANG, Q. HASSANEIN, H.; XU, K.. A Practical Perspective on Wireless Sensor Networks. In: MAHGOUB, I; MOHAMMAD, I. **Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems**. London: CRC Press, 2005. Cap. 9, pp. 9.1-9.28.

[WAN05] WANG, S.; LIU, K. Z.; HU, F. P.. **Simulating Wireless Sensor Networks with OMNeT++**. In Proceedings of the 2nd International Conference on Mobile Technology. Guangzhou, China, 2005. pp. 1-6.

[WEC09] WeC Mote. Disponível em: <http://www-bsac.eecs.berkeley.edu/archive/users/hollar-seth/publications/cotsdust.pdf>. Acesso em: 23 mar. 2009.

[WIN01] Projeto WINS (*Wireless Integrated Network Sensors*). Disponível em: <http://ee.ucla.edu/~pottie/papers/nae_01.pdf>. Acesso em: 17 fev 2013.

[WOO97] WOOLDRIDGE, M.. Agent-based software engineering. **IEE Proceedings of Software Engineering**. Fev. 1997. Vol. 144, nº 1. pp. 26-37.

[WOO01] WOO, A.; COOLER, D. E.. **A Transmition Control Scheme for Media Access in Sensor Networks**. In Proceedings of 7th Annual Conference on Mobile Computing and Networking (MobiCom). Roma, Itália, 2001. pp. 221-235.

[WSN06] WSNet: An event-driven simulator for large scale wireless sensor network. Disponível em: <http://wsnet.gforge.inria.fr/>. Acesso em: 13 dez. 2012.

[WU_06] WU, X.; CHEN, G.; DAS, S. K.. **On the Energy Hole Problem of Nonuniform Node Distribution in Wireless Sensor Networks**. In Proceedings of the 3rd IEEE International Conference on Mobile Ad-Hoc and Sensor Systems. Vancouver, Canadá, 2006. pp. 180-187.

[XIA07] XIANGNING, F.; YJLIN, S.. **Improvement on LEACH Protocol of Wireless Sensor Network**. International Conference on Sensor Technologies and Applications. Valência, Espanha, 2007. pp. 260-264.

[XIA08] XIAN, X.; SHI, W.; HUANG, H.. **Comparison of OMNET++ and Other Simulation for WSN Simulation**. In Proceedings of the 3rd IEEE Conference on Industrial Electronics and Applications. Singapura, 2008. pp. 1439-1443.

[XUY06] XUYONG, H.; PEI, L.. **A Discrete Distributed Topology-control Algorithm of Wireless Sensor Network**. In Proceedings of the 3rd Annual International Conference on Mobile and Ubiquitous Systems: Networking & Services. San Jose, EUA, 2006. pp. 1-6.

[YEW87] YEW, P. C.; TZENG, N. F.; LAWRIE, D. H.. Distributing Hot Spot Addressing in Large Scale Multiprocessors. **IEEE Transactions on Computers**. Vol. C-36, nº 4. Abr. 1987. pp. 388-395.

[YI_03] YI, Y.; GERLA, M.; KWON, T.. **Efficient Flooding in Ad Hoc Networks Using On-demand Passive Cluster Formation**. In Proceedings of the 2nd Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net). Mahdia, Tunisia, 2003.

[ZAY07] ZAYANI, H. et al.. **Eco-MAC: An energy-efficient and low-latency hybrid MAC protocol for wireless sensor networks**. In Proceedings of the 2nd ACM International Workshop on Performance Monitoring, Measurement and Evaluation of Heterogeneous Wireless and Wired Networks. Ilha de Creta, 2007. pp. 68-71.

[ZEN98] ZENG, X. et al.. **GloMoSim: A Library for Parallel Simulation of Large-scale Wireless Networks**. In Proceedings of the 12th Workshop on Parallel and Distributed Simulation (PADS). Banff, Canadá, 1998. pp. 154-161.

[ZHA09] ZHAO, L. et al.. **Flooding and Directed Diffusion Routing Algorithm in Wireless Sensor Networks**. In Proceedings of the 9th International Conference on Hybrid Intelligent Systems. Shenyang, China, 2009. Vol. 2. pp. 235-239.

Apêndice A
Ponto Flutuante e BigDecimal

A conversão de números da base decimal (comumente utilizada na comunicação dos seres humanos e visualizada na tela do computador) para a base binária (utilizada no armazenamento e processamento computacional) produz erros representativos, uma vez que alguns números finitos na base decimal ao serem convertidos para base binária adquirem uma representação não finita. Este problema pode ser traduzido como um problema de arredondamento numérico, uma vez que a representação de um número não finito necessita de algum critério de arredondamento para representá-lo de forma finita e, assim, permitir que ele seja armazenado.

Os tipos de dados primitivos *float* e *double*, disponíveis na linguagem JAVA e em outras linguagens de programação, são utilizados na representação digital de valores numéricos reais. O padrão IEEE 754 [IEE08] define a formatação destes tipos primitivos em termos de ponto flutuante para números representados na forma binária, além de especificar as operações básicas entre números, conversões e as condições que podem gerar exceções ao lidar com esse tipo de representação numérica. A diferença entre o tipo *float* e o tipo *double* reside no fato do primeiro armazenar números utilizando 32 bits e o segundo 64 bits.

A representação de um número em ponto flutuante do padrão IEEE 754 é realizada utilizando fração, expoente e sinal, conforme ilustrado na Figura A.1 para um número representado com 32 bits (*float*). A Equação A.1 é utilizada para a conversão de números binários representados em ponto flutuantes de 32 bits do padrão IEEE 754 para números correspondentes na base decimal. Nesta equação, v é o número correspondente na base decimal, S é o sinal, e é o expoente e f é a fração.

$$v = (-1)^S \times 2^{(e-127)} \times (1+f) \quad \text{(A.1)}$$

O exemplo apresentado na Figura A.1 é a representação binária do número decimal 0,15625 utilizando o padrão IEEE 754 para ponto flutuante de 32 bits (*float*).

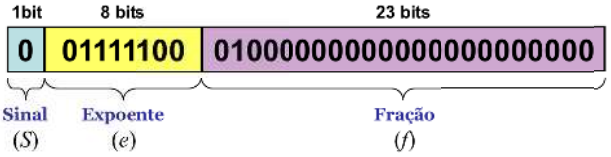


Figura A.1 – Representação binária do número decimal 0,15625 utilizando o tipo ponto flutuante *float* (32 bits), segundo a representação estabelecida no padrão IEEE 754. *S*=0; *e*=$2^6+2^5+2^4+2^3+2^2$=124; f=$(0\times2^{-1})+(1\times2^{-2})$=0,25; v=$(-1)^0\times2^{(124-127)}\times(1+0,25)$=0,15625

A principal motivação para se representar os números reais na forma de ponto flutuante em ambiente computacional está na rapidez com que as operações aritméticas são executadas. Contudo, os problemas de arredondamento numérico produzido tanto na conversão numérica decimal➔binária e binária➔decimal, assim como nos procedimentos das operações aritméticas, geram algumas inconsistências matemáticas.

Por exemplo, a soma dos números decimais 0,1 e 0,2 teria como resultado (obvio) o valor decimal 0,3. Contudo, o número decimal finito 0,1 quando convertido para representação binária torna-se o número não finito 0,000110011001... (vide procedimento para conversão de bases numéricas em [SAL06], Apêndice C), o mesmo ocorre para o número 0,2, que na base binária torna-se o número 0,00110011001..., também não finito. A soma desses dois números em um ambiente de processamento binário resulta em um valor não finito e a conversão deste resultado para a base decimal gera um número diferente do esperado. A soma dos números 0,1 e 0,2 representados como ponto flutuante resulta no valor "inesperado" de 0.30000000000000004.

A diferença no resultado das operações (de 4×10^{17} no exemplo da soma de 0,1 e 0,2) pode não significar muito em determinadas aplicações, mas em outras pode ser crucial, principalmente porque estes "resíduos" numéricos vão se acumulando aos números conforme são realizadas sucessivas operações aritméticas.

Um problema crítico que advém dessa inexatidão na representação binária são as comparações realizadas entre números. Expressões verdadeiras como `((0.1+0.2)==0.3)` resultam em falso (!), gerando erros de decisão nos comandos de desvio condicional e em estruturas de repetição.

A linguagem de programação JAVA oferece a classe *BigDecimal* [JAV12] como alternativa para contornar esse problema de representação do números decimais em sistemas binários. Além de a representação numérica ser feita na base decimal, isto é, armazenando os valores numéricos divididos em mantissa e expoente com base decimal ($mantissa \times 10^{expoente}$), esta classe permite um controle da forma com que o arredondamento numérico é realizado no caso de operações aritméticas que resultem em números não finitos, como no caso do resultado de 1 dividido por 3.

A classe *MathContext* [JAV12] pode ser utilizada na construção de objetos pertencentes à classe *BigDecimal*, fornecendo os parâmetros de precisão (número de dígitos que devem ser utilizados na representação numérica) e o modo de arredondamento (especifica o algoritmo que é utilizado no processo de arredondamento numérico). Alguns padrões foram definidos na classe *MathContext* para números de 32 bits/7 dígitos (*MathContext.DECIMAL32*), 64 bits/16 dígitos (*MathContext.DECIMAL64*) e 128 bits/32 dígitos (*MathContext.DECIMAL128*) a partir das especificações do padrão IEEE 754. Nestes três padrões o arredondamento é realizado pelo algoritmo do vizinho mais próximo, isto é, no caso do padrão de 128 bits, o número máximo de dígitos do número é 34, e, se o algarismo decimal do 35º dígito gerado por uma operação aritmética for menor que cinco, o algarismo do 34º dígito não se modifica; se o algarismo decimal do 35º dígito for maior ou igual a cinco, o algarismo do 34º dígito é incrementado de uma unidade. Esta forma de arredondamento está em acordo com o algoritmo de arredondamento *half-even*, também definido no padrão IEEE 754.

Além desses três padrões, outro padrão definido como ilimitado (*UNLIMITED*) pode ser utilizado para não haver restrições ao tamanho do número gerado pelas operações aritméticas. Em operações de soma, subtração e multiplicação isso não é problema, pois os resultados são sempre números finitos; mas se uma operação de divisão resultar em um número não finito (como no caso de 1 dividido por 3), uma exceção é levantada pela Máquina Virtual Java (JVM).

Uma particularidade dos métodos construtores da classe *BigDecimal* deve ser observada para que os números sejam representados de forma correta. Os construtores que levam como argumento parâmetros do tipo *float* ou *double* não devem ser utilizados, pois antes destes números serem carregados nos objetos criados da classe *BigDecimal*, eles já são pontos flutuantes, ou seja, eles já foram convertidos para a base binária e possivelmente se

tornaram números com representação não finita. Deve-se sempre utilizar o construtor com parâmetro do tipo *String*, que garante que o número será interpretado pela classe *BigDecimal* como de fato foi escrito pelo programador.

Assim, no exemplo apresentado na Figura A.2 o valor de `bd3`, resultado da soma de `bd1` com `bd2` na é 0,3000000000000000166533453693773481, enquanto que bd6, resultado da soma de `bd4` e `bd5`, é o valor correto de 0,3. Todos os objetos *BigDecimal* criados nesse exemplo, assim como as operações de soma, foram realizadas utilizando o contexto matemático *MathContext.DECIMAL128*.

```
BigDecimal bd1 = new BigDecimal(0.1, MathContext.DECIMAL128 );
BigDecimal bd2 = new BigDecimal(0.2, MathContext.DECIMAL128 );
BigDecimal bd3 = bd1.add( bd2, MathContext.DECIMAL128 );

BigDecimal bd4 = new BigDecimal("0.1", MathContext.DECIMAL128 );
BigDecimal bd5 = new BigDecimal("0.2", MathContext.DECIMAL128 );
BigDecimal bd6 = bd4.add( bd5, MathContext.DECIMAL128 );
```

Figura A.2 – Construção de objetos da classe BigDecimal utilizando o construtor que utiliza como parâmetro um tipo *double* (objetos db1 e db2) e o construtor que utiliza como parâmetro uma *String* (objetos db4 e db5). Deve-se evitar a utilização dos construtores que utilizam como parâmetros tipos *double* e *float* em detrimento do construtor que utiliza como parâmetro uma *String*, uma vez que a representação numérica dos primeiros pode não ser correta devido às conversões de base que ocorrem na representação de números no formato de ponto flutuante.

Apêndice B
Bloqueio e desbloqueio de *threads*

A conversão de números da base decimal (comumente utilizada na comunicação dos seres humanos e visualizada na tela do computador) para a base binária (utilizada no armazenamento e processamento computacional) produz erros representativos, uma vez que alguns números finitos na base decimal ao serem convertidos para base binária adquirem uma representação não finita. Este problema pode ser traduzido como um problema de arredondamento numérico, uma vez que a representação de um número não finito necessita de algum critério de arredondamento para representá-lo de forma finita e, assim, permitir que ele seja armazenado.

O bloqueio e o desbloqueio de *threads* na linguagem de programação Java, necessário à implementação do simulador, é apresentado de forma simplificada nesse apêndice. As classes *BlockControl* (Figura B.2) e *BlockThread* (Figura B.3), representam a classe de controle da simulação e a classe da *Thread* cujos comandos devem ser executados em pseudo-paralelismo, respectivamente. No exemplo apresentado, o controle do bloqueio e desbloqueio não é realizado baseado em um critério de tempo, como é feito na simulação, mas pela ordem de inserção das *BlockThreads* no objeto de controle. Assim, *BlockThreads* que

foram inseridas primeiro no controle são executadas primeiro, de forma sequencial até a última *BlockThread*.

O método *main* da Figura B.1, apresentado como exemplo de utilização das classes, inicializa um objeto *BlockControl* e insere 100 objetos do tipo *BlockThread* nesse controle.

```
01  public static void main( String args[] ) {
02        BlockControl bc = new BlockControl();
03        for ( int i=0; i<100; i++ )
04              bc.add( new BlockThread( "TH" + i ) );
05        bc.start();
06  }
```

Figura B.1 – Método *main* que inicializa o objeto *BlockControl* e insere 100 *BlockThreads* nesse objeto de controle.

O esquema de bloqueio e desbloqueio é realizado conforme o fluxograma apresentado na Figura 4.18. Considerando o critério temporal para realizar o desbloqueio das *threads*, cada objeto da classe *BlockThread* deveria possuir um atributo que marcaria o próximo tempo para o desbloqueio. A próxima *thread* a ser desbloqueada, portanto, seria a que possuísse o menor atributo próximo tempo.

O método *run()* (linha 24, Figura B.2) do objeto *BlockControl* é inicializado pela chamada do método *start()* do método *main* (linha 05, Figura B.1), inicializando todos os objetos da classe *BlockThread* inseridos no controle (linha 26, Figura B.2). A cada momento da execução apenas uma *thread* encontra-se desbloqueada, podendo ser um dos objetos da classe *BlockThread* ou o objeto de controle da classe *BlockControl*. Sempre que uma *thread* desbloqueia outra, ela deve bloquear-se, garantindo assim que duas *threads* não estejam em execução simultânea. O método *block()* e *unblock()* são idênticos nas classes *BlockThread* e *BlockControl*, garantindo a consistência nos processos de bloqueio e desbloqueio.

Inicialmente todos os objetos da classe *BlockThread* são bloqueados (linha 28, Figura B.3), sendo o processo de desbloqueio acionado pelo controle (linha 28, Figura B.2). O atributo *blocked* (definido na linha 05, Figura B.2 e linha 04, Figura B.3) é o semáforo binário utilizado como sinalizador de bloqueio e desbloqueio que evita que as *threads* sejam desbloqueadas de forma aleatória pela JVM.

O atributo *finished* (linha 05, Figura B.3) é utilizado para indicar que a *BlockThread* finalizou a sua execução (linha 30, Figura B.3). No exemplo apresentado esse estado nunca é atingido, pois o método de execução da classe *BlockThread* é executado indefinidamente (`while (true)` da linha 15, Figura B.3).

A execução do exemplo apresentado imprime na tela (linha 16, Figura B.3) o nome da *BlockThread* e um número que é acrescentado de uma unidade a cada desbloqueio da

BlockThread. A Figura B.4 apresenta um trecho de execução do programa da Figura B.1. A execução desse programa não possui um término definido, uma vez que tanto o controle como os objetos da *BlockThread* executam seus comandos em laço infinito.

```
01  import java.util.Vector;
02
03  public class BlockControl extends Object implements Runnable {
04
05        private volatile boolean blocked = false;
06        private Thread thread;
07        private Vector<BlockThread> threads;
08
09        public BlockControl() {
10              this.thread = new Thread( this );
11              this.threads = new Vector<BlockThread>();
12        }
13
14        public final void start() {
15              this.thread.start();
16        }
17
18        public void add( BlockThread thread ) {
19              thread.setControl( this );
20              this.threads.add( thread );
21        }
22
23        public final void run() {
24              synchronized ( this.thread ) {
25                    for ( BlockThread bt : this.threads )
26                          bt.start();
27                    while ( true ) {
28                          this.threads.firstElement().unblock();
29                          this.block();
30                          this.threads.add( this.threads.remove(0) );
31                    }
32              }
33        }
34
35        protected final void block() {
36              synchronized ( this.thread ) {
37                    this.blocked = true;
38                    while ( this.blocked ) {
39                          try {
40                                  if ( !this.finished )
41                                          this.thread.wait();
42                          } catch ( InterruptedException e ) { }
43                    }
44              }
45        }
46
47        protected final void unblock() {
48              synchronized ( this.thread ) {
49                    this.blocked = false;
50                    this.thread.interrupt();
51              }
52        }
53
54  }
```

Figura B.2 – Classe *BlockControl*.

```
01  public class BlockThread extends Object implements Runnable {
02
03        private BlockControl control;
04        private volatile boolean blocked = false;
05        private volatile boolean finished = false;
06        private Thread thread;
07
08        public BlockThread( String name ) {
09              this.thread = new Thread( this );
10              this.thread.setName( name );
11        }
12
13        public void exec() {
14              int c = 0;
15              while ( true ) {
16                    System.out.println( this.thread.getName() + " => [" + (++c) + "]" );
17                    this.control.unblock();
18                    this.block();
19              }
20        }
21
22        public final void start() {
23              this.thread.start();
24        }
25
26        public final void run() {
27              synchronized ( this.thread ) {
28                    this.block();
29                    this.exec();
30                    this.finished = true;
31                    this.unblock();
32                    this.control.unblock();
33              }
34        }
35
36        protected void setControl( BlockControl control ) {
37              this.control = control;
38        }
39
40        public String getName() {
41              return this.thread.getName();
42        }
43
44        protected final void block() {
45              synchronized ( this.thread ) {
46                    this.blocked = true;
47                    while ( this.blocked ) {
48                          try {
49                                if ( !this.finished )
50                                      this.thread.wait();
51                          } catch ( InterruptedException e ) { }
52                    }
53              }
54        }
55
56        protected final void unblock() {
57              synchronized ( this.thread ) {
58                    this.blocked = false;
59                    this.thread.interrupt();
60              }
61        }
62
63  }
```

Figura B.3 – Classe *BlockThread*.

```
TH0 => [1]
TH1 => [1]
TH2 => [1]
TH3 => [1]
TH4 => [1]
TH5 => [1]
TH6 => [1]
TH7 => [1]
TH8 => [1]
TH9 => [1]
TH10 => [1]
TH11 => [1]
TH12 => [1]
TH13 => [1]
TH14 => [1]
TH15 => [1]
TH16 => [1]
TH17 => [1]
TH18 => [1]
TH19 => [1]
TH20 => [1]
TH21 => [1]
TH22 => [1]
TH23 => [1]
TH24 => [1]
TH25 => [1]
TH26 => [1]
TH27 => [1]
TH28 => [1]
TH29 => [1]
TH30 => [1]
TH31 => [1]
TH32 => [1]
TH33 => [1]
TH34 => [1]
TH35 => [1]
TH36 => [1]
TH37 => [1]
TH38 => [1]
TH39 => [1]
TH40 => [1]
TH41 => [1]
TH42 => [1]
TH43 => [1]
TH44 => [1]
TH45 => [1]
TH46 => [1]
TH47 => [1]
TH48 => [1]
TH49 => [1]
```

```
TH50 => [1]
TH51 => [1]
TH52 => [1]
TH53 => [1]
TH54 => [1]
TH55 => [1]
TH56 => [1]
TH57 => [1]
TH58 => [1]
TH59 => [1]
TH60 => [1]
TH61 => [1]
TH62 => [1]
TH63 => [1]
TH64 => [1]
TH65 => [1]
TH66 => [1]
TH67 => [1]
TH68 => [1]
TH69 => [1]
TH70 => [1]
TH71 => [1]
TH72 => [1]
TH73 => [1]
TH74 => [1]
TH75 => [1]
TH76 => [1]
TH77 => [1]
TH78 => [1]
TH79 => [1]
TH80 => [1]
TH81 => [1]
TH82 => [1]
TH83 => [1]
TH84 => [1]
TH85 => [1]
TH86 => [1]
TH87 => [1]
TH88 => [1]
TH89 => [1]
TH90 => [1]
TH91 => [1]
TH92 => [1]
TH93 => [1]
TH94 => [1]
TH95 => [1]
TH96 => [1]
TH97 => [1]
TH98 => [1]
TH99 => [1]
```

```
TH0 => [2]
TH1 => [2]
TH2 => [2]
TH3 => [2]
TH4 => [2]
TH5 => [2]
TH6 => [2]
TH7 => [2]
TH8 => [2]
TH9 => [2]
TH10 => [2]
TH11 => [2]
TH12 => [2]
TH13 => [2]
TH14 => [2]
TH15 => [2]
TH16 => [2]
TH17 => [2]
TH18 => [2]
TH19 => [2]
TH20 => [2]
TH21 => [2]
TH22 => [2]
TH23 => [2]
TH24 => [2]
TH25 => [2]
TH26 => [2]
TH27 => [2]
TH28 => [2]
TH29 => [2]
TH30 => [2]
TH31 => [2]
TH32 => [2]
TH33 => [2]
TH34 => [2]
TH35 => [2]
TH36 => [2]
TH37 => [2]
TH38 => [2]
TH39 => [2]
TH40 => [2]
TH41 => [2]
TH42 => [2]
TH43 => [2]
TH44 => [2]
TH45 => [2]
TH46 => [2]
TH47 => [2]
TH48 => [2]
TH49 => [2]
```

```
TH50 => [2]
TH51 => [2]
TH52 => [2]
TH53 => [2]
TH54 => [2]
TH55 => [2]
TH56 => [2]
TH57 => [2]
TH58 => [2]
TH59 => [2]
TH60 => [2]
TH61 => [2]
TH62 => [2]
TH63 => [2]
TH64 => [2]
TH65 => [2]
TH66 => [2]
TH67 => [2]
TH68 => [2]
TH69 => [2]
TH70 => [2]
TH71 => [2]
TH72 => [2]
TH73 => [2]
TH74 => [2]
TH75 => [2]
TH76 => [2]
TH77 => [2]
TH78 => [2]
TH79 => [2]
TH80 => [2]
TH81 => [2]
TH82 => [2]
TH83 => [2]
TH84 => [2]
TH85 => [2]
TH86 => [2]
TH87 => [2]
TH88 => [2]
TH89 => [2]
TH90 => [2]
TH91 => [2]
TH92 => [2]
TH93 => [2]
TH94 => [2]
TH95 => [2]
TH96 => [2]
TH97 => [2]
TH98 => [2]
TH99 => [2]
```

```
TH0 => [3]
TH1 => [3]
TH2 => [3]
TH3 => [3]
TH4 => [3]
TH5 => [3]
TH6 => [3]
TH7 => [3]
TH8 => [3]
TH9 => [3]
TH10 => [3]
TH11 => [3]
TH12 => [3]
TH13 => [3]
TH14 => [3]
TH15 => [3]
TH16 => [3]
TH17 => [3]
TH18 => [3]
TH19 => [3]
TH20 => [3]
TH21 => [3]
TH22 => [3]
TH23 => [3]
TH24 => [3]
TH25 => [3]
TH26 => [3]
TH27 => [3]
TH28 => [3]
TH29 => [3]
TH30 => [3]
TH31 => [3]
TH32 => [3]
TH33 => [3]
TH34 => [3]
TH35 => [3]
TH36 => [3]
TH37 => [3]
TH38 => [3]
TH39 => [3]
TH40 => [3]
TH41 => [3]
TH42 => [3]
TH43 => [3]
TH44 => [3]
TH45 => [3]
TH46 => [3]
TH47 => [3]
TH48 => [3]
TH49 => [3]     ...
```

Figura B.4 – Trecho de execução do programa inicializado na Figura B.1.

Apêndice C

Exemplo de simulação utilizando o BaSS

A classe *TestThread.java* apresentada neste apêndice é a implementação das *threads* descritas em pseudocódigo na Figura 4.14. O código de execução (método *main(),* linha 14) seleciona os comandos que serão executados de acordo com o nome da thread: *Thread U*, *Thread V*, *Thread W* e *Thread X*, em conformidade com a Figura 4.14.

A simulação é criada no método estático *main()* (linhas 112 a 142), sendo controle da simulação é feito pelo objeto *ctrl* (linha 113). Foram inseridos na simulação um objeto da classe *TestLogger* (linha 114), subclasse de *BaSSLogger* (ver definição da classe *TestLogger.java* neste apêndice) para realizar o registro das atividades da simulação e um objeto da classe *TestUpdater* (linha 115 ou linha 116, comentadas; a primeira refere-se a um atualizador do tipo *EVENT_UPDATER* e a segunda a um atualizador do tipo *SIMULATION_TIME_UPDATER*).

As linhas 136 e 137 da classe da *TestThread.java* estão comentadas, mas teriam a funcionalidade de limitar a execução da simulação através do tempo máximo de execução (3 minutos e 20 segundos, linha 136) ou do tempo máximo de simulação (1,5 segundos, linha 137). Na linha 138 é inserido um atraso na simulação (atraso de 1 segundo), mas o mesmo é

desativado na linha 139.

Neste exemplo de simulação utilizando o BaSS, os comandos executados são apenas os registros (*logs*) de uma letra e um número que representariam a execução de um comando. Os *logs* realizados pela classe *TestLogger* imprimem na tela o tempo da simulação, a classe do objeto e o comando.

A execução (saída na tela) da simulação correspondente à classe *TestThread*, considerando os objetos da superclasse *BaSSUpdater* não presentes e depois considerando suas inclusões na simulação, são apresentadas no final deste apêndice.

TestThread.java

```
01  import br.eng.rsalustiano.bass.*;
02
03  public class TestThread extends BaSSThread {
04
05      public TestThread( String name, int priority ) {
06          super( name );
07          super.setPriority( priority );
08      }
09
10      public void init() {
11          super.getControl().log( this, "INIT" );
12      }
13
14      public void main() {
15
16          if ( super.getName().equals( "Thread U" ) ) {
17              super.getControl().log( this, "U1" );
18              super.getControl().log( this, "U2" );
19              super.wait( new BaSSStandardTime( "00:02:00" ) );
20              super.getControl().log( this, "U3" );
21              super.wait( new BaSSStandardTime( "00:03:00" ) );
22              super.getControl().log( this, "U4" );
23              super.getControl().getThreadByName( "Thread V" ).callInterruption( "intV1" );
24              super.wait( new BaSSStandardTime( "00:01:00" ) );
25              super.getControl().log( this, "U5" );
26              super.getControl().log( this, "U6" );
27              super.wait( new BaSSStandardTime( "00:03:00" ) );
28          }
29
30          if ( super.getName().equals( "Thread V" ) ) {
31              super.getControl().log( this, "V1" );
32              super.getControl().log( this, "V2" );
33              super.getControl().log( this, "V3" );
34              super.wait( new BaSSStandardTime( "00:01:00" ) );
35              super.getControl().log( this, "V4" );
36              super.wait( new BaSSStandardTime( "00:02:00" ) );
37              super.getControl().log( this, "V5" );
38              super.wait( new BaSSStandardTime( "00:01:00" ) );
39              super.getControl().log( this, "V6" );
40              super.getControl().log( this, "V7" );
41              super.wait( new BaSSStandardTime( "00:02:00" ) );
42              super.getControl().log( this, "V8" );
43              super.wait( new BaSSStandardTime( "00:02:00" ) );
44          }
45
46          if ( super.getName().equals( "Thread W" ) ) {
47              super.wait( new BaSSStandardTime( "00:03:00" ) );
48              super.getControl().log( this, "W1" );
49              super.getControl().log( this, "W2" );
50              super.wait( new BaSSStandardTime( "00:01:00" ) );
51              super.getControl().log( this, "W3" );
```

```
52            super.wait( new BaSSStandardTime( "00:02:00" ) );
53            super.getControl().log( this, "W4" );
54            super.getControl().log( this, "W5" );
55            super.wait( new BaSSStandardTime( "00:03:00" ) );
56            super.getControl().log( this, "W6" );
57            super.wait( new BaSSStandardTime( "00:01:00" ) );
58         }
59
60         if ( super.getName().equals( "Thread X" ) ) {
61            super.getControl().log( this, "X1" );
62            super.wait( new BaSSStandardTime( "00:05:00" ) );
63            super.getControl().log( this, "X2" );
64            super.getControl().getThreadByName( "Thread U" ).callInterruption( "intU1" );
65            super.wait( new BaSSStandardTime( "00:04:00" ) );
66            super.getControl().log( this, "X3" );
67            super.getControl().getThreadByName( "Thread V" ).callInterruption( "intV1" );
68            super.wait( new BaSSStandardTime( "00:01:00" ) );
69            super.getControl().log( this, "X4" );
70            super.wait( new BaSSStandardTime( "00:02:00" ) );
71         }
72
73      }
74
75      public void intU1() {
76         super.getControl().log( this, "U11" );
77         super.getControl().log( this, "U12" );
78         super.wait( new BaSSStandardTime( "00:01:00" ) );
79      }
80
81
82      public void intV1() {
83         super.getControl().log( this, "V11" );
84         super.wait( new BaSSStandardTime( "00:01:00" ) );
85         super.getControl().getThreadByName( "Thread U" ).callInterruption( "intU1" );
86         super.getControl().log( this, "V12" );
87         super.wait( new BaSSStandardTime( "00:01:00" ) );
88      }
89
90      public void timerV2() {
91         super.getControl().log( this, "V21" );
92         super.getControl().log( this, "V22" );
93         super.wait( new BaSSStandardTime( "00:01:00" ) );
94      }
95
96      public void timerW1() {
97         super.getControl().log( this, "W11" );
98         super.wait( new BaSSStandardTime( "00:01:00" ) );
99         super.getControl().log( this, "W12" );
100        super.wait( new BaSSStandardTime( "00:01:00" ) );
101     }
102
103     public void timerW2() {
104        super.getControl().log( this, "W21" );
105        super.wait( new BaSSStandardTime( "00:02:00" ) );
106     }
107
108     public void finalize() {
109        super.getControl().log( this, "FINALIZE" );
110     }
111
112     public static void main( String[] args ) throws Exception {
113        BaSSControl ctrl = new BaSSControl( BaSSStandardTime.class );
114        ctrl.addLogger( new TestLogger() );
115  //    ctrl.addUpdater( new TestUpdater( BaSSUpdater.Type.EVENT_UPDATER ) );
116  //    ctrl.addUpdater( new TestUpdater( BaSSUpdater.Type.SIMULATION_TIME_UPDATER ) );
117        TestThread ttU, ttV, ttW, ttX;
118        ttU = new TestThread( "Thread U", 0 );
119        ttV = new TestThread( "Thread V", 1 );
120        ttW = new TestThread( "Thread W", 2 );
121        ttX = new TestThread( "Thread X", 3 );
122
123        ttU.addInterruptionEvent( new BaSSInterruption( "intU1", "intU1", 1, true ) );
124
125        ttV.addInterruptionEvent( new BaSSInterruption( "intV1", "intV1", 2, true ) );
```

```
126         ttV.addInterruptionEvent( new BaSSTimer( "timerV2", "timerV2",
127                                   new BaSSStandardTime( "00:05:00" ), 1, true ) );


128         ttW.addInterruptionEvent( new BaSSTimer( "timerW1", "timerW1",
129                                   new BaSSStandardTime( "00:04:00" ), 1, true ) );
            ttW.addInterruptionEvent( new BaSSTimer( "timerW2", "timerW2",
130                                   new BaSSStandardTime( "00:08:00" ), 2, true ) );

131         ctrl.addThread( ttU );
132         ctrl.addThread( ttV );
133         ctrl.addThread( ttW );
134         ctrl.addThread( ttX );
135
136  //     ctrl.setWallclockTimeLimit( new BaSSStandardTime( "00:03:20.000" ) );
137  //     ctrl.setSimulationTimeLimit( new BaSSStandardTime( "00:00:01.500" ) );
138         ctrl.setDelay( new BaSSDelay( 1000, BaSSDelay.Type.EVENT_DELAY ) );
139         ctrl.disableDelay();
140         ctrl.start();
141      }
142  }
143
```

TestLogger.java

```
01  import br.eng.rsalustiano.bass.*;
02
03  public class TestLogger implements BaSSLogger {
04
05     public TestLogger() { }
06
07     public void init( BaSSControl control ) {
08        System.out.println( "[" + control.getSimulationTime().toString() +
09                            "][BaSSLogger][LOG INIT]" );
10     }
11
12     public void log( BaSSControl control, Object sender, Object... related ) {
13        String logString = "";
14        if ( sender instanceof BaSSUpdater ) {
15           logString = (String) related[0];
16           System.out.println( "[" + control.getSimulationTime().toString() +
17                               "][BaSSUpdater][" + logString + "]" );
18        }
19        if ( sender instanceof BaSSThread ) {
20           logString = (String) related[0];
21           System.out.println( "[" + control.getSimulationTime().toString() +
22                               "][BaSSThread][" + logString + "]" );
23        }
24     }
25
26     public void finalize( BaSSControl control ) {
27        System.out.println( "[" + control.getSimulationTime().toString() +
28                            "][BaSSLogger][LOG FINALIZE]" );
29     }
30  }
31
```

TestUpdater.java

```
01  import br.eng.rsalustiano.bass.*;
02
03  public class TestUpdater implements BaSSUpdater {
04
05     BaSSUpdater.Type updaterType;
06
07     public TestUpdater( BaSSUpdater.Type updaterType ) {
```

```
08          this.updaterType = updaterType;
09       }
10
11       public BaSSUpdater.Type getUpdaterType() {
12          return this.updaterType;
13       }
14
15       public void init( BaSSControl control ) {
16          control.log( this, "UPDATE INIT" );
17       }
18
19       public void update( BaSSControl control ) {
20          control.log( this, "UPDATE -> " + this.updaterType.toString() );
21       }
22
23       public void finalize( BaSSControl control ) {
24          control.log( this, "UPDATE FINALIZE" );
25       }
26
27    }
```

Resultado da execução da classe TestThread

```
[00:00:00.000][BaSSLogger][LOG INIT]
[00:00:00.000][BaSSThread][INIT]
[00:00:00.000][BaSSThread][INIT]
[00:00:00.000][BaSSThread][INIT]
[00:00:00.000][BaSSThread][INIT]
[00:00:00.000][BaSSThread][X1]
[00:00:00.000][BaSSThread][V1]
[00:00:00.000][BaSSThread][V2]
[00:00:00.000][BaSSThread][V3]
[00:00:00.000][BaSSThread][U1]
[00:00:00.000][BaSSThread][U2]
[00:01:00.000][BaSSThread][V21]
[00:01:00.000][BaSSThread][V22]
[00:02:00.000][BaSSThread][V4]
[00:02:00.000][BaSSThread][U3]
[00:03:00.000][BaSSThread][W21]
[00:04:00.000][BaSSThread][V5]
[00:05:00.000][BaSSThread][X2]
[00:05:00.000][BaSSThread][W11]
[00:05:00.000][BaSSThread][V21]
[00:05:00.000][BaSSThread][V22]
[00:05:00.000][BaSSThread][U11]
[00:05:00.000][BaSSThread][U12]
[00:06:00.000][BaSSThread][W12]
[00:06:00.000][BaSSThread][V6]
[00:06:00.000][BaSSThread][V7]
[00:06:00.000][BaSSThread][U4]
[00:07:00.000][BaSSThread][W1]
[00:07:00.000][BaSSThread][W2]
[00:07:00.000][BaSSThread][U5]
[00:07:00.000][BaSSThread][U6]
[00:08:00.000][BaSSThread][W21]
[00:08:00.000][BaSSThread][V11]
[00:09:00.000][BaSSThread][X3]
[00:09:00.000][BaSSThread][V12]
[00:10:00.000][BaSSThread][X4]
[00:10:00.000][BaSSThread][W11]
[00:10:00.000][BaSSThread][V21]
[00:10:00.000][BaSSThread][V22]
[00:10:00.000][BaSSThread][U11]
[00:10:00.000][BaSSThread][U12]
[00:11:00.000][BaSSThread][W12]
[00:11:00.000][BaSSThread][V8]
[00:11:00.000][BaSSThread][FINALIZE]
[00:12:00.000][BaSSThread][FINALIZE]
[00:12:00.000][BaSSThread][W11]
[00:13:00.000][BaSSThread][W12]
[00:13:00.000][BaSSThread][FINALIZE]
```

```
[00:14:00.000][BaSSThread][W3]
[00:16:00.000][BaSSThread][W21]
[00:18:00.000][BaSSThread][W11]
[00:19:00.000][BaSSThread][W12]
[00:20:00.000][BaSSThread][W11]
[00:21:00.000][BaSSThread][W12]
[00:22:00.000][BaSSThread][W4]
[00:22:00.000][BaSSThread][W5]
[00:25:00.000][BaSSThread][W21]
[00:27:00.000][BaSSThread][W11]
[00:28:00.000][BaSSThread][W12]
[00:29:00.000][BaSSThread][W6]
[00:30:00.000][BaSSThread][FINALIZE]
[00:30:00.000][BaSSLogger][LOG FINALIZE]
```

Resultado da execução da classe TestThread incluindo objetos da classe TestUpdater na simulação

```
[00:00:00.000][BaSSLogger][LOG INIT]
[00:00:00.000][BaSSUpdater][UPDATE INIT]
[00:00:00.000][BaSSUpdater][UPDATE INIT]
[00:00:00.000][BaSSThread][INIT]
[00:00:00.000][BaSSThread][INIT]
[00:00:00.000][BaSSThread][INIT]
[00:00:00.000][BaSSThread][INIT]
[00:00:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:00:00.000][BaSSThread][X1]
[00:00:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:00:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:00:00.000][BaSSThread][V1]
[00:00:00.000][BaSSThread][V2]
[00:00:00.000][BaSSThread][V3]
[00:00:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:00:00.000][BaSSThread][U1]
[00:00:00.000][BaSSThread][U2]
[00:00:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:00:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:01:00.000][BaSSThread][V21]
[00:01:00.000][BaSSThread][V22]
[00:01:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:01:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:02:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:02:00.000][BaSSThread][V4]
[00:02:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:02:00.000][BaSSThread][U3]
[00:02:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:02:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:03:00.000][BaSSThread][W21]
[00:03:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:03:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:04:00.000][BaSSThread][V5]
[00:04:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:04:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:05:00.000][BaSSThread][X2]
[00:05:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:05:00.000][BaSSThread][W11]
[00:05:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:05:00.000][BaSSThread][V21]
[00:05:00.000][BaSSThread][V22]
[00:05:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:05:00.000][BaSSThread][U11]
[00:05:00.000][BaSSThread][U12]
[00:05:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:05:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:06:00.000][BaSSThread][W12]
[00:06:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:06:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:06:00.000][BaSSThread][V6]
[00:06:00.000][BaSSThread][V7]
[00:06:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:06:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
```

```
[00:06:00.000][BaSSThread][U4]
[00:06:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:06:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:07:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:07:00.000][BaSSThread][W1]
[00:07:00.000][BaSSThread][W2]
[00:07:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:07:00.000][BaSSThread][U5]
[00:07:00.000][BaSSThread][U6]
[00:07:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:07:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:08:00.000][BaSSThread][W21]
[00:08:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:08:00.000][BaSSThread][V11]
[00:08:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:08:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:09:00.000][BaSSThread][X3]
[00:09:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:09:00.000][BaSSThread][V12]
[00:09:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:09:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:10:00.000][BaSSThread][X4]
[00:10:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:10:00.000][BaSSThread][W11]
[00:10:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:10:00.000][BaSSThread][V21]
[00:10:00.000][BaSSThread][V22]
[00:10:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:10:00.000][BaSSThread][U11]
[00:10:00.000][BaSSThread][U12]
[00:10:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:10:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:11:00.000][BaSSThread][W12]
[00:11:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:11:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:11:00.000][BaSSThread][V8]
[00:11:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:11:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:11:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:11:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:11:00.000][BaSSThread][FINALIZE]
[00:11:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:11:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:12:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:12:00.000][BaSSThread][FINALIZE]
[00:12:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:12:00.000][BaSSThread][W11]
[00:12:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:12:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:13:00.000][BaSSThread][W12]
[00:13:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:13:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:13:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:13:00.000][BaSSThread][FINALIZE]
[00:13:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:13:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:14:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:14:00.000][BaSSThread][W3]
[00:14:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:14:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:16:00.000][BaSSThread][W21]
[00:16:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:16:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:18:00.000][BaSSThread][W11]
[00:18:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:18:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:19:00.000][BaSSThread][W12]
[00:19:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:19:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:20:00.000][BaSSThread][W11]
[00:20:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:20:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:21:00.000][BaSSThread][W12]
[00:21:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
```

```
[00:21:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:22:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:22:00.000][BaSSThread][W4]
[00:22:00.000][BaSSThread][W5]
[00:22:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:22:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:25:00.000][BaSSThread][W21]
[00:25:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:25:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:27:00.000][BaSSThread][W11]
[00:27:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:27:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:28:00.000][BaSSThread][W12]
[00:28:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:28:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:29:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:29:00.000][BaSSThread][W6]
[00:29:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:29:00.000][BaSSUpdater][UPDATE -> SIMULATION_TIME_UPDATER]
[00:30:00.000][BaSSUpdater][UPDATE -> EVENT_UPDATER]
[00:30:00.000][BaSSThread][FINALIZE]
[00:30:00.000][BaSSUpdater][UPDATE FINALIZE]
[00:30:00.000][BaSSUpdater][UPDATE FINALIZE]
[00:30:00.000][BaSSLogger][LOG FINALIZE]
```

Apêndice D

Exemplo de simulação utilizando o WSN-BaSS

Os documentos XML, a programação das classes e os resultados das simulações apresentados neste apêndice referem-se à simulação descrita no item 5.7.1.

wsnA.xml

```
01  <?xml version="1.0" encoding="UTF-8"?>
02
03  <!DOCTYPE WSNSIMULATOR SYSTEM "wsnsimulator.dtd">
04
05  <WSNSIMULATOR propagation_model_class="wsntese.FriisPropagationModel" random_seed="1"
    allowed_network_node_classes="wsntese.SensorNode,wsntese.BaseStation">
06
07      <NETWORKNODE class="wsntese.BaseStation" name="BS01" x="-11.80" y="19.40"/>
08      <NETWORKNODE class="wsntese.SensorNode" name="SN01" x="45.32" y="-18.63"/>
09      <NETWORKNODE class="wsntese.SensorNode" name="SN02" x="122.55" y="79.89"/>
10      <NETWORKNODE class="wsntese.SensorNode" name="SN03" x="155.36" y="21.50"/>
11      <NETWORKNODE class="wsntese.SensorNode" name="SN04" x="87.84" y="1.96"/>
12      <NETWORKNODE class="wsntese.SensorNode" name="SN05" x="153.13" y="-26.27"/>
13      <NETWORKNODE class="wsntese.SensorNode" name="SN06" x="111.37" y="33.90"/>
14      <NETWORKNODE class="wsntese.SensorNode" name="SN07" x="22.68" y="20.41"/>
15      <NETWORKNODE class="wsntese.SensorNode" name="SN08" x="191.76" y="42.46"/>
16      <NETWORKNODE class="wsntese.SensorNode" name="SN09" x="78.80" y="-49.83"/>
17      <NETWORKNODE class="wsntese.SensorNode" name="SN10" x="229.59" y="22.68"/>
18
19      <UPDATER class="br.eng.rsalustiano.wsn.gui.impl.AnalysisFrame"/>
20      <THREAD class="wsntese.VariableTemperatureDynamic" name="Temperature"/>
21      <GENERIC class="br.eng.rsalustiano.wsn.gui.impl.StatisticLoggerFrame"/>
22
23  </WSNSIMULATOR>
```

wsnB.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE WSNSIMULATOR SYSTEM "wsnsimulator.dtd">

<WSNSIMULATOR propagation_model_class="wsntese.FriisPropagationModel" random_seed="1"
allowed_network_node_classes="wsntese.SensorNode,wsntese.BaseStation">

    <NETWORKNODE class="wsntese.BaseStation" name="BS01" x="-182.20" y="17.41"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN01" x="-29.67" y="-52.75"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN02" x="-118.28" y="56.35"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN03" x="-51.30" y="104.80"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN04" x="-16.75" y="-87.50"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN05" x="-140.90" y="40.65"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN06" x="-1.64" y="47.71"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN07" x="-39.65" y="-87.11"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN08" x="-79.39" y="77.96"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN09" x="-32.80" y="92.17"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN10" x="36.70" y="-91.60"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN11" x="0.30" y="-19.00"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN12" x="38.10" y="-18.80"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN13" x="108.30" y="-103.90"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN14" x="-27.37" y="2.70"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN15" x="-146.10" y="67.80"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN16" x="94.12" y="71.59"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN17" x="-161.20" y="-107.80"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN18" x="109.84" y="67.59"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN19" x="146.53" y="23.12"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN20" x="178.20" y="96.40"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN21" x="24.40" y="-42.10"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN22" x="-127.30" y="-11.70"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN23" x="-80.36" y="-47.96"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN24" x="57.49" y="20.57"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN25" x="14.74" y="-102.11"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN26" x="79.00" y="-94.40"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN27" x="-58.19" y="-31.21"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN28" x="-156.50" y="-17.80"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN29" x="116.95" y="-54.34"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN30" x="161.44" y="73.34"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN31" x="105.60" y="33.10"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN32" x="-11.30" y="101.50"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN33" x="-101.28" y="109.84"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN34" x="106.71" y="-32.38"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN35" x="-81.10" y="-8.70"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN36" x="-145.32" y="-79.73"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN37" x="69.31" y="38.90"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN38" x="-130.59" y="-45.06"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN39" x="15.10" y="-85.20"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN40" x="-69.77" y="40.77"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN41" x="24.50" y="100.30"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN42" x="104.01" y="101.91"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN43" x="140.58" y="-27.04"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN44" x="67.83" y="111.14"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN45" x="-101.96" y="74.92"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN46" x="-44.12" y="25.97"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN47" x="50.40" y="77.60"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN48" x="147.46" y="-74.32"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN49" x="25.29" y="17.43"/>
    <NETWORKNODE class="wsntese.SensorNode" name="SN50" x="-173.10" y="74.62"/>

    <OBSTACLE x1="-146.74" y1="16.46" x2="-5.53" y2="16.16" attenuation_in_dBm="-35.0"/>
    <OBSTACLE x1="72.66" y1="94.35" x2="91.03" y2="21.54" attenuation_in_dBm="-35.0"/>
    <OBSTACLE x1="-92.29" y1="131.07" x2="-35.92" y2="39.90"
     attenuation_in_dBm="-35.0"/>
    <OBSTACLE x1="-7.76" y1="-38.94" x2="-7.75" y2="-108.90"
     attenuation_in_dBm="-35.0"/>
    <OBSTACLE x1="21.37" y1="-0.32" x2="91.03" y2="21.22" attenuation_in_dBm="-35.0"/>

    <UPDATER class="br.eng.rsalustiano.wsn.gui.impl.AnalysisFrame"/>
    <THREAD class="wsntese.VariableTemperatureDynamic" name="Temperature"/>
    <GENERIC class="br.eng.rsalustiano.wsn.gui.impl.StatisticLoggerFrame"/>

</WSNSIMULATOR>
```

Antenna.java

```
01  package wsntese;
02
03  import br.eng.rsalustiano.wsn.WSNAntenna;
04
05  //Antenna W1030 (for 2.4GHz radios; Gain: 2.0dBi; omnidirecional)
06  public class Antenna implements WSNAntenna {
07
08      public double getGainIndBi() {
09         return 2.0;
10      }
11
12      public double getDirection() {
13         return 0.0;
14      }
15
16      public double getBeamwidth() {
17         return 360.0; //Omnidirecional
18      }
19
20  }
```

BaseStation.java

```
01  package wsntese;
02
03  import br.eng.rsalustiano.wsn.*;
04  import br.eng.rsalustiano.wsn.gui.*;
05  import br.eng.rsalustiano.bass.*;
06
07  import java.util.*;
08  import java.awt.Color;
09
10  //ATMega64L microcontroler time and consumption paramenters
11  public class BaseStation extends WSNBaseStation implements WSNGUINetworkNode  {
12
13      private Radio radio;
14
15      private static enum SensorNodeState { ACTIVE, SLEEP, OFF };
16
17      public BaseStation( String name ) {
18         super( name );
19         this.radio = new Radio( name + ".Radio", this );
20      }
21
22      public void setName( String name ) {
23         super.setName( name );
24         this.radio.setName( name + ".Radio" );
25      }
26
27      public WSNRadio getRadio() {
28         return this.radio;
26      }
27
28      public WSNThread[] getThreads() {
29         return new WSNThread[] { (WSNThread) this.radio };
30      }
31
32      public void init() {
33         try {
34            //Register interruption to receive messagem from radio
35            super.addInterruptionEvent( new BaSSInterruption( "receiveMessage",
                                                                "receiveMessage", 0, true ) );
36            super.setPriority( 2 );
37         } catch ( Exception e ) { }
38      }
39
```

```
40      public void main() {
41         this.radio.setState( Radio.RadioState.WOR );
42         super.waitForever();
43      }
44
45      public void finalize() {
46         super.getSimulator().log( this,  "FINALIZE" );
47      }
48
49      private Hashtable<String,Integer> msgs = new Hashtable<String,Integer>();
50      private Hashtable<String,String> msgsLast = new Hashtable<String,String>();
51      int counter = 0;
52
53      //Interruption to receive messagem from radio
54      public void receiveMessage() {
55         //250 clocks to perform this action
56         String msg = null;
57         while ( ( msg = (String) this.radio.getMessage() ) != null ) {
58            this.wait(  new BaSSTime( "0.0001" ) );
59            String m[] = msg.split( ":" );
60            //Count messages per sensor node
61            if ( msgs.get( m[0] ) != null )
62                  msgs.put( m[0], msgs.get( m[0] ) + 1 );
63            else
64                  msgs.put( m[0], 1 );
65            //Archieve message
66            msgsLast.put( m[0], "(id=" + String.format( Locale.US, "%04d",
                                        Integer.parseInt( m[1] ) ) + "):[" + m[2] + "]" );
67            counter++;
68         }
69      }
70
71      public WSNGUIShape getShape() {
72         return WSNGUIShape.SQUARE;
73      }
74
75      public Color getColor() {
76         return Color.GREEN;
77      }
78
79      public String getInformation() {
80         return Integer.toString( this.counter );
81      }
82
83      public String getReport() {
84         String ret = "";
85         String[] keys = (String[]) msgs.keySet().toArray( new String[0] );
86          Arrays.sort( keys );
87         for ( String n: keys )
88            ret += n + ":" + String.format( Locale.US, "%04d", msgs.get(n) ) + " last:" +
                                                                  msgsLast.get( n ) + "\n";
89         return ret;
90      }
91   }
```

Battery.java

```
01   package wsntese;
02
03   import br.eng.rsalustiano.wsn.WSNBattery;
04
05   //CR2032 Battery Model (240mAh; 1.5% discharge per year)
06   public class Battery implements WSNBattery {
07
08      private static final double START_CHARGE_IN_mAh = 240.0;
09      private static final double START_CHARGE_IN_COULOMB =
                                                    ( START_CHARGE_IN_mAh * 3600.0  );
10      private static final double AUTO_DISCHARGE_PERCENT_PER_YEAR = 1.5;
11      private static final double RETAIN_FACTOR_PER_YEAR =
                                    ( 1.0 - ( AUTO_DISCHARGE_PERCENT_PER_YEAR / 100.0 ) );
```

```
12      private static final double SECONDS_PER_YEAR = ( 356.0 * 24.0 * 3600.0 );
13
14      private double batteryChargeInCoulombs = START_CHARGE_IN_COULOMB;
15
16      public double getRemainingCoulombs() {
17         return this.batteryChargeInCoulombs;
18      }
19
20      public boolean isEmpty() {
21         return ( this.batteryChargeInCoulombs == 0.0 );
22      }
23
24      public void use( double currentInAmpere, double timeInSeconds ) {
25         //Auto discharge
26         double new_charge = this.batteryChargeInCoulombs * Math.pow(
                                RETAIN_FACTOR_PER_YEAR, timeInSeconds / SECONDS_PER_YEAR );
27         //Used battery
28         new_charge = new_charge - ( currentInAmpere * timeInSeconds );
29         this.batteryChargeInCoulombs = ( new_charge < 0.0 ) ? 0.0 : new_charge;
30      }
31
32      public double getRemainingPercent() {
33         return ( this.batteryChargeInCoulombs / START_CHARGE_IN_COULOMB ) * 100.0;
34      }
35
36   }
```

FriisPropagationModel.java

```
01   package wsntese;
02
03   import br.eng.rsalustiano.wsn.WSNPropagationModel;
04   import br.eng.rsalustiano.wsn.WSNRadio;
05   import br.eng.rsalustiano.wsn.WSNSimulator;
06
07   // Friis Law Propagation Model
08   public class FriisPropagationModel implements WSNPropagationModel {
09
10      public double getRxPowerInMilliWatt( WSNSimulator simulator, WSNRadio radioTx,
11                                           WSNRadio radioRx, double distanceInMeters ) {
12         double f = radioTx.getFrequencyInHz();
13         double spaceLoss = Math.pow( ( ( 299792458 / f ) / ( 4 * Math.PI *
14                            distanceInMeters ) ), 2 );
15         double P_Tx_mW = Math.pow( 10, ( radioTx.getTxPowerIndBm() / 10 ) );
16   double gain_Tx_mW = Math.pow( 10, radioTx.getAntenna().getGainIndBi()/10 );
17      double gain_Rx_mW = Math.pow( 10, radioRx.getAntenna().getGainIndBi()/10 );
18      return ( gain_Tx_mW * gain_Rx_mW * spaceLoss * P_Tx_mW );
19      }
20
21   }
```

Radio.java

```
001  package wsntese;
002
003  import br.eng.rsalustiano.wsn.*;
004  import br.eng.rsalustiano.bass.*;
005  import java.util.Vector;
006
007  //Radio CC2500 characteristics
008  public class Radio extends WSNThread implements WSNRadio {
009
010     public static enum RadioState { TX, RX, SLEEP, WOR, OFF };
011     public static final String errorMessage = new String( "ERROR" );
012
013     private Antenna antenna = new Antenna();
014     public RadioState state = RadioState.OFF;
```

```
015     private String buffer = null;
016
017     private Vector<String> txQueue = new Vector<String>();
018     private Vector<String> rxQueue = new Vector<String>();
019
020     private WSNNetworkNode networkNode;
021
022     public Radio( String name, WSNNetworkNode networkNode ) {
023        super( name );
024        this.networkNode = networkNode;
025     }
026
027     public double getActualConsumptionInAmpere() {
028        switch ( this.state ) {
026           case TX      : return 11.1E-6; //11.1 uA
027           case RX      : return 15.0E-6; //13,3 a 16,6 uA
028           case WOR     : return 900E-12; //900 nA
029        }
030        return 0.0;
031     }
032
033     public int getTxSize() {
034        return this.txQueue.size();
035     }
036
037     public void init() {
038        try {
039           //Register interruption to receive message
040           super.addInterruptionEvent( new BaSSInterruption( "receiveMessage",
                                                                "receiveMessage", 1, true ) );
041           //Register interruption to transmit message
042           super.addInterruptionEvent( new BaSSInterruption( "transmitMessage",
                                                                "transmitMessage", 0, true ) );
043        } catch ( Exception e ) { }
044     }
045
046     public void main() {
047        super.waitForever();
048     }
049
050     public void finalize() {
051        this.setState( RadioState.OFF );
052     }
053
054     public WSNAntenna getAntenna() {
055        return antenna;
056     }
057
058     public boolean isCompatibleWith( WSNRadio radio ) {
059        return ( radio.getFrequencyInHz() == this.getFrequencyInHz() );
060     }
061
062     public double getFrequencyInHz() {
063        return 2.4E9;
064     }
065
066     public double getTxPowerIndBm() {
067        return -12.0;
068     }
069
070     public double getSensitivityIndBm() {
071        return -83.0;
072     }
073
074     public void setState( RadioState state ) {
075        if ( state == RadioState.TX )
076           super.setPriority( 2 );
077        else
078           super.setPriority( 1 );
079        this.state = state;
080     }
081
082     public RadioState getState() {
083        return this.state;
```

```
084      }
085
086      public boolean isReceiving() {
087         return ( this.state == RadioState.WOR ) || ( this.state == RadioState.RX );
088      }
089
090      public boolean isTransmitting() {
091         return ( this.state == RadioState.TX );
092      }
093
094      public Object getBuffer() {
095         return new String( this.buffer.toString() );
096      }
097
098      public void putBuffer( Object msg ) {
099         this.buffer = new String( msg.toString() );
100      }
101
102      public void flushBuffer() {
103         if ( ( this.state == Radio.RadioState.RX ) &&
104              ( !this.buffer.toString().equals( Radio.errorMessage ) ) )
105            this.rxQueue.add( new String( this.buffer.toString() ) );
106         this.buffer = null;
107         if ( ( this.state == RadioState.RX ) && super.isWaitingSignal() )
108            super.signal();
109      }
110
111      //Tells the simulator (WSNSimulator) which interruption to call when
112       // receive a message
113      public void callInterruptionToReceiveMessage() {
114         super.callInterruption( "receiveMessage" );
115      }
116
117      //Interruption to transmit a message
118      public void transmitMessage() {
119         while ( !this.txQueue.isEmpty() ) {
120            //Time to go from WOR to TX state
121            this.wait( new BaSSTime( "0.000809" ) );
122            this.setState( RadioState.TX );
123            String msg = new String( this.txQueue.remove(0) );
124            this.putBuffer( msg );
125            super.getSimulator().startTransmission( this );
126            //Time to transmit a message of 32Bytes(+preamble, synch, CRC) @250kbps
127            this.wait( new BaSSTime( "0.001536" ) );
128            super.getSimulator().endTransmission( this );
129         }
130
131        //Time to go from TX to WOR state
132         this.wait(  new BaSSTime( "0.000721" ) );
133         this.setState( RadioState.WOR );
134      }
135
136      //Interruption to receive message
137      public void receiveMessage() {
138         //Time to go from WOR to RX state
139         this.wait( new BaSSTime( "0.000809" ) );
140         this.setState( RadioState.RX );
141         this.waitSignal();
142         if ( !this.rxQueue.isEmpty() ) {
143            this.wait( new BaSSTime( "0.005" ) );
144            this.networkNode.callInterruption( "receiveMessage" );
145         }
146        //Time to go from RX to WOR state
147         this.wait( new BaSSTime( "0.000721" ) );
148         this.setState( RadioState.WOR );
149      }
150
151      public void putMessage( String msg ) {
152         this.txQueue.add( msg );
153      }
154
155      public String getMessage() {
156         if ( this.rxQueue.isEmpty() )
157            return null;
```

```
158         return this.rxQueue.remove( 0 );
159      }
160
161      public void modifyBufferWithAnErroneousInstance() {
162         this.buffer = new String( Radio.errorMessage );
163      }
164   }
```

SensorNode.java

```
001   package wsntese;
002
003   import br.eng.rsalustiano.wsn.*;
004   import br.eng.rsalustiano.wsn.gui.*;
005   import br.eng.rsalustiano.bass.*;
006
007   import java.util.*;
008   import java.awt.Color;
009
010   //ATMega64L microcontroler time and consumption paramenters
011   public class SensorNode extends WSNSensorNode implements WSNGUINetworkNode  {
012
013      private static enum SensorNodeState { ACTIVE, IDLE, OFF };
014
015      private Radio radio;
016      private Battery battery;
017      private SensorTemperatura sensor;
018      private SensorNodeState state = SensorNodeState.OFF;
019
020      public SensorNode( String name ) {
021         super( name );
022         this.radio = new Radio( name + ".Radio", this );
023         this.battery = new Battery();
024         this.sensor = new SensorTemperatura( name + ".Sensor" );
025      }
026
027      public void setName( String name ) {
028         super.setName( name );
026         this.radio.setName( name + ".Radio" );
027         this.sensor.setName( name + ".Sensor" );
028      }
029
030      public WSNRadio getRadio() {
031         return this.radio;
032      }
033
034      public WSNSensor[] getSensors() {
035         return new WSNSensor[] { this.sensor };
036      }
037
038      public WSNBattery getBattery() {
039         return this.battery;
040      }
041
042      public WSNThread[] getThreads() {
043         return new WSNThread[] { this.radio, this.sensor };
044      }
045
046      public double getActualConsumptionInAmpere() {
047         double proc = 0.0;
048         switch ( this.state ) {
049            case ACTIVE  : proc = 5.0E-3; break;       //5.0 mA
050            case IDLE    : proc = 2.5E-3; break;       //2.5 mA
051         }
052         return proc + this.radio.getActualConsumptionInAmpere() +
                        this.sensor.getActualConsumptionInAmpere();
053      }
054
055      private int id = 0; //Message ID
056      private int counterTra = 0; //Transmit message counter
057      private int counterRec = 0; //Receive message counter
```

```
058     private int counterRet = 0; //Retransmit message counter
059
060     public void init() {
061        try {
062           //Interruption to receive a message from radio
063           super.addInterruptionEvent( new BaSSInterruption( "receiveMessage",
                                                              "receiveMessage", 1, true ) );
064           //Timer to send a menssage every 15 minutes
065           super.addInterruptionEvent( new BaSSTimer( "timer", "timer",
                                       new BaSSStandardTime( "00:15:00.000" ), 0, true ) );
066        } catch ( Exception e ) { e.printStackTrace(); }
067        super.setPriority( 2 );
068     }
069
070     public void main() {
071        this.wait( new BaSSTime( "0.005" ) );
072        this.radio.setState( Radio.RadioState.WOR );
073
074        this.wait( new BaSSTime( "0.001" ) );
075        this.setState( SensorNodeState.IDLE );
076
077        this.wait( new BaSSTime( "0.001" ) );
078     }
079
080     //Timer to send a message (every 15 minutes)
081     public void timer() {
082        this.wait( new BaSSTime( "0.00001" ) );
083        this.setState( SensorNodeState.ACTIVE );
084        this.sensor.callInterruption( "sense" );
085
086        //Wait to read the temperature
087        super.wait( new BaSSStandardTime( "00:00:01.000" ) );
088        double temp = this.sensor.getValue();
089
090        // Transmit a message if temperature less than 26°C
091        if ( temp < 26.0 ) {
092           this.wait( new BaSSTime( "0.001" ) );
093           counterTra++;
094           String msg = super.getName() + ":" + Integer.toString( ++id ) + ":" +
                                                              Double.toString( temp );
095           this.requestTransmission( msg );
096        }
097        this.setState( SensorNodeState.IDLE );
098     }
099
100    private void requestTransmission( String msg ) {
101       this.wait( new BaSSTime( "0.005" ) );
102       this.radio.putMessage( msg );
103       this.wait( new BaSSTime( "0.005" ) );
104       this.radio.callInterruption( "transmitMessage" );
105    }
106
107    private void setState( SensorNodeState state ) {
108       this.state = state;
109    }
110
111    public void finalize() {
112       super.getSimulator().log( this,  "FINALIZE (" + String.format( Locale.US,
                              "%.2f%%", this.getBattery().getRemainingPercent() ) + "%)" );
113    }
114
115    Hashtable<String,Integer> rec = new Hashtable<String,Integer>();
116    //Interruption to receive the message
117    public void receiveMessage() {
118       counterRec++;
119       this.state = SensorNodeState.ACTIVE;
120       String msg = null;
121       this.wait( new BaSSTime( "0.005" ) );
122       while ( ( msg = (String) this.radio.getMessage() ) != null ) {
123          // Verify if the message needs to be retransmited
124          String[] pmsg = msg.split( ":" );
125          String msgNode = pmsg[0];
126          if ( msgNode.equals( super.getName() ) )
127                continue;
```

```
128           int msgID = Integer.parseInt( pmsg[1] );
129           if ( rec.get( msgNode ) == null ) {
130                 rec.put( msgNode, msgID );
131           } else {
132                 int k = rec.get( msgNode ).intValue();
133                 if ( msgID <= k )
134                       continue;
135                 rec.put( msgNode, msgID );
136           }
137           this.wait( new BaSSTime( "0.005" ) );
138           counterRet++;
139           //if the ID of received message for a specific node name is
140           //greater than the last received, than retransmit the message
141           this.requestTransmission( msg );
142        }
143        this.wait( new BaSSTime( "0.001" ) );
144         this.setState( SensorNodeState.IDLE );
145     }
146
147     public WSNGUIShape getShape() {
148        return WSNGUIShape.CIRCLE;
149     }
150
151     public Color getColor() {
152        return Color.BLUE;
153     }
154
155     public String getInformation() {
156        return Integer.toString( this.counterRec ) + "/" +
                                    Integer.toString( this.counterTra+this.counterRet );
157     }
158
159     public String getReport() {
160        String rep = "Radio.State = " + this.radio.state.toString() + "\n\n";
161        rep += "Transmited...: " + counterTra + "\n";
162        rep += "Received.....: " + counterRec + "\n";
163        rep += "Retransmited.: " + counterRet + "\n";
164        rep += "Radio Queue..: " + this.radio.getTxSize() + "\n";
165        rep += "Messages:\n";
166        for ( String node: this.rec.keySet() )
167           rep += node + " : " + this.rec.get( node ) + "\n";
168        return rep;
169     }
170
171  }
```

SensorTemperature.java

```
01   package wsntese;
02
03   import br.eng.rsalustiano.wsn.*;
04   import br.eng.rsalustiano.bass.*;
05
06   //SHT11 parameters
07   public class SensorTemperatura extends WSNThread implements WSNSensor {
08
09      //Time cost to read the temperature
10      private static final BaSSTime READ_TEMPERATURE_TIME = new BaSSTime( "0.320" );
11
12      public static enum SensorState { SLEEP, MEASURING, OFF };
13      public double buffer = 0.0;
14      public SensorState state = SensorState.OFF;
15
16      public SensorTemperatura( String name ) {
17         super( name );
18      }
19
20      public Class<?> getSensorVariableClass() {
21         return VariableTemperature.class;
22      }
23
```

```
24      public void setState( SensorState state ) {
25         this.state = state;
26      }
27
28      public void init() {
29         try {
30            //Register interruption to read temperature
31            super.addInterruptionEvent( new BaSSInterruption( "sense", "sense", 1, true ));
32         } catch ( Exception e ) { }
33      }
34
35      public void main() {
36         this.setState( SensorState.SLEEP );
37         super.waitForever();
38      }
39
40      //Interruption to read temperature
41      public void sense() {
42         this.setState( SensorState.MEASURING );
43         this.wait( READ_TEMPERATURE_TIME );
44         this.buffer = super.getSimulator().getVariableValue(
                                              VariableTemperatureDynamic.class, this );
45         this.setState( SensorState.SLEEP );
46      }
47
48      public double getValue() {
49         return this.buffer;
50      }
51
52      public void finalize() {
53      }
54
55      public double getActualConsumptionInAmpere() {
56         switch ( this.state ) {
57            case SLEEP:          return 1.5E-6; //1.5 uA
58            case MEASURING:      return 1.0E-3; //1 mA
59         }
60         return 0.0;
61      }
62
63   }
```

VariableTemperatureDynamic.java

```
01   package wsntese;
02
03   import br.eng.rsalustiano.wsn.*;
04   import br.eng.rsalustiano.bass.*;
05
06   //Variable Temperature between 25 and 27°C - Dynamic
07   public class VariableTemperatureDynamic extends WSNThread implements WSNVariable {
08      private double x1=20, y1=50, x2=200, y2=-20, s1=100, s2=40;
09
10      public VariableTemperatureDynamic( String name ) {
11         super( name );
12      }
13
14      public double getMinValue() {
15         return 25.0;
16      }
17
18      public double getMaxValue() {
19         return 27.0;
20      }
21
22      public String getUnit() {
23         return "°C";
24      }
25
26      private double gaussian3D( double a, double x0, double y0, double rox,
```

```
                                    double roy, double x, double y ) {
27          return a * Math.exp( -( ( Math.pow( ( x - x0 ), 2 ) / ( 2 * rox * rox ) ) +
                                   ( Math.pow( ( y - y0 ), 2 ) / ( 2 * roy * roy ) ) ) );
28       }
29
30       public double getValue( WSNSimulator simulator, WSNPosition position ) {
31          double a2 = ( this.getMaxValue() - this.getMinValue() ) / 4;
32          double v = this.getMinValue() +
33                     this.gaussian3D( 3*a2, this.x1, this.y1, this.s1, this.s1, position.x,
                                        position.y ) + this.gaussian3D( a2, this.x2, this.y2,
                                        this.s2, this.s2, position.x, position.y );
34          return ( (int) ( 10 * v ) ) / 10.0;
35       }
36
37       public void init() { }
38
39       //Dynamic movement. The movement occurs randomly between 5 and 10 minutes;
40       //The two gaussian position and their apertures change randomdy from -1m to 1m
41       public void main() {
42          while ( true ) {
43             super.wait( super.getSimulator().getRandom().nextBaSSTime(
                                              new BaSSStandardTime( "00:05:00.000" ),
                                              new BaSSStandardTime( "00:10:00.000" ) ) );
44             this.x1 += super.getSimulator().getRandom().nextDouble( -1.0, 1.0 );
45             this.y1 += super.getSimulator().getRandom().nextDouble( -1.0, 1.0 );
46             this.s1 += super.getSimulator().getRandom().nextDouble( -1.0, 1.0 );
47             this.x2 += super.getSimulator().getRandom().nextDouble( -1.0, 1.0 );
48             this.y2 += super.getSimulator().getRandom().nextDouble( -1.0, 1.0 );
49             this.s2 += super.getSimulator().getRandom().nextDouble( -1.0, 1.0 );
50          }
51       }
52
53       public void finalize() { }
54
55       public boolean isNoise() {
56          return false;
57       }
58    }
```

Resultado da Simulação A (*wsnA.xml*)

```
-------------------------------------------------------------------------------------------------
 Network Statistic at 8y05m06d18:51:41.636 (Simulation Time)
 Simulation State: COMPLETED
-------------------------------------------------------------------------------------------------
 Node   Battery  # Tx   # Rx [OK/Coll./Noise/Lost Conn.]   Time of death   Death sequence
-------------------------------------------------------------------------------------------------
 BS01               0  [695088/0/0/0]
 SN01    0.00%  684119  [1413849/64839/0/0]             7y01m04d01:56:06.927         4
 SN02    0.00%  684139  [684170/0/0/0]                  6y11m13d13:45:00.000         3
 SN03    0.00%  740305  [1414027/101822/0/0]            7y04m01d22:08:58.765         5
 SN04    0.00%  728101  [1336097/158891/0/0]            7y06m07d15:56:37.833         7
 SN05    0.00%  641151  [618684/0/0/0]                  6y04m18d14:51:51.373         2
 SN06    0.00%  728703  [1466668/105626/0/0]            7y04m13d16:39:35.821         6
 SN07    0.00%  695088  [675851/0/0/0]                  7y06m27d14:28:43.544         8
 SN08    0.00%  733212  [1223674/26333/0/0]             8y05m06d18:51:41.636        10
 SN09    0.00%  500017  [807927/49100/0/0]              4y09m20d13:02:16.650         1
 SN10    0.00%  749174  [729336/0/0/0]                  8y00m18d01:53:50.350         9
-------------------------------------------------------------------------------------------------
```

Resultado da Simulação B (*wsnB.xml*)

```
-------------------------------------------------------------------------------------------------
 Network Statistic at 9y07m17d20:59:43.783 (Simulation Time)
 Simulation State: COMPLETED
-------------------------------------------------------------------------------------------------
 Node   Battery  # Tx   # Rx [OK/Coll./Noise/Lost Conn.]   Time of death   Death sequence
-------------------------------------------------------------------------------------------------
 BS01              0  [628687/25573/0/0]
 SN01    0.00%  693018  [1848100/464688/0/0]            8y07m19d09:58:14.662        41
 SN02    0.00%  369317  [787933/221306/0/0]             3y03m09d20:13:27.259         1
 SN03    0.00%  395664  [662484/111398/0/0]             6y10m29d14:23:19.319        16
 SN04    0.00%  538558  [853922/34548/0/0]              7y04m06d03:15:00.000        20
 SN05    0.00%  227311  [538564/218565/0/0]             4y03m20d12:18:54.670         9
 SN06    0.00%  710079  [1622681/175214/0/0]            8y08m23d01:19:23.648        42
 SN07    0.00%  528680  [845039/33647/0/0]              3y05m11d21:23:27.164         2
 SN08    0.00%  381769  [786032/151398/0/0]             5y02m13d01:21:12.544        12
 SN09    0.00%  461622  [1044026/79686/0/0]             3y08m23d03:13:44.782         3
 SN10    0.00%  750735  [1727173/204074/0/0]            8y07m04d09:14:24.440        39
 SN11    0.00%  752089  [1978865/313679/0/0]            8y02m21d11:53:56.646        31
 SN12    0.00%  570625  [1000476/65117/0/0]             4y02m09d06:45:00.000         8
 SN13    0.00%  642939  [1312699/173723/0/0]            7y06m23d15:53:28.826        25
 SN14    0.00%  591554  [1634403/324151/0/0]            7y05m00d12:00:00.000        23
 SN15    0.00%  281841  [588421/218445/0/0]             8y05m19d11:14:55.099        38
 SN16    0.00%  561331  [1431807/169605/0/0]            7y04m24d09:11:51.876        22
 SN17    0.00%  493691  [508167/0/0/0]                  9y03m27d01:23:22.494        47
 SN18    0.00%  536662  [1301634/213732/0/0]            7y02m25d11:44:44.781        17
 SN19    0.00%  770143  [1474982/98386/0/0]             8y07m09d07:00:00.000        40
 SN20    0.00%  532693  [547388/0/0/0]                  5y02m02d22:18:08.577        11
 SN21    0.00%  756237  [1750909/215649/0/0]            8y01m00d03:30:00.000        29
 SN22    0.00%  512462  [1105609/97149/0/0]             6y05m03d22:45:00.000        15
 SN23    0.00%  545781  [1284772/227939/0/0]            7y04m22d01:00:00.000        21
 SN24    0.00%  529568  [926646/48397/0/0]              4y00m25d22:08:26.731         6
 SN25    0.00%  570346  [914146/101910/0/0]             7y05m20d16:15:00.000        24
 SN26    0.00%  767900  [1497672/169805/0/0]            8y02m00d05:15:00.000        30
 SN27    0.00%  637109  [1510199/260629/0/0]            8y03m01d13:40:25.232        33
 SN28    0.00%  466447  [794135/76479/0/0]              5y10m28d11:30:00.000        14
 SN29    0.00%  728932  [1939434/290837/0/0]            7y11m25d01:00:00.000        28
 SN30    0.00%  741360  [1391237/73959/0/0]             8y11m28d21:19:58.572        45
 SN31    0.00%  563322  [1191332/127406/0/0]            4y04m23d03:59:50.52         10
 SN32    0.00%  633181  [1408102/137064/0/0]            8y03m24d03:56:37.437        34
 SN33    0.00%  299695  [365619/187144/0/0]             9y07m17d20:59:43.783        50
 SN34    0.00%  574965  [957995/81135/0/0]              4y02m03d17:26:26.156         7
 SN35    0.00%  634713  [1422113/208537/0/0]            7y11m03d11:29:27.34         26
 SN36    0.00%  515849  [892250/17022/0/0]              9y04m04d08:26:19.808        48
 SN37    0.00%  718720  [1467508/89056/0/0]             8y04m28d08:57:59.714        35
 SN38    0.00%  523848  [1296717/140129/0/0]            8y10m03d10:15:00.000        43
 SN39    0.00%  574599  [1247539/125298/0/0]            4y00m05d06:21:28.550         5
 SN40    0.00%  606644  [931828/305934/0/0]             8y11m25d03:00:00.000        44
 SN41    0.00%  505231  [1124178/96354/0/0]             5y02m14d09:50:58.179        13
 SN42    0.00%  520724  [1110745/140701/0/0]            7y03m00d03:38:49.263        18
 SN43    0.00%  780345  [1684879/222148/0/0]            8y05m17d01:30:00.000        37
 SN44    0.00%  673428  [1467933/220275/0/0]            8y02m27d11:50:34.909        32
 SN45    0.00%  379858  [919601/231722/0/0]             3y11m06d17:00:00.000         4
 SN46    0.00%  670195  [1135960/20632/0/0]             9y00m07d18:24:49.114        46
 SN47    0.00%  682478  [1427078/105697/0/0]            8y05m02d01:00:00.000        36
 SN48    0.00%  601030  [1213734/177967/0/0]            7y03m06d22:34:36.759        19
 SN49    0.00%  709026  [1893892/302369/0/0]            7y11m18d03:25:16.603        27
 SN50    0.00%  224966  [259427/117774/0/0]             9y06m22d02:00:00.000        49
-------------------------------------------------------------------------------------------------
```

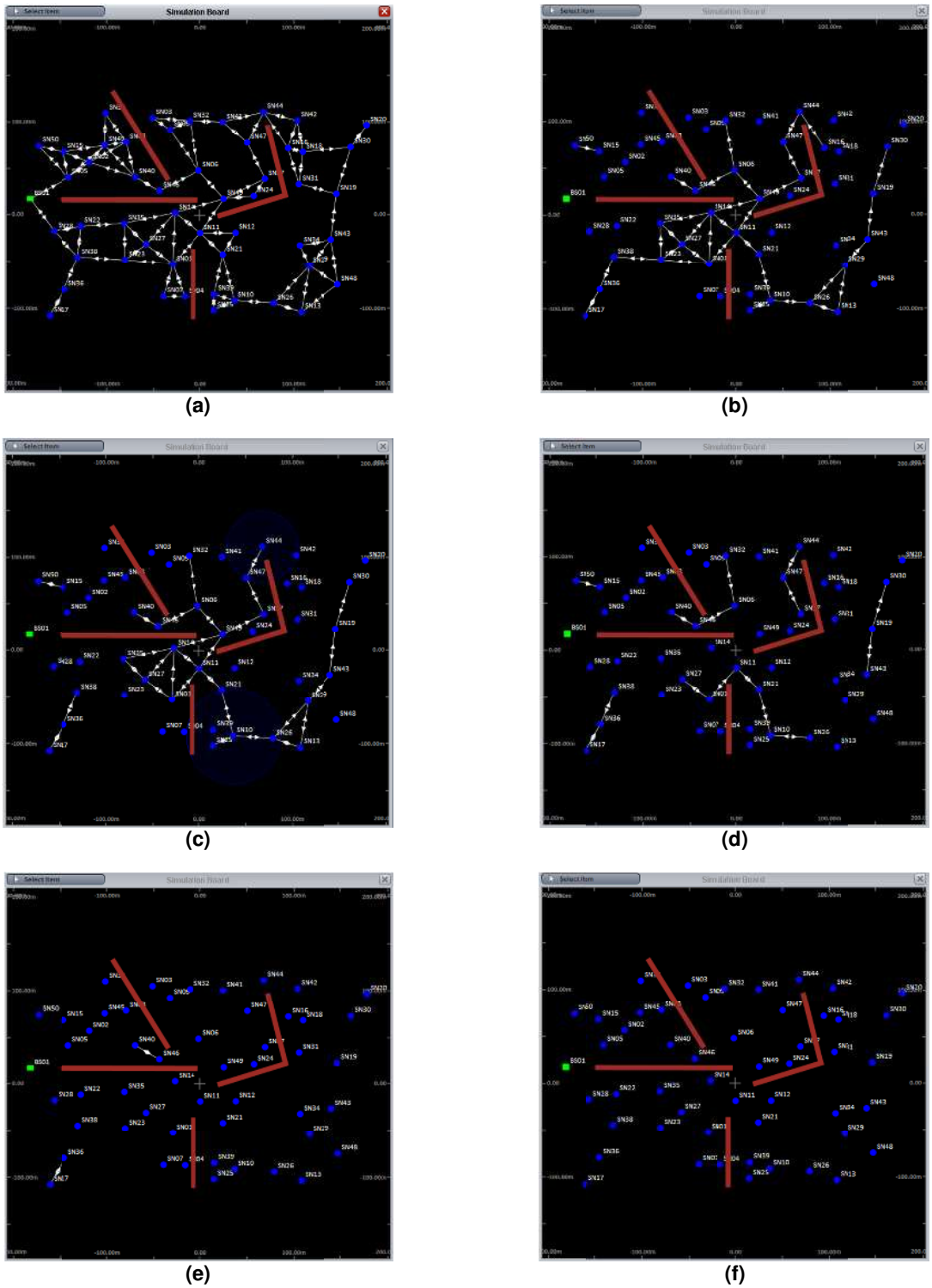


Figura D.1 – Imagens evidenciando a perda de conexão da rede em decorrência da "morte" dos nós durante a execução da Simulação B. (a) início da simulação (00y00m00d00:00:00.000).; (b) No momento 07y04m09d00:00:01.001. (c) No momento 07y04m28d20:15:01.048. (d) No momento 07y11m28d18:15:00.006. (e) No momento 08y10m10d05:30:01.041. (f) No final da simulação (09y07m17d20:59:43.783).