Universidade Federal do Rio de Janeiro

Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais

Paulo Soares da Costa

Um estudo sobre as redes de sensores sem fio

Rio de Janeiro

2014

Paulo Soares da Costa

Um estudo sobre as redes de sensores sem fio

Monografia apresentada para obtenção do título de Especialista em Gerência de Redes de Computadores no Curso de Pós-Graduação Lato Sensu em Gerência de Redes de Computadores e Tecnologia Internet do Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro – NCE/UFRJ.

Orientador:

Claudio Miceli de Farias, D.Sc., UFRJ, Brasil

Rio de Janeiro

2014

Paulo Soares da Costa

Um estudo sobre as redes de sensores sem fio

Monografia apresentada para obtenção do título de Especialista em Gerência de Redes de Computadores no Curso de Pós-Graduação Lato Sensu em Gerência de Redes de Computadores e Tecnologia Internet do Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro – NCE/UFRJ.

Aprovada em março de 2014.

Claudio Miceli de Farias, D.Sc., UFRJ, Brasil

Dedico este trabalho, em especial, à minha família, que sempre acreditou em meu potencial, e sempre me apoiou, incondicionalmente, em todos os momentos, dando-me assim, um forte motivo para sempre progredir.

AGRADECIMENTOS

Aos meus mestres, sem exceção, que me motivaram a cada vez exceder meus limites, a buscar sempre a excelência, e dar o melhor de mim.

Também aos meus colegas de turma, que com sua alegria e cooperação, tornaram este curso muito mais interessante, além, é claro, do próprio conteúdo em si, que já é uma motivação enorme.

Em especial, gostaria de agradecer ao meu irmão, por todo apoio, amizade, carinho e disponibilidade, ajudando a reescrever minha história, sendo fundamental na minha vida acadêmica.

RESUMO

COSTA, Paulo Soares da. **Um estudo sobre as redes de sensores sem fio**. Monografia (Especialização em Gerência de Redes e Tecnologia Internet). Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro. Rio de Janeiro, 2014.

O presente trabalho sintetiza uma visão das Redes de Sensores sem Fio. Apresenta desde os características gerais, aplicações, plataformas de hardware, sistemas operacionais e ferramentas de desenvolvimento, abordando vantagens e desvantagens decorrentes da escolha entre estas características, que podem ser adequadas ou não, dependendo do ambiente onde irá ser utilizada uma Rede de Sensores sem Fio.

ABSTRACT

COSTA, Paulo Soares da. **Um estudo sobre as redes de sensores sem fio**. Monografia (Especialização em Gerência de Redes e Tecnologia Internet). Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro. Rio de Janeiro, 2014.

This paper summarizes an overview of Wireless Sensor Networks. Presents from the general characteristics, applications, hardware platforms, operating systems and development tools, addressing advantages and disadvantages of choosing between these features, that may be appropriate or not, depending on the environment where a wireless sensor network will be used.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SMEM	Sistemas Micro-Elétrico-Mecânicos
RAM	Random Access Memory
Sec	Sistema em Chip
RSSF	Redes de Sensores Sem Fio
I/O	Input/Output
MAC	Media Access Control
FIFO	First-In-First-Out
MSB	Modular Sensor Board
MPU	Micro Processor Unit
USB	Universal Serial Bus
FCC	Federal Communications Community
CSMA	Carrier Sense Multiple Access
CSMA/CA	Carrier Sense Multiple Access with Colision Avoidance
ADC	Analog to Digital Converter
DAC	Digital to Analog Converter
EEPROM	Electrically Erasable Programmable Read-Only Memory
PDA	Personal Digital Assistant
UART	Universal Assyncronous Receiver/Transmitter
SRAM	Static Random Access Memory
FSK	Frequency Shift Keying
SPI	Serial Peripheral Interface
ISM	Industrial, Scientific and Medical
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos
MMX	MultiMedia eXtensions
DSP	Digital Signal Processor
6LoWPAN	IPv6 Over Low Power Wireless Personal Network
Ipv6	Internet Protocol version 6
OTAP	Over The Air Programming
Sun SPOT	Sun Small Programmable Object Technology
WPAN	Wireless Personal Area Networks
SO	Sistema Operacional
API	Application Programming Interface
CMOS	Complementary Metal-Oxide-Semiconductor
NSMSF	Nós Sensores de Multimídia Sem Fio
RSMSF	Redes de Sensores de Multimídia Sem Fio
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
TCB	Thread Control Block
EDF	Earliest Deadline First
ROM	Read Only Memory
ICMP	Internet Control Message Protocol
IP	Internet Protocol
MLD	Multicast Listener Discovery
TCB	Task Control Block
JDK	Java Development Kit

SUMÁRIO

1 **INTRODUÇÃO**

Os avanços na tecnologia de sensores baseada em sistemas micro-elétrico-mecânicos (SMEM) têm levado ao desenvolvimento de nós sensores cada vez menores e mais baratos, capazes de comunicação sem a utilização de fios, de fazer sensoriamento e executar algumas instruções computacionais. Um nó sensor sem fio é composto de micro-controlador, transceptor, relógio, memória RAM, fonte de alimentação, rádio, atuador e conversor analógico-digital.

Os recursos principais e mais críticos são: a energia (que é tipicamente provida por uma bateria), e memória principal muito limitada, que frequentemente permite armazenar apenas poucos kilobytes. Uma fonte de energia secundária que captura energia do meio ambiente, como painéis solares, pode ser adicionada ao nó, dependendo do ambiente onde o sensor for colocado. Dependendo da aplicação e do tipo de sensores utilizados, atuadores podem ser incorporados aos sensores.

O micro-controlador utilizado em um nó sensor sem fio opera em baixa frequência comparado a unidades de processamento tradicionais, como os processadores de desktop e notebooks. Estes sensores de recursos limitados, são um exemplo impressionante de um sistema em chip (SeC). A utilização massiva de nós sensores é necessária para alcançar alta qualidade e tolerância a falhas em redes de sensores sem fio (RSSFs).

A partir do fato de que os nós sensores são colocados tipicamente em locais de difícil acesso, um rádio é implementado para comunicação sem fio, para transferir os dados para uma estação base (por exemplo, um dispositivo pessoal de mão, ou um ponto de acesso de uma infraestrutura física).

Ao contrário de redes tradicionais, uma RSSF tem sua própria limitação de projeto e recursos. A limitação de recursos inclui uma limitada quantidade de energia, comunicação de curto alcance, baixa largura de banda e armazenamento e processamento limitado em cada nó. As limitações de projeto são dependentes da aplicação e são baseadas no ambiente monitorado. O ambiente desempenha um papel fundamental na determinação do tamanho da rede, o esquema de implantação e a topologia da rede. O tamanho da rede varia com o ambiente monitorado. Para ambientes fechados, poucos nós são requeridos para formar uma rede em um espaço limitado, enquanto que um ambiente aberto pode requerer mais nós para cobrir uma área maior.

Uma RSSF tipicamente tem pequena ou nenhuma infraestrutura. Isto consiste no número de nós sensores (de dezenas a milhares) trabalhando juntos para monitorar uma região, para obter dados sobre o ambiente. Existem dois tipos de RSSFs: com infraestrutura e sem infraestrutura. Uma RSSF sem infraestrutura é a que contém uma densa coleção de nós sensores. Nós sensores podem ser colocados em uma maneira *ad hoc* (numa distribuição *ad hoc*, os nós sensores podem ser aleatoriamente distribuídos no campo) no campo. Uma vez distribuída, a rede não é capaz de fazer funções de monitoramento e notificação. Em uma RSSF sem infraestrutura, a manutenção da rede, como gerenciamento da conectividade, e detecção de falhas é difícil porque existem muitos nós. Em uma RSSF com infraestrutura, todos ou alguns nós sensores são colocados de forma pré-planejada. A vantagem de uma rede com infraestrutura é que poucos nós podem ser colocados com baixo índice de manutenção da rede, e custo de gerenciamento. Poucos nós podem ser colocados para prover cobertura, enquanto a distribuição *ad hoc* pode ter regiões sem cobertura.

Uma implantação *ad hoc* é preferida em vez de uma implantação pré-planejada, quando um ambiente é inacessível por humanos ou quando a rede é composta de centenas a milhares de nós. Obstruções no ambiente podem também limitar a comunicação entre nós, que por sua vez, afetam a conectividade da rede (ou topologia).

Sensores de diversos tipos, mecânicos, térmicos, biológicos, químicos, ópticos e magnéticos podem ser conectados ao nó sensor para medir propriedades do meio ambiente. Os nós sensores são utilizados para monitorar diversos fenômenos naturais, bem como os feitos pelo homem, como por exemplo, o monitoramento do clima, através da medição de temperaturas, umidade, pressão, etc; monitoramento da vida selvagem, ao controlar animais de espécies em extinção, bem como áreas de proibição de caça, por ser áreas de reprodução; monitoramento de pacientes, com doenças difíceis de tratar, facilitando a pesquisa e diagnóstico dos médicos; monitoramento e controle de processos industriais, monitoramento de tropas em combate, através da implantação de chips na pulseira de cada soldado, para que se saiba se a tropa está unida, ou se há alguém precisando de auxílio, ou seqüestrado, etc; monitoramento de automóveis, através da instalação de diversos sensores nos carros, que reportará quando esse ou aquele componente está ao final de sua vida útil, ou sensores que funcionam como tacômetro, que enviam informações à uma empresa de transportes, dos seus carros que trafegaram além da velocidade permitida, gerando informações sobre responsabilidade sobre multas, etc, para citar alguns.

Pesquisas em RSSFs buscam encontrar limitações introduzindo novos conceitos de projeto, criando ou melhorando protocolos existentes, construindo

novas aplicações, e desenvolvendo novos algoritmos. As áreas de aplicação para sensores estão crescendo e novas aplicações para redes de sensores estão surgindo rapidamente.

Neste estudo, vamos examinar algumas características das redes de sensores sem fio, como: aplicações, arquitetura, plataformas, incluindo detalhes sobre placas de sensores e micro controladores, sistemas operacionais e ferramentas de desenvolvimento.

Este trabalho busca mostrar as características principais de uma RSSF, num panorama atual, citando, por exemplo, plataformas recentes como o *SunSpot*, e sistemas operacionais recentes como *Contiki* e *LiteOS*. No fim deste estudo, abordaremos também as tendências para as redes de sensores sem fio.

O trabalho se divide como a seguir: No capítulo 2, falaremos sobre a caracterização de uma RSSF. Aplicações de uma RSSF será abordada no capítulo 3. No capítulo 4, serão abordadas algumas das diversas plataformas para RSSF. No capítulo 5, citaremos alguns dos sistemas operacionais para RSSF, desde os mais populares, e robustos, até alguns voltados exclusivamente para determinada plataforma de hardware, e/ou com foco em economia de recursos, como memória e energia. No capítulo 6, falaremos sobre as ferramentas de desenvolvimento para RSSF, e finalmente, o capítulo 7 conclui este trabalho.

2 **CARACTERIZAÇÃO DAS REDES DE SENSORES SEM FIO**

A classificação de uma RSSF depende de seu objetivo e área de aplicação. A aplicação influenciará diretamente nas funções exercidas pelos nós da rede, assim como na arquitetura desses nós (processador, memória, dispositivos sensores, fonte de energia, transceptor), na quantidade de nós que compõem a rede, na distribuição inicialmente planejada para a rede, no tipo de deposição dos nós no ambiente, na escolha dos protocolos da pilha de comunicação, no tipo de dado que será tratado, no tipo de serviço que será provido pela rede e, conseqüentemente, no tempo de vida dessa rede.

De acordo com [36], as RSSFs podem ser classificadas segundo a configuração (ver tabela 1), o sensoriamento (ver tabela 2) e segundo o tipo de comunicação (ver tabelas 3 e 4). Uma RSSF também pode ser diferente segundo o tipo de processamento que executa (ver tabela 5).

O potencial de observação e controle do mundo real permite que as RSSFs se apresentem como uma solução para diversas aplicações: monitoramento ambiental, gerenciamento de infraestrutura, biotecnologia, monitoramento e controle industrial, segurança pública e de ambientes em geral, áreas de desastres e risco para vidas humanas, transporte, medicina e controle militar [37].

A visão é que RSSFs se tornem disponíveis em todos os lugares executando as tarefas mais diferentes possíveis [38]. Este potencial tem estimulado ainda mais o desenvolvimento de hardware e software para RSSFs e atraído a atenção da comunidade acadêmica.

Como mencionado antes, uma RSSF é um tipo de sistema dependente da aplicação. Qualquer projeto ou solução proposta para estas redes deve levar em consideração os requisitos da aplicação a ser desenvolvida, as características e restrições dos componentes dos nós sensores, assim como as características do ambiente onde tais redes serão aplicadas.

Tabela 1 - Caracterização das RSSF, segundo a configuração

Configuração		
Composição	Homogênea	Rede composta de nós que apresentam a mesma capacidade de hardware. Eventualmente os nós podem executar software diferente.
	Heterogênea	Rede composta por nós com diferentes capacidades de hardware.
Organização	Hierárquica	RSSF em que os nós estão organizados em grupos (*clusters*). Cada grupo terá um líder (*cluster-head*)que poderá ser eleito pelos nós comuns. Os grupos podem organizar hierarquias entre si.
	Plana	Rede em que os nós não estão organizados em grupos
Mobilidade	Estacionária	Todos os nós sensores permanecem no local onde foram depositados durante todo o tempo de vida da rede.
	Móvel	Rede em que os nós sensores podem ser deslocados do local onde inicialmente foram depositados.
Densidade	Balanceada	Rede que apresenta uma concentração e distribuição de nós por unidade de área considerada ideal segundo a função objetivo da rede.
	Densa	Rede que apresenta uma uma alta concentração de nós por unidade de área.
	Esparsa	Rede que apresenta uma baixa concentração de nós por unidade de área.
Distribuição	Irregular	Rede que apresenta uma distribuição não uniforme dos nós na área monitorada.
	Regular	Rede que apresenta uma distribuição uniforme de nós sobre a área monitorada

Tabela 2 - Caracterização das RSSF, segundo o sensoriamento

Sensoriamento		
Coleta	Periódica	Os nós sensores coletam dados sobre o(s) fenômeno(s) em intervalos regulares. Um exemplo são as aplicações que monitoram o canto dos pássaros. Os sensores farão a coleta durante o dia e permaneceram desligados durante a noite.
	Contínua	Os nós sensores coletam os dados continuamente. Um exemplo são as aplicações de exploração interplanetária que coletam dados continuamente para a formação de base de dados para pesquisas.
	Reativa	Os nós sensores coletam dados quando ocorrem eventos de interesse ou quando solicitado pelo observador. Um exemplo são as aplicações que detectam a presença de objetos na área monitorada.
	Tempo Real	Os nós sensores coletam a maior quantidade de dados possível no menor intervalo de tempo. Um exemplo são aplicações que envolvem risco para vidas humanas tais como aplicações em escombros ou áreas de desastres. Um outro exemplo são as aplicações militares onde o dado coletado é importante na tomada de decisão e definição de estratégias.

Tabela 3 - Caracterização das RSSF, segundo a comunicação (Parte A)

Classificação segundo a Comunicação		
Disseminação	Programada	Os nós disseminam em intervalos regulares.
	Contínua	Os nós disseminam os dados continuamente.
	Sob Demanda	Os nós disseminam os dados em resposta à consulta do observador e à ocorrência de eventos.
Tipo Conexão	Simétrica	Todas as conexões existentes entre os nós sensores, com exceção do nó sorvedouro têm o mesmo alcance.
	Assimétrica	As conexões entre os nós comuns têm alcance diferente.
Transmissão	Simplex	Os nós sensores possuem transceptor que permite apenas transmissão da informação.
	Half-duplex	Os nós sensores possuem transceptor que permite transmitir ou receber em um determinado instante.
	Full-duplex	Os nós sensores possuem transceptor que permite transmitir ou receber dados ao mesmo tempo.

Tabela 4 - Caracterização das RSSF, segundo a comunicação (Parte B)

Classificação segundo a Comunicação		
Alocação de Canal	Estática	Neste tipo de rede se existirem "n" nós, a largura de banda é dividida em "n" partes iguais na freqüência (FDMA – *Frequency Division Multiple Access*), no tempo (TDMA – *Time Division Multiple Access*), no código (CDMA – Code Division Multiple Access), no espaço (SDMA – *Space Division Multiple Access*) ou ortogonal (OFDM – *Orthogonal Frequency Division Multiplexing*). A cada nó é atribuída uma parte privada da comunicação, minimizando interferência.
	Dinâmica	Neste tipo de rede não existe atribuição fixa de largura de banda. Os nós disputam o canal para comunicação dos dados.
Fluxo de Informação	*Flooding*	Neste tipo de rede, os nós sensores fazem *broadcast* de suas informações para seus vizinhos que fazem *broadcast* desses dados para outros até alcançar o ponto de acesso. Esta abordagem promove um alto *overhead* mas está imune às mudanças dinâmicas de topologia e a alguns ataques de impedimento de serviço (DoS – *Denial of Service*).
	Multicast	Neste tipo de rede os nós formam grupos e usam o *multicast* para comunicação entre os membros do grupo.
	Unicast	Neste tipo de rede, os nós sensores podem se comunicar diretamente com o ponto de acesso usando protocolos de roteamento multi-saltos.
	Gossiping	Neste tipo de rede, os nós sensores selecionam os nós para os quais enviam os dados.
	Bargaining	Neste tipo de rede, os nós enviam os dados somente se o nó destino manifestar interesse, isto é, existe um processo de negociação.

Tabela 5 - Caracterização das RSSF, segundo o processamento

Classificação segundo o Processamento		
Cooperação	Infra-estrutura	Os nós sensores executam procedimentos relacionados à infra-estrutura da rede como por exemplo, algoritmos de controle de acesso ao meio, roteamento, eleição de líderes, descoberta de localização e criptografia.
	Localizada	Os nós sensores executam além dos procedimentos de infra-estrutura, algum tipo de processamento local básico como por exemplo, tradução dos dados coletado pelos sensores baseado na calibração.
	Correlação	Os nós estão envolvidos em procedimentos de correlação de dados como fusão, supressão seletiva, contagem, compressão, multi-resolução e agregação.

3 **APLICAÇÕES DE REDES DE SENSORES SEM FIO**

Redes de sensores consistem em diferentes tipos de sensores como sísmicos, magnéticos de baixa taxa de amostragem, termais, visuais, infravermelhos, acústicos, e de radar, que estão aptos a monitorar uma grande variedade de condições ambientais que incluem as seguintes:

- Temperatura;
- Umidade;
- Movimento veicular;
- Condição de luminosidade;
- Pressão;
- Composição do solo;
- Níveis de ruído;
- Presença ou ausência de certos tipos de objetos;
- Níveis de estresse mecânico em objetos conectados;
- Características atuais como velocidade, direção e tamanho de um objeto.

Nós sensores podem ser utilizados para sensoriamento contínuo, detecção de eventos, identificador de eventos, sensoriamento de local e controle local de atuadores. A concepção de microssensoriamento e conexão sem fio destes nós prometem muitas áreas de aplicações novas. Categorizam-se as aplicações em: militares, ambientais, de saúde, domésticas e outras áreas comerciais. É possível

expandir esta classificação em mais categorias como exploração espacial, processamento químico e recuperação de desastres.

3.1 APLICAÇÕES MILITARES

As redes de sensores sem fio podem ser uma parte integral de sistemas militares de comando, controle, reconhecimento, e direcionamento (C4ISRT). As características de rápido desenvolvimento, auto-organização, e tolerância a falhas das redes de sensores fazem delas uma técnica de sensoriamento muito promissora para C4ISRT militares. Uma vez que as redes de sensores baseiam-se na implantação densa de nós descartáveis e de baixo custo, a destruição de alguns nós por ações hostis, não afeta uma operação militar tanto quanto a destruição de um nó tradicional, que torna o conceito de redes de sensores uma melhor abordagem para campos de batalha. Algumas das aplicações militares de redes de sensores são: monitoramento de exércitos aliados, seu equipamento e munição; vigilância do campo de batalha, reconhecimento de exércitos e terrenos inimigos; direcionamento; avaliação de danos de batalha; e reconhecimento e detecção de ataque biológico e químico.

3.1.1 **Monitoramento de exércitos aliados, seus equipamentos e munição**

Líderes e comandantes podem constantemente monitorar o status de tropas aliadas, as condições e disponibilidade do equipamento e a munição no campo de batalha pelo uso de redes de sensores. Qualquer tropa, veículo, equipamento e munição crítica pode ter sensores pequenos conectados para relatar o status. Estes relatórios são reunidos em nós sorvedouros e enviados para os líderes da tropa. Os dados também podem ser encaminhados para os níveis superiores da hierarquia de comando, ao serem agregados com os dados de outras unidades em cada nível.

3.1.2 **Vigilância do campo de batalha**

Terrenos críticos, rotas de aproximação, caminhos e estreitos, podem ser rapidamente cobertos com redes de sensores, e observados de perto pelas atividades de forças inimigas. À medida que as operações se desenvolvem e novos planos operacionais são preparados, novas redes de sensores podem ser implantadas a qualquer tempo para vigilância do campo de batalha.

3.1.3 **Reconhecimento de forças e terrenos inimigos**

Redes de sensores podem ser depositadas em terrenos críticos, e alguma inteligência valiosa, detalhada e temporal sobre as forças e terrenos inimigos, pode ser colhida em minutos, antes que as forças inimigas possam interceptá-las.

3.1.4 **Direcionamento**

Redes de sensores podem ser incorporadas em sistemas de orientação de munição inteligente.

3.1.5 **Avaliação de danos de batalha**

Logo antes ou após dos ataques, as redes de sensores podem ser implantadas na área alvo para colher os dados de avaliação de danos de batalha.

3.1.6 **Detecção e reconhecimento de ataque nuclear, biológico e químico**

Na guerra química e biológica, por estar próximo do marco zero, é importante para a detecção temporal e precisa dos agentes. Redes de sensores implantadas na região aliada, e usadas como sistemas de alerta químico e biológico, podem prover às tropas aliadas um tempo de reação crítico, que diminui as casualidades

drasticamente. Também se pode usar as redes de sensores para reconhecimento detalhado após a detecção de um ataque NBC. Por exemplo, é feito um reconhecimento nuclear sem expor uma equipe de reconhecimentos à radiação nuclear.

3.2 APLICAÇÕES AMBIENTAIS

Algumas aplicações ambientais de redes de sensores incluem: rastreamento dos movimentos dos pássaros, pequenos animais, e insetos; monitoramento de condições ambientais que afetam a agricultura e a pecuária; irrigação; macro instrumentos para monitoramento terrestre em larga escala e exploração planetária; detecção químico-biológica; agricultura de precisão; monitoramento biológico, terrestre e ambiental nos contextos marítimo, do solo e atmosféricos; pesquisas meteorológicas ou geofísicas; mapeamento da biocomplexidade do ambiente e estudo da poluição.

3.2.1 **Detecção de incêndio florestal**

Desde que os nós sensores podem ser depositados estrategicamente, aleatoriamente e densamente numa floresta, os nós sensores podem transmitir a origem exata do fogo para os usuários finais, antes que o fogo se espalhe incontrolavelmente. Milhões de nós sensores podem ser depositados e integrados usando frequências de rádio e sistemas ópticos. Além disso, eles podem ser equipados com métodos de captura de energia eficazes, como células solares, porque os sensores podem ser deixados sozinhos por meses ou até anos. Os nós sensores vão colaborar entre si para fazer um sensoriamento distribuído e ultrapassar obstáculos, como árvores e rochas, que bloqueiam a visada de sensores com fio.

3.2.2 **Mapeamento da biocomplexidade do ambiente**

O mapeamento da biocomplexidade do ambiente requer abordagens sofisticadas para integrar informação através de escalas temporais e espaciais. Os avanços da tecnologia no sensoriamento remoto e coleção de dados automatizada têm permitido maior resolução espacial, espectral e temporal, a um custo geometricamente decrescente por unidade de área. Junto com esses avanços, os nós sensores também têm a capacidade de conectar-se à Internet, que permite aos usuários remotos controlar, monitorizar e observar a biocomplexidade do ambiente.

Embora os sensores satelitais e aéreos sejam úteis para observar grande biodiversidade, como por exemplo, a complexidade espacial das espécies de plantas dominantes, eles não são de granularidade fina o suficiente para observar a biodiversidade de tamanho pequeno, que compõe a maior parte da biodiversidade em um ecossistema. Como resultado, existe uma necessidade para implantação ao nível do solo, de nós de sensores sem fio, para observar a biocomplexidade. Um exemplo de mapeamento da biocomplexidade do ambiente é feito na *James Reserve,* no sul da Califórnia. Três redes de monitoramento cada uma tendo de 25 a 100 nós sensores, são implementadas para coletores fixos de visão multimídia e dados de sensores ambientais.

3.2.3 **Detecção de Inundação**

Um exemplo de detecção de inundação é o sistema *ALERT*, implantado nos Estados Unidos. Alguns dos tipos de sensores implantados no sistema *ALERT* são: sensores pluviométricos, de nível da água e meteorológicos. Estes sensores provêem informação ao sistema de banco de dados centralizado, numa forma predefinida. Projetos de pesquisa, como o projeto de banco de dados de dispositivos

COUGAR da *Cornell University* e o projeto de espaço de dados da *Rutgers*, estão investigando abordagens distribuídas, interagindo com nós sensores no campo do sensor, para proporcionar consultas de duração instantânea e de longa duração.

3.2.4 **Agricultura de Precisão**

Alguns dos benefícios são: a habilidade de monitorar os níveis de pesticidas na água potável, o nível de erosão do solo, e o nível de poluição do ar em tempo real.

3.3 APLICAÇÕES DE SAÚDE

Algumas das aplicações de saúde para redes de sensores fornecem interfaces para os deficientes; monitoramento integrado do paciente; diagnóstico; administração da droga em hospitais; monitoramento dos movimentos e processos internos de insetos ou outros pequenos animais; tele monitoramento de dados fisiológicos humanos e; rastreamento e monitoramento de médicos e pacientes dentro de um hospital.

3.3.1 **Tele monitoramento de dados fisiológicos humanos**

Os dados fisiológicos coletados pelas redes de sensores podem ser armazenados por um longo período de tempo, e podem ser usados para exploração médica. As redes de sensores instaladas também podem monitorar e detectar o comportamento das pessoas idosas, como por exemplo, uma queda. Estes pequenos nós sensores permitem ao indivíduo uma maior liberdade de movimentos e permitem que os doutores identifiquem sintomas predefinidos mais cedo. Além disso, eles facilitam uma maior qualidade de vida para os indivíduos comparados aos centros de tratamento. Uma "casa de saúde inteligente" é projetada na

Faculdade de Medicina de Grenoble, na França, para validar a viabilidade deste sistema.

3.3.2 **Rastreamento e monitoramento de doutores e pacientes dentro de um hospital**

Cada paciente tem nós sensores pequenos e leves conectados à eles. Cada nó sensor tem uma tarefa específica. Por exemplo, um nó sensor pode detectar a frequência cardíaca, enquanto outro está detectando a pressão sanguínea. Doutores podem também carregar um nó sensor, que permite à outros doutores a localizá-los dentro do hospital.

3.3.3 **Administração de drogas nos hospitais**

Se os nós sensores podem ser conectados aos medicamentos, a chance de ter e prescrever uma medicação errada aos pacientes pode ser minimizada, porque os pacientes vão ter nós sensores que identificarão suas alergias e medicações necessárias. Sistemas computadorizados têm mostrado que podem ajudar a minimizar eventos adversos da droga.

3.4 APLICAÇÕES DOMÉSTICAS

Enquanto a tecnologia avança, nós sensores e atuadores inteligentes podem ser instalados em eletrodomésticos, como aspiradores de pó, fornos micro-ondas e geladeiras. Estes nós sensores dentro de dispositivos domésticos podem interagir entre si, e com a rede externa via Internet ou satélite. Eles permitem aos usuários finais gerenciar dispositivos domésticos localmente e remotamente, de forma mais fácil.

3.4.1 **Ambientes inteligentes**

O projeto de um ambiente inteligente pode ter duas diferentes perspectivas, por exemplo, uma centrada em humanos e outra centrada na tecnologia. Para a centrada em humanos, um ambiente inteligente tem que se adaptar às necessidades dos usuários finais, nos termos de capacidade de entrada e saída. Para a centrada na tecnologia, novas tecnologias de *hardware*, soluções de rede, e serviços de *middleware* tem que ser desenvolvidos. Os nós sensores podem ser embutidos dentro do mobiliário e de eletrodomésticos, e eles podem comunicar-se entre si e o servidor do quarto. O servidor do quarto pode também se comunicar com outros servidores dos quartos para aprender sobre os serviços que eles oferecem, como por exemplo, impressão, escaneamento e fax. Estes servidores dos quartos e nós sensores podem ser integrados com dispositivos embutidos existentes, para se tornar sistemas auto organizáveis, autorreguláveis e adaptativos, baseados nos modelos de teoria de controle. Outro exemplo de ambiente inteligente é o "Laboratório residencial" no *Georgia Institute of Technology*. A computação e sensoriamento neste ambiente têm que ser confiável, persistente e transparente.

3.5 OUTRAS APLICAÇÕES COMERCIAIS

Algumas das aplicações comerciais são: monitoramento de fadiga de material; construção de teclados virtuais; gerenciamento de estoque; monitoramento da qualidade do produto; construção de espaços de escritórios inteligentes; controle ambiental em construções comerciais; controle e orientação do robô em ambientes de produção automática; brinquedos interativos; museus interativos; controle e automação de processos industriais; monitoramento de área de desastres; estruturas inteligentes com nós sensores embutidos; diagnósticos de máquinas;

transporte; instrumentação industrial; controle local de atuadores; detecção e monitoramento de furto de carros; rastreamento e detecção de veículos; e instrumentação de câmaras de processamento de semicondutores, máquinas rotativas, túneis de vento, e câmaras anecóicas.

3.5.1 **Controle ambiental em construções comerciais**

O ar-condicionado e o calor da maioria dos edifícios são controlados centralmente. Entretanto, a temperatura dentro de uma sala pode variar em alguns graus: Um lado pode ser mais quente de que o outro, porque existe apenas um controle na sala, e o fluxo de ar do sistema central não é bem distribuído. Um sistema de redes de sensores sem fio distribuídos pode ser instalado para controlar o fluxo de ar e a temperatura em diferentes partes da sala. É estimado que cada tecnologia distribuída pode reduzir o consumo de energia em dois quatrilhões de BTUs (*British Termal Units*) nos EUA, enquanto economiza um montante de $55 bilhões de dólares por ano, e reduz 35 milhões de toneladas em emissões de carbono.

3.5.2 **Museus interativos**

No futuro, as crianças vão poder interagir com objetos nos museus para aprender mais sobre eles. Estes objetos irão responder a seu toque e voz. Além disso, crianças podem participar de experimentos de causa e efeito em tempo real, que pode ensiná-los sobre ciência e meio ambiente. Mais ainda, as redes de sensores sem fio podem prover paginação e localização dentro do museu. Um exemplo destes museus, é o *San Francisco Exporatorium*, que mostra uma combinação de medição de dados e experimentos de causa e efeito.

3.5.3 **Detecção e monitoramento de furto de carros**

Nós sensores estão sendo usados para detectar e identificar ameaças numa região geográfica, e reportar estas ameaças aos usuários finais remotos pela Internet, para análise.

3.5.4 **Gerenciando o controle de estoque**

Cada item num estoque pode ter um nó sensor instalado. Os usuários finais podem descobrir a localização exata do item e contar o número de itens na mesma categoria. Se os usuários finais querem inserir novos inventários, o que todos os usuários precisam fazer é unir os nós sensores apropriados aos estoques. Os usuários finais podem rastrear e localizar onde os estoques estão em todos os momentos.

3.5.5 **Rastreamento e detecção de veículos**

Existem duas abordagens para rastrear e detectar o veículo: em primeiro lugar, a linha de apoio do veículo é determinada localmente dentro dos clusters e, em seguida, ela é encaminhada para a estação de base; em segundo lugar, os dados brutos coletados pelos nós sensores são encaminhados para a estação de base para determinar a localização do veículo.

4 **PLATAFORMAS PARA REDES DE SENSORES SEM FIO**

Neste capítulo, serão apresentadas algumas plataformas que vêm sendo propostas ou adotadas na construção de RSSFs, tanto em projetos acadêmicos, quanto por fabricantes de dispositivos para esse fim.

Devido às restrições existentes nas RSSFs, elas são compostas somente pelos protocolos da camada de enlace (MAC) e rede (roteamento), que serão apresentados nos exemplos de plataforma descritos nas próximas seções. É importante salientar que a tecnologia para projetar e construir RSSFs, está comercialmente disponível e tende a se tornar cada vez mais acessível com a produção em larga escala de diferentes tipos de microssensores.

A figura 1 apresenta alguns exemplos de nós sensores sem fio resultantes de pesquisas em diversas instituições, como o *COTS Dust* e o *Smart Dust*, da Universidade da California, Berkeley; *WINS (Wireless Integrated Network Sensors*) da Universidade da California, Los Angeles e *JPL Sensor Webs* do *Jet Propulsion Lab* da NASA.

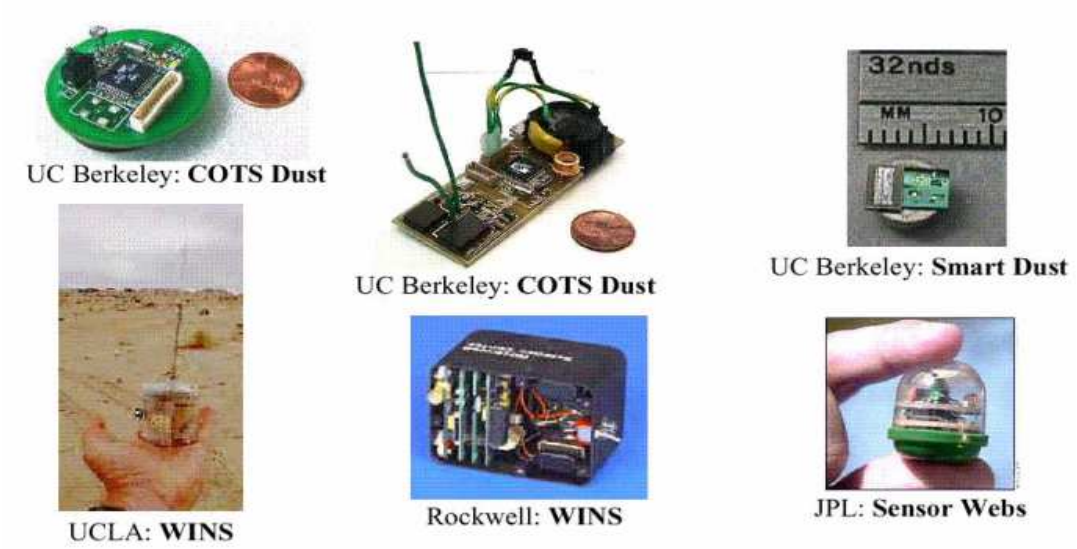

Figura 1 - Projetos acadêmicos de nós sensores

4.1 PRINCIPAIS PLATAFORMAS

As diversas plataformas citadas abaixo, atendem necessidades específicas de sensoriamento, dadas por um determinado ambiente à ser monitorado. Algumas apresentam um hardware mais robusto, com quantidade razoável de memória, dotadas de capacidade de multitarefa e/ou *multithreading*, que podem ser utilizadas num número maior de situações. Outras, têm seu foco em economia de energia, ou maior número de sensores, tendo um hardware mais pobre, mas ainda sim tendo capacidade aceitável de computação, pois o sistema operacional compatível com essas plataformas, requer menor quantidade de memória, executa menos tarefas simultânea ou serializadamente (FIFO), e têm seu foco em maior longevidade de bateria, mirando ambientes ou eventos específicos de sensoriamento.

As plataformas a seguir, possuem um maior detalhamento por terem uma maior relevância atualmente.

4.1.1 **MSB430**

A placa de sensor modular (MSB) é a nova versão do módulo universal projetado especialmente para pesquisa e educação pela *ScatterWeb*, uma arquitetura da Universidade Livre de Berlim. Seu predecessor, a placa de sensor embutida (ESB), está em uso em muitas universidades e instituições de pesquisa ao longo da Europa. Ela tem as seguintes características principais:

- MPU: A MSB430 usa um processador MSP430F1612IPM com 55 kB de Flash e 5 kB de RAM;
- Radio: Um transceptor de rádio CC1020 da *Texas Instruments* é o dispositivo de comunicação padrão do MSB430. Ele opera na banda ISM

de 868 MHz e alcança a taxa máxima de dados de 153,6 kbps (a taxa de dados típica é 19 kbps). Além disso, vários conectores externos à MPU existem e podem ser usados para incluir transceptores adicionais. Assim, acessa GSM/GPRS, Bluetooth, 802.11 e outras redes comerciais ou locais possíveis;

- Sensores/Atuadores: O MSB430 tem os seguintes sensores/atuadores embutidos: sensor de umidade (*Sensirion SHT11*), sensor de temperatura (*Sensirion SHT11*), um acelerômetro de três eixos MMA7260Q e um Led vermelho. Sensores/atuadores adicionais podem ser conectados via conectores externos disponíveis a MPU;

- Interfaces externas: As interfaces externas incluem um soquete para SD-card, interface JTAG, e alguns conectores externos a MPU;

- Interface de programação sem fio: (Re)programação sem fio remota dos nós é possível, usando a USB embutida e os dispositivos web embarcados que são conectados aos PCs. Estas interfaces também permitem depuração e coleta de dados do sensor. Transceptores adicionais podem também ser conectados ao *mote*, possibilitando reconfiguração sem fio remota “fora-da-banda” dos nós;

- Acesso à redes legadas: O acesso ao outras redes sem fio é possível usando o USB embutido e os dispositivos web incorporados, para conectar os nós ao PC que está conectado a uma rede legada;

- Aplicações: Aplicações incluem monitoramento do ambiente, entretenimento móvel avançado, construções inteligentes, recuperação de desastres e redes *ad hoc*;

- Interface de desenvolvimento de protocolos: Acesso aberto para configurar o MAC e o algoritmo de roteamento usando a interface *JTAG* para programar a MPU, que é diretamente responsável por escrever os bytes de dados no transceptor de rádio via *UARTs* das MPUs. O degrau de flexibilidade é sujeito a limitações de *hardware*. O transceptor provê acesso para o canal de sensoriamento e dados da RSSI, que pode ser usado em várias implementações de protocolo;

- Software e sistema operacional: Uma interface C está disponível para programação dos nós com o sistema operacional *ScatterWeb*, mas o *NesC* com o *TinyOS* também pode ser usado;

- Fontes de energia: O MSB430 pode ser alimentado tanto por um grupo de baterias, contendo três baterias AAA (1.5V), e através de uma fonte de alimentação externa usando, por exemplo, um gerador de tensão externo, um painel solar ou um capacitor de alta potência;

- Modos de economia de energia: Operação de baixa potência é em parte devida ao micro controlador TI MSP430 de baixíssimo consumo de energia, que apresenta consumo de corrente extremamente baixo em modo ativo e em modo de economia de energia. A fim de minimizar o consumo de energia, ele está em modo de repouso na maior parte do tempo, acordando rapidamente a cada nova tarefa, e rapidamente retorna

ao modo de repouso após completar a tarefa. O rádio CC1020 suporta transmissão de energia programável que possibilita aplicações de baixo consumo;

- Memória e armazenamento: Em adição aos 55 kB de memória flash, e 5 kB de ram embutidos na MPU, e um soquete de cartão SD está também disponível, que permite capacidade de armazenamento de dados adicional escalável;

- Custo: O custo é aproximadamente 100 euros por nó, excluindo acessórios;

- Tamanho: O tamanho é de 36 por 41 mm;

- Suporte e Disponibilidade: Documentação online disponível no website do *ScatterWeb*, bem como no website da Universidade Livre de Berlim. O *Hardware* MSB está disponível no *ScatterWeb*, enquanto que o *software* é obtido no site da Universidade Livre de Berlim. A placa de sensor modular MSB é provida para pesquisa e demonstrações na área de redes de sensores sem fio. Ela não foi certificada, nem testada para uso em qualquer ambiente crítico fora dos laboratórios de pesquisa. O Suporte a usuários da *ScatterWeb* está focado principalmente em seus produtos comerciais, embora o suporte limitado está disponível para o MSB, bem como na Universidade Livre de Berlim;

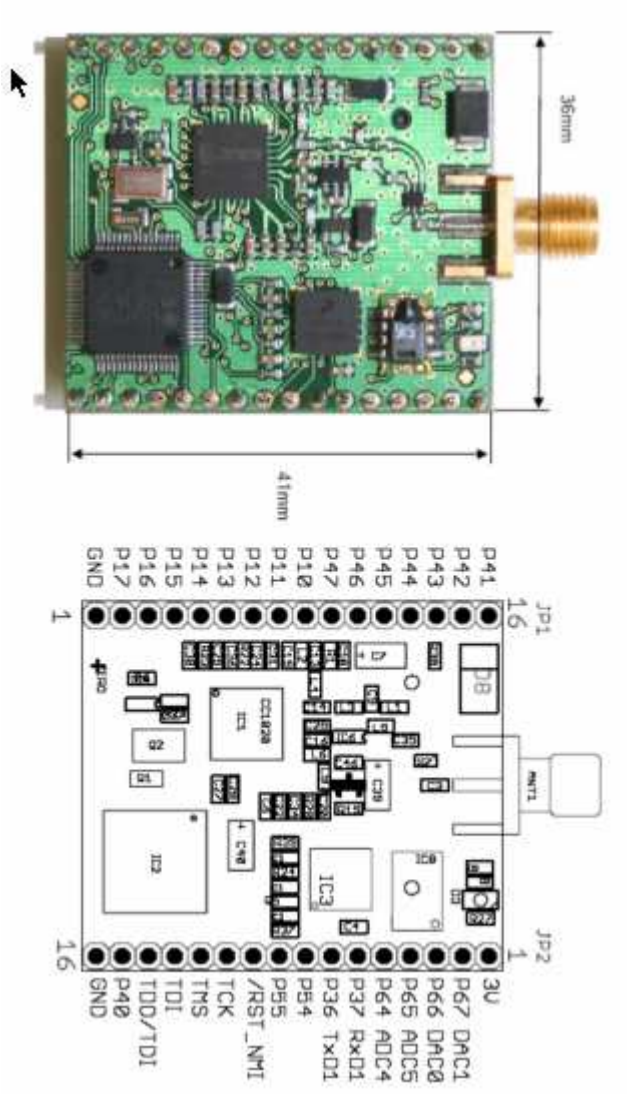

Figura 2 - O módulo MSB430

4.1.2 **TMote SKY**

O *Tmote Sky* é uma plataforma para redes de sensores sem fio de propósito geral, com um grande mercado tanto na academia quanto na indústria. Ele é o sucessor das plataformas populares *TelosA* e *TelosB* da Universidade de Berkeley.

O *Tmote Sky* é uma das poucas plataformas de redes de sensores sem fio certificadas pela FCC disponíveis no mercado. O *Tmote Sky* foi desenvolvido e suportado pelo *moteiv*, que é agora propriedade da *Sentilla*.

- MPU: O *Tmote Sky* usa um processador MSP430F1611 com 48 kB de memória flash e 10 kB de RAM;

- Radio: O *Tmote Sky* usa um transceptor de rádio CC2420 da *Texas Instruments*, complacente com o IEEE 802.15.4, operando na banda ISM de 2,4GHz, na taxa de dados de 250 kbps, 40 kbps e 20 kbps. O transceptor emprega um MAC CSMA-CA padrão;

- Sensores: Em adição ao sensor de temperatura interna do micro controlador MSP430, a placa do *Tmote Sky* tem posições predefinidas para montar um sensor de umidade/temperatura do *Sensirion* AG (modelos SHT11 e SHT15 são suportados), bem como para os sensores de luz como o *Hamamatsu Corporation* S1087 para sensoriamento;

- Interfaces Externas: As interfaces externas incluem USB e JTAG para (re)programar a flash da CPU, bem como os I2C, ADC, DAC e SPI. O *Tmote Sky* tem dois conectores de expansão e um par de jumpers embutidos, onde podem ser configurados como seus dispositivos adicionais (sensores analógicos, displays de LCD e periféricos digitais), e que podem ser controlados pelo módulo *Tmote Sky*;

- Interface de programação sem fio: (Re)programação sem fio remota dos nós não é uma característica padrão do *Tmote Sky*, e esta pode ser alcançada apenas conectando um transceptor compatível ao conector JTAG, que opera em diferentes canais do que este dispositivo padrão de comunicação por nós;

- Acesso à redes Legadas: Por cada nó possuir um rádio complacente com 802.15.4 (ZigBee), ele comunica-se com dispositivos gateway de redes legadas que são complacentes com o ZigBee. Alternativamente, um transceptor adicional pode ser adicionado para permitir que cada nó acesse uma rede legada;

- Interface de desenvolvimento de protocolos: O *Tmote Sky* é equipado com um transceptor compatível com o protocolo 802.15.4. Assim, ele oferece flexibilidade limitada para desenvolvimento de protocolos proprietários;

- Software e Sistema operacional: É compatível com *TinyOS;*

- Fontes de Energia: Um grupo de baterias contendo duas baterias AA (1,5V). O *Tmote Sky* pode também ser energizado via interface embutida USB quando plugado dentro de uma porta USB do computador host para programação ou comunicação;

- Modos de economia de energia: Operação de baixa energia é devida em parte ao micro controlador de baixíssimo consumo de energia TI MSP430, que apresenta consumo de corrente em modo ativo e inativo extremamente baixo. Em vista de minimizar o consumo de energia, ele está em modo de repouso boa parte do tempo, acordando rapidamente para processar cada nova tarefa, e retorna rapidamente ao modo de repouso após completar a tarefa. O rádio suporta transmissão de energia programável que habilita aplicações de baixo consumo, e permite um modo de desligamento para ciclos de baixa carga;

- Memória e armazenamento: Em adição aos 48 kB de memória flash e aos 10 kB de RAM embutidos na MPU, a memória flash serial de 1MB, M25P80, pode ser lida e escrita pelo MSP430;

- Custo: O custo é aproximadamente $ 130 dólares por nó, excluindo-se acessórios;

- Tamanho: O tamanho é de 3,2cm por 6,55cm por 0,66cm;

- Suporte e disponibilidade: Existe documentação online, mas não há suporte ao usuário da *Sentilla*;

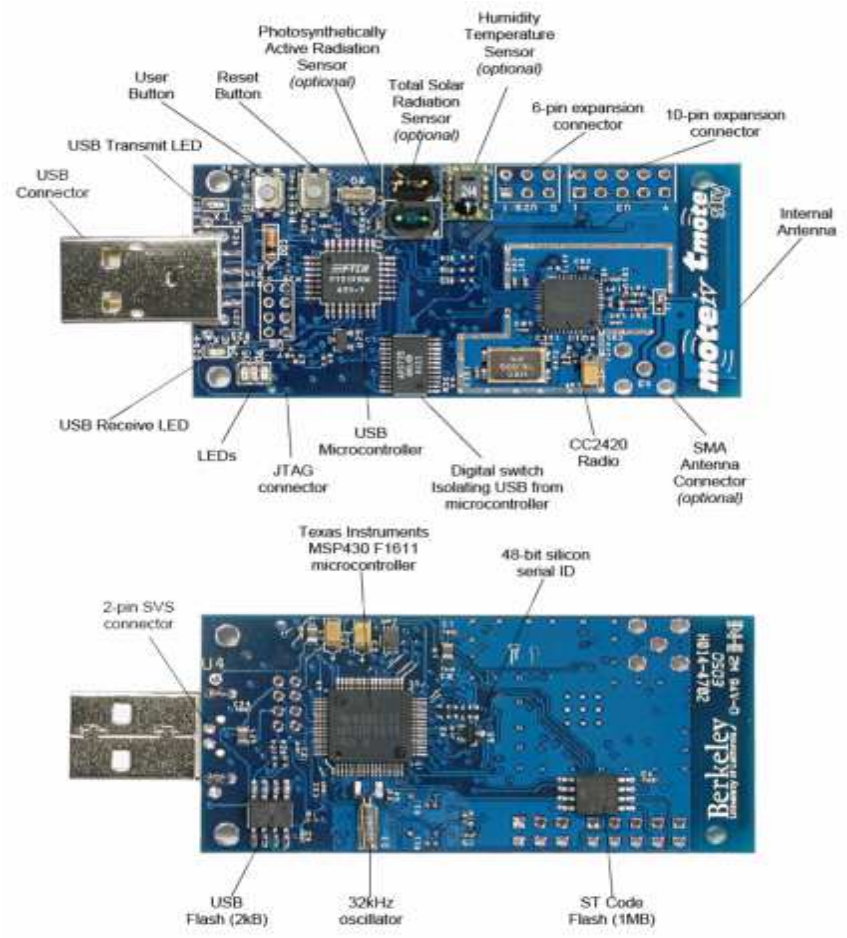

Figura 3 - Frente e Verso do módulo *Tmote Sky*

4.1.3 **BTnode**

O *BTnode* é uma plataforma de computação versátil e de comunicação sem fio autônoma baseada no rádio Bluetooth, um rádio de baixo consumo secundário e um micro controlador. Ele serve como demonstração e plataforma de prototipagem para pesquisas em dispositivos móveis, e redes *ad hoc* conectadas (MANETs) e redes de sensores distribuídos (RSSFs). O BTnode tem sido desenvolvido juntamente na *ETH Zurich* (Instituto Federal Suíço de Tecnologia em Zurique), pelo Grupo de pesquisa para sistemas distribuídos, e pelo Laboratório de Engenharia de computação e redes. Seu desenvolvimento foi suportado primeiramente pelos projetos do NCCR-MICS e do *Smart-Its*, e ultimamente sendo parte de uma iniciativa européia "*The Disappearing Computer*", e fundado tanto pela comissão da união européia e do escritório federal suíço para educação e ciência.

Houve três grandes revisões de hardware na plataforma de hardware do *BTnode*: *BTnode* rev 1, *BTnode* rev 2 e *BTnode* rev 3. A *BTnode* rev 3 é usada com sucesso agora em vários projetos de pesquisa abrangendo desde simples aplicações com poucos nós até aplicações interativas de rede. Como um sistema de software, o *BTnode* pode executar tanto os sistemas operacionais *BTnut* quanto *TinyOS*.

- MPU: O processador *Atmel* ATmega128L é usado, com 64+180 kB de SRAM, 128 kB de memória flash e 4 kB de EEPROM;

- Rádio: O *BTnode* usa um único chip de rádio *Zeevo* ZV4002 (Certificado pelo Bluetooth v1.2), que emprega *Scatternets* com saltos de frequência adaptativos/lentos (AFH/SFH) com 4 *Piconets* simultâneas, tendo um máximo de 7 escravos. Ele opera na frequência de 2.4 a 2.483GHz a

taxa máxima de 1 Mbits/s. O *BTnode* tem um segundo rádio, chamado *Texas Instruments* CC1000 operando geralmente, na frequência de 433 a 915 MHz, e em particular na frequência ISM de 868 MHz. Ele é o mesmo rádio usado nos motes *Mica2*;

- Sensores/Atuadores: Não há sensores embutidos. Capacidades de sensoriamento da plataforma *BTnode* são obtidas conectando sensores independentes à placa de sensores, nas linhas apropriadas da extensão J1 ou conectores de depuração J2 (conectores “cabo-a-placa” *Molex* de 1.25mm e placa-a-placa *Hirose* DF17). A alta flexibilidade da plataforma *BTnode* permite aos usuários adicionar uma variedade de sensores diferentes, desta forma, permitindo fácil customização da plataforma e prototipagem de uma variedade de aplicações ubíquas baseadas em sensor. Quando se adicionam novos sensores ao *BTnode*, entretanto, os *drivers* correspondentes do sensor customizado tem que ser implementados pelo usuário. Placas de sensor disponíveis são placas de sensor *ssmall* do *Particle Computer GmbH* e do *BTsense*.

  Existem duas versões da placa de sensor *ssmall*. Uma versão, a placa de sensor *ssmall medium*, é equipada com os seguintes sensores: TSL2500 sensor de luz do dia e luz de infravermelho, produzida pela TAOS (*Texas Advanced Optoeletronic Solutions*), sensor de temperatura TC74 (acurácia típica: ± 0,5ºC), 2 LEDs (podem ser substituídos por outros atuadores), microfone capacitivo MAX8261 OP (alta precisão, alta linearidade), sensor de aceleração de 2 eixos ADXL210 produzido pela *Analog Devices* (máximo de 10g, resolução de ± 40 mg, capacidade de

resposta < 1ms). Ele é conectado ao *BTnode* através da placa de programação USP. A outra versão, a placa de sensor *ssmall full*, é idêntica à placa *médium*, mas apresenta um sensor de aceleração de 3 eixos (composição de 2 ADXL210).

A placa *BTsense* pode ser facilmente afixada ao lado do *BTnode*. Ela apresenta o sensor de temperatura TC74 (digital, I2C), sensor de luminosidade TSL252R (analógico), sensor passivo de movimento infravermelho AMN1 e o alto-falante piezelétrico 7BB-12-9. Sensores digitais I2C adicionais, como um sensor analógico externo, podem ser adicionados à placa.

Atuadores podem também ser facilmente conectados à plataforma *BTnode* através do conector de extensão J1. Para fins de depuração, 4 LEDs são embutidos na placa de circuito do *BTnode*;

- Interfaces externas: As interfaces externas incluem UART, SPI, I2C, GPIO, ADC e conectores padrão “cabo-a-placa” *Molex* 1.25mm e “placa-a-placa” *Hirose* DF17;

- Interface de programação sem fio: (Re)programação dos nós é possível, pela existência de dois módulos de rádio independentes. O rádio Bluetooth, por exemplo, pode ser usado como um canal de retorno discreto de programação/depuração e monitoramento, enquanto que o rádio *Chipcom* opera como um canal de comunicação padrão entre os nós;

- Acesso à redes legadas: O *BTnode* é capaz de interagir facilmente (através do rádio Bluetooth) com outros dispositivos como telefones celulares ou PDAs;

- Interface de desenvolvimento de protocolos: livre acesso para configurar o MAC e algoritmos de roteamento usando a interface JTAG para programar a MPU, que é diretamente responsável por escrever os bytes de dados para os transceptores de rádio via as *UARTs* das MPUs. O degrau de flexibilidade é sujeito a restrições de hardware;

- Software e sistema operacional: Usa o sistema operacional *BTnut* (programação C multitarefa) e *TinyOS* (programação *NesC* com contribuição do *TinyBT*);

- Fontes de energia: Usa fontes chaveadas separadas para Bluetooth, rádio de baixo consumo, periféricos e *core* da MPU. Ele é alimentado por uma fonte DC externa de 3.8 a 5V ou duas células AA com chave de liga/desliga. Ela é descrita como tendo alto consumo de energia, e provê um monitor de bateria. A fonte de energia chaveada no *BTnode* oferece acesso direto à corrente para perfis *in-site* de consumo de energia, tanto nos sistemas de rádio, quanto no *core* do micro controlador, sob condições de operação em tempo real. Isto pode ser usado para análise de desempenho detalhada e a avaliação da eficiência de energia de vários protocolos;

- Modos de economia de energia: Os modos de repouso permitem que a MPU desligue módulos não utilizados, economizando energia. Existem

vários modos de repouso programáveis permitindo que o usuário adapte o consumo de energia aos requerimentos das aplicações. O *hardware* do Bluetooth é, entretanto, ótimo em termos de consumo de energia;

- Memória e armazenamento: A memória interna do *BTnode* consiste de 128 kbytes de uma memória flash programável no sistema, 4 kbytes de memória EEPROM e 4 kbytes de SRAM. A SRAM embutida é estendida a 64 kbytes através de um módulo externo de memória de 256 kbytes. Os 180 kbytes dos restantes 196 da SRAM externa são providos como três bancos de memória cache, de 60 bytes cada. Os 16 kbytes restantes são inutilizados;

- Custo: O custo é aproximadamente 100 dólares por nó, excluindo acessórios;

- Tamanho: O tamanho é 58,15 x 33 mm, conectado a um suporte de bateria de 2 células AA;

- Suporte e disponibilidade: O *BTnode* é uma plataforma de prototipagem bem estabelecida e é usada em mais de 30 projetos de pesquisa. O *website* do *BTnode* é uma fonte rica de informação útil e documentação técnica. A lista de e-mail ativa, e uma comunidade de código aberto adicionalmente, oferecem a possibilidade de colocar questões e problemas, e ter assistência dos usuários de *BTnode* mais experientes. Produtos estão disponíveis na *Art of Technology* (fabricante contratada), Zurique, Suíça, e e-mail: btnode@art-of-technology.ch.

O *BTnode* é uma plataforma de hardware muito robusta, que tem sido largamente utilizada tanto em pesquisa ou ensino. As ferramentas de desenvolvimento de software podem ser facilmente instaladas tanto em Windows, Linux ou Mac. Sistemas operacionais e guias detalhados de instalação e tutoriais estão disponíveis no *website* do *BTnode*. O *BTnode* pode ser programado em linguagem C padrão, quando o *software BTnut* está instalado, ou em *NesC*, quando o sistema operacional *TinyOS* é utilizado.

Figura 4 - *BTnode* revisão 3

4.1.4 **EyesIFXv2**

O *EyesIFXv2* é um nó sensor desenvolvido pela *Infineon* para as "Redes de sensores sem fio eficientes em energia, auto-organizáveis e colaborativas", projeto *EYES* (IST-2001-34734). A *Infineon* tem combinado o *hardware* base do *EYES* com um número de conjuntos de periféricos otimizados para criar uma série de chips focados em aplicações automotivas específicas, industriais e para consumidores. A *Infineon* recentemente liberou uma nova versão do nó *Eyes*, o *EyesIFXv2.1*. Os

componentes dessa nova placa são ainda os mesmos da antiga v2.0. Dois leds foram adicionados para mostrar a atividade do rádio. As duas versões do nó, têm a mesma unidade de rádio, então elas podem comunicar-se sem problemas.

- MPU: O processador MSP430F1611 é usado, com 10 kb de RAM embutidos e 48 kb de memória flash/ROM. Adicionalmente, há uma EPROM serial *Atmel* de 4Mb conectada via um barramento SPI;

- Rádio: O rádio é um transceptor de único chip TDA5250 de baixo consumo na faixa de 868 a 870 MHz, operando com modulação FSK e acima de 64 kbps (mas tipicamente é mais baixo, 19 kbps) com conectividade sem fio *half-duplex*;

- Sensores/Atuadores: O equipamento sensor da plataforma *EYES* engloba um sensor de temperatura e um sensor de luminosidade. O sensor de temperatura é do modelo LM61 produzido pela *National Semiconductor Range*. O sensor de luminosidade é do modelo NSL19-M51, resistor dependente de luz. Além dos sensores embutidos, sensores externos adicionais podem ser conectados usando um extensor de porta provido pela plataforma. Para mostrar a informação de *status*, uma matriz de 4 LEDs está disponível (na versão 2.1 existem 6 LEDs disponíveis). Adicionalmente, há uma porta de expansão que permite a conexão de placas secundárias com sensores analógico-digitais adicionais e atuadores;

- Interfaces Externas: Interfaceamento de dados externos é possível através de uma interface USB ou JTAG, que permite programação do

micro controlador e depuração no circuito. Adicionalmente, há uma porta de expansão que permite a conexão de placas de sensores/atuadores secundárias de instrumentos de medição (por exemplo, analisador lógico). Este conector provê acesso ao barramento de 3 fios SPI. Adicionalmente, o nível analógico RSSI do TDA5250 pode ser medido e uma referência externa de voltagem para o conversor analógico-digital pode ser provido. Finalmente, alguns bits das portas do MSP430 podem ser acessadas para operações de I/O;

- Interface de programação sem fio: (Re)programação sem fio remota dos nós é possível através da interface USB que pode ser conectada a um cartão USB sem fio para Bluetooth ou outro padrão;

- Acesso à redes legadas: Comunicação sobre redes sem fio diferentes pode ser habilitada usando dispositivos USB sem fio conectados à interface USB;

- Aplicações: A plataforma *EYES* é visada para aplicações de consumo mínimo de energia e baixa transmissão de dados, demandando ambientes de RF onde robustez a ruídos e interferências são essenciais;

- Interface de desenvolvimento de protocolos: Os nós *EYES* são totalmente programáveis (através da interface USB ou JTAG), então estes usuários podem projetar e desenvolver blocos funcionais customizados, como o MAC, roteamento e outros. Porém, uma pilha de protocolos muito essencial é provida pela *Infineon* e pelo *TinyOS* para iniciantes. A pilha inclui uma implementação do CSMA para MAC;

- Sistema operacional e software: Utiliza *TinyOS* com *NesC;*

- Fontes de energia: Os nós funcionam em baterias de lítio com a capacidade de 1000mAh. Alternativamente, há um conector externo de energia para alimentar com corrente DC bem como uma porta USB que pode alimentar o nó;

- Modos de economia de energia: *EYES* usa o MSP430 de baixo consumo, bem como o TDA5250 de baixo consumo. Em comparação a antigos transceptores ISM usados para aplicações de *TinyOS*, o TDA5250 tem menor consumo de energia. No modo de transmissão, ele precisa de apenas 12mA; enquanto que recebendo, 9mA são suficientes. Por essa razão, o transceptor é particularmente adequado para aplicações operadas por bateria que requerem um grande tempo de operação;

- Memória e armazenamento: A capacidade de memória e armazenamento é baseada nos 10 kB de RAM, e 48 kB de flash/ROM embutidos no micro controlador MSP430, e na EPROM serial externa de 4Mb da *Atmel*;

- Custo: O custo é aproximadamente 70 euros por nó, excluindo-se acessórios;

- Tamanho: O Tamanho é 3,2cm por 6,55cm por 0,66cm;

- Suporte e disponibilidade: Documentação pode ser baixada, de graça, do *website* da *Infineon*. Informação adicional pode também ser encontrada no *website* dos grupos acadêmicos de pesquisa, que estão usando a plataforma para experimentação, como a *Twente University* (Holanda),

Universidade Técnica de Berlim (Alemanha), CINI (Itália) e a Universidade de Padova (Itália);

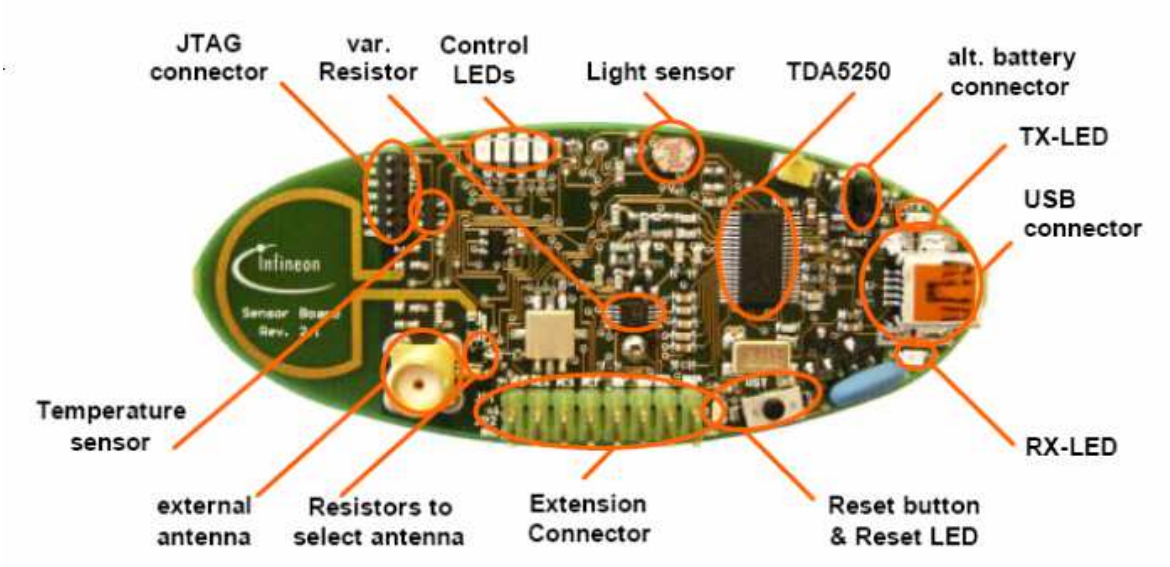

Figura 5 - Lado superior do nó *EyesIFXv2*

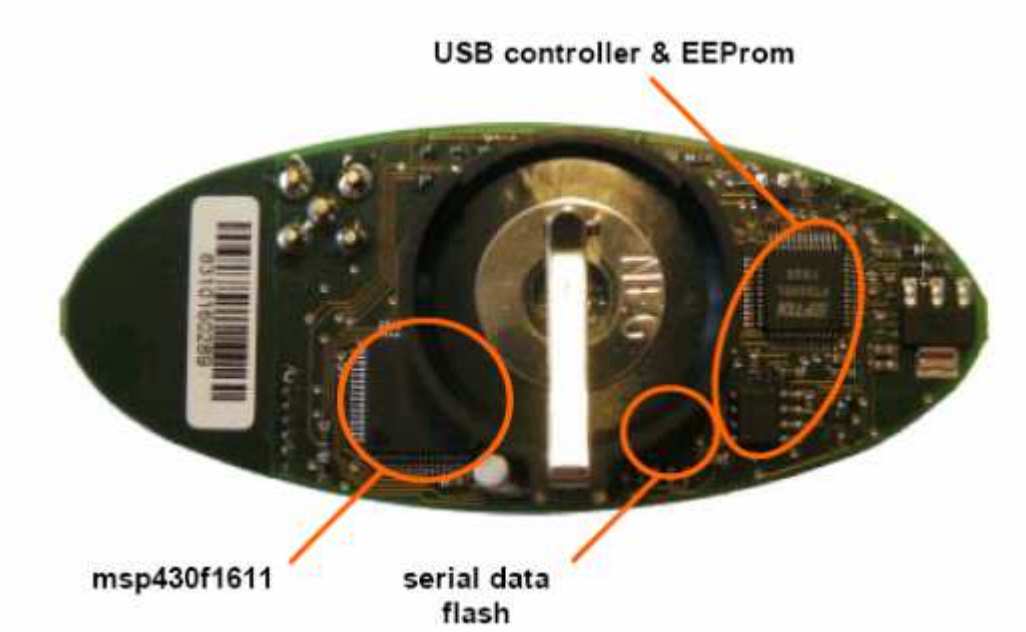

Figura 6 - Lado Inferior do nó *EyesIFXv2*

4.1.5 **Mote MicaZ**

A *Crossbow* traz três famílias MICA de módulos *mote* processador/rádio: *MICAz* (MPR2400), *MICA2* (MPR400) e *MICA2DOT* (MPR500). O rádio *MICAz* trabalha na banda ISM global de 2,4GHz e suporta IEEE 802.15.4 e Zigbee.

- MPU: O processador *Atmel* ATmega128L é usado, com 128 kb de memória flash, 4 kb de SRAM e memória flash serial externa de 512 kb;

- Rádio: O Rádio é um *Texas Instruments* CC2420, que é complacente com IEEE 802.15.4 (ZigBee), operando de 2,4 a 2,48 GHz, com 250 kbps de taxa de dados;

- Sensores/Atuadores: A *Crossbow* oferece uma variedade de sensores e placas de aquisição de dados para o *mote MICAz*. Todas estas placas conectam-se ao *MICAz* via conector de expansão de 51 pinos. Sensores customizados e placas de aquisição de dados estão também disponíveis. Não há sensores embutidos. Ele também tem 3 LEDs programáveis;

- Interfaces externas: Interfaceamento com dados externos é possível através de 2 *UARTs* (Transmissão e recepção assíncrona universal), barramento de Interface de porta serial (SPI), barramento I2C de *hardware* dedicado, conector de expansão de 51 pinos, interfaces de I/O analógico-digitais e uma porta JTAG;

- Interface de programação sem fio: (Re)programação sem fio remota dos nós é possível através da estação base, com qualquer *mote MICAz* que pode funcionar plugando a placa do processador/rádio na placa de

interface serial *Crossbow* MIB510CA. A MIB510CA provê uma interface serial RS-232 tanto para programar ou comunicação de dados;

- Acesso à redes legadas: Comunicação através de redes sem fio diferentes pode ser habilitada usando a estação base para conectar a um PC, que pode ser habilitado com uma interface para qualquer outra rede;

- Aplicações: Aplicações incluem monitoramento e segurança *indoor* de construções e acústica, vídeo, vibração, e coleta de alta velocidade de dados de sensores. As redes de sensores de larga escala (1000 ou mais pontos) são possíveis;

- Interface de desenvolvimento de protocolos: O *MICAz* está equipado com um transceptor compatível com o protocolo 802.15.4 padrão, e oferece flexibilidade limitada para desenvolvimento de protocolo proprietário;

- Software e sistema operacional: Ele utiliza *TinyOS* com *NesC;*

- Fontes de Energia: Os nós são alimentados por duas baterias AA

- Modos de economia de energia: Combina as características de economia de energia da MPU ATmega128L e o transceptor CC2420;

- Memória e armazenamento: a capacidade de memória/armazenamento inclui 128 kb de memória flash e 4 kb de SRAM embutidas na MPU e memória flash serial externa de 512 kb;

- Custo: O custo é de aproximadamente 100 euros por nó, excluindo acessórios;

- Tamanho: O tamanho é 58 x 32 x 7 mm;
- Suporte e desenvolvimento: Documentação e suporte no *website* da *Crossbow* e contatos.

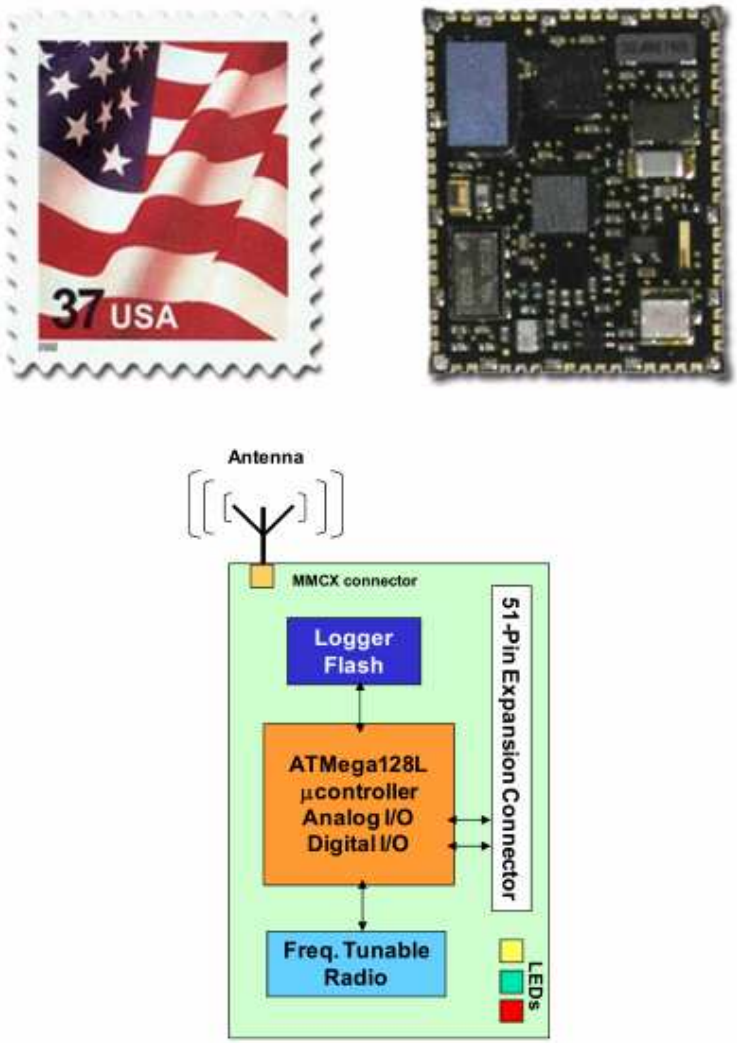

Figura 7 - A Plataforma *MicaZ*

4.1.6 **Mote Mica2**

O *MICA2* está disponível em configurações multicanal de 868/900 MHz e suporta operação ágil de frequência. O *MICA2* da *Crossbow* é similar ao *MICAZ*, exceto por utilizar um rádio diferente: Um *Texas Instruments* CC1000 que opera a

868/900 MHz, com taxa de dados de 38,4 kbps. Assim, ele é usado para aplicações de alta taxa de dados. Há também suporte explícito para programação remota sem fio.

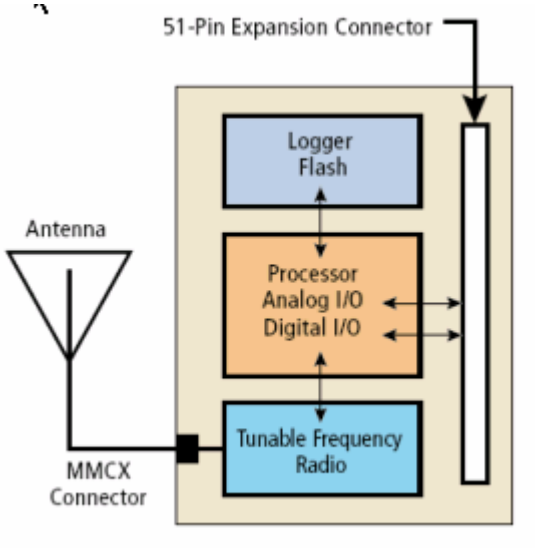

Figura 8 - A Plataforma *Mica2*

4.1.7 **Mote Mica2DOT**

O *MICA2DOT* está disponível em operação multicanal (Compatível com o *MICA2*) de 868/916 MHz, 433 MHz ou 315 MHz com taxa de dados de 38,4 kbps e suporta operação ágil de frequência. O *MICA2DOT* difere do *MICA2* em seu tamanho (25 x 6 mm) e transceptor diferente com operação de canal múltiplo. Seu tamanho pequeno torna-o adequado para etiquetas inteligentes "bidirecionais" ativas, emblemas espertos e computação vestível. Ele possui conectores de expansão de 18 pinos, ao contrário dos 51 pinos do *MICA2* e *MICAZ.*

4.1.8 **IMote2**

O *Imote2* é construído em torno da CPU *XScale* PXA271 de baixo consumo, e também integra um rádio complacente com IEEE 802.15.4. O projeto é modular e

empilhável com conectores de interface para placas de expansão, tanto na parte de cima, quanto na parte de baixo. Os conectores da parte de cima provêm um conjunto padrão de sinais de I/O para placas de expansão básicas. Os conectores da parte de baixo provêm interfaces de alta velocidade adicionais, para I/O de aplicações específicas. Uma placa de bateria para fornecimento de energia do sistema pode ser conectado no outro lado.

- MPU: A CPU *Marvell* PXA271 pode operar em baixa voltagem (0,85V), modo de baixa frequência (13MHz), com isso habilitando operação de consumo muito baixo de energia. O PXA271 é um módulo *multichip* que inclui três chips em um único pacote, uma CPU com 256kb de SRAM, 32MB de SDRAM e 32MB de memória flash. O PXA271 inclui um co-processador MMX para acelerar operações de multimídia. Ele adiciona 30 novas instruções (DSP) de processador, suporte para alinhamento e operações de vídeo;

- Rádio: O *Imote2* usa o rádio transceptor IEEE 802.15.4 CC2420 da *Texas Instruments*. O CC2420 suporta uma taxa de dados de 250kbps com 16 canais na banda de 2,4GHz. Outras opções de rádio vão ser habilitadas através de cartões SDIO e UART/USB;

- Sensores/Atuadores: Nenhum sensor embutido, mas placas de sensores estão disponíveis;

- Interfaces externas: A CPU integra muitas opções de I/O tornando-o extremamente flexível suportando diferentes sensores, conversores A/D, rádios, etc. Estas características de I/O incluem I2C, 2 Portas Seriais

Síncronas (SPI) uma das quais dedicadas ao rádio, 3 *UARTs* de alta velocidade, GPIOs, SDIO, cliente e host USB, interfaces de *codec* de áudio AC97 e I2S, porta rápida de infravermelho, PWM, uma interface de câmera e um barramento de alta velocidade;

- Interface de programação sem fio: (Re)programação sem fio remota dos nós é possível através de cartões SDIO e interfaces UART/USB, que habilitam opções adicionais de rádio a serem incluídas;

- Acesso a redes legadas: A Comunicação sobre redes diferentes, pode ser habilitada usando cartões SDIO e interfaces UART/USB;

- Aplicações: As aplicações incluem processamento digital de imagem, manutenção baseada em condição, monitoramento e análise industrial, e monitoramento sísmico e de vibração;

- Interface de desenvolvimento de protocolos: O uso de um transceptor complacente com IEEE 802.15.4 limita a flexibilidade do desenvolvimento de protocolos quando usando o transceptor primário. Entretanto, a possibilidade de conectar outros dispositivos transceptores via interfaces SDIO e UART/USB, permite o desenvolvimento de protocolos usando estes rádios secundários;

- Software e sistema operacional: Ele usa *TinyOS*, Linux, SOS, e uma série de outros sistemas operacionais e ferramentas de programação associadas;

- Fontes de energia: A placa de baterias do *Crossbow Imote2* conectada tanto a conectores básicos ou avançados*, Li-Ion* recarregável ou baterias

Li-Poly com um carregador embutido, conector USB mini-B (também para carregar uma bateria conectada) e uma bateria primária adequada ou outra fonte de energia podem ser conectadas via um conjunto dedicado de blocos de bateria de soldadas na placa do *Imote2*;

- Modo de economia de energia: A CPU Intel PXA271 do Imote2 pode operar em baixa voltagem (0,85v), modo de baixa frequência (13MHz), assim habilitando operação em baixíssimo consumo de energia. Ele tem um número de modos de baixa energia diferentes como modo de repouso e modo de repouso profundo;

- Memória e armazenamento: A capacidade de memória/armazenamento inclui memórias embutidas na MPU (256kb de SRAM, 32MB de SDRAM e 32MB de memória flash), e memória/armazenamento expansíveis através das interfaces SDIO e USB;

- Custo: O custo é aproximadamente 100 euros por nó, excluindo-se acessórios;

- Tamanho: O tamanho é 36 mm x 48 mm x 9 mm;

- Suporte e disponibilidade: Documentação e suporte no *website* da *Crossbow* e contatos.

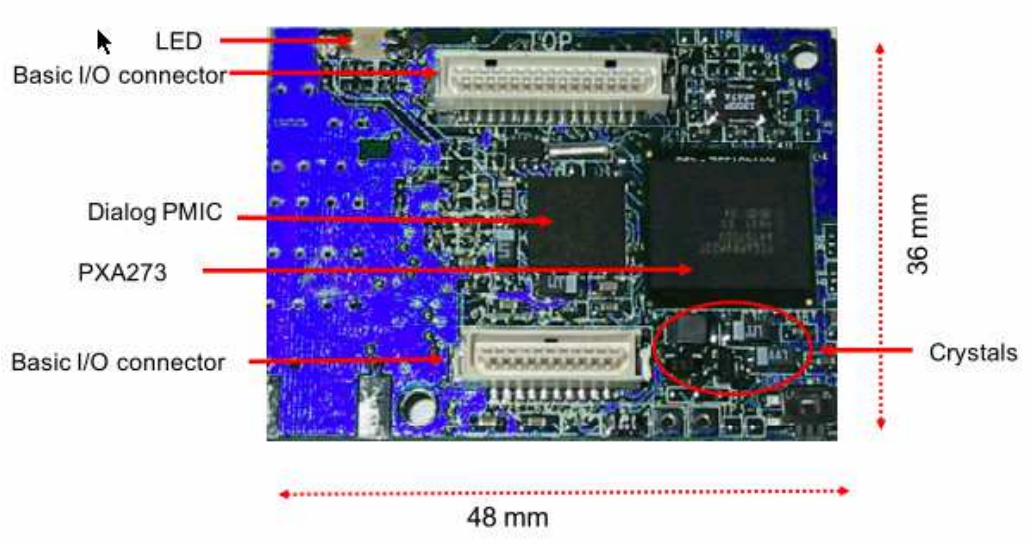

Figura 9 - Vista superior do *Imote2*

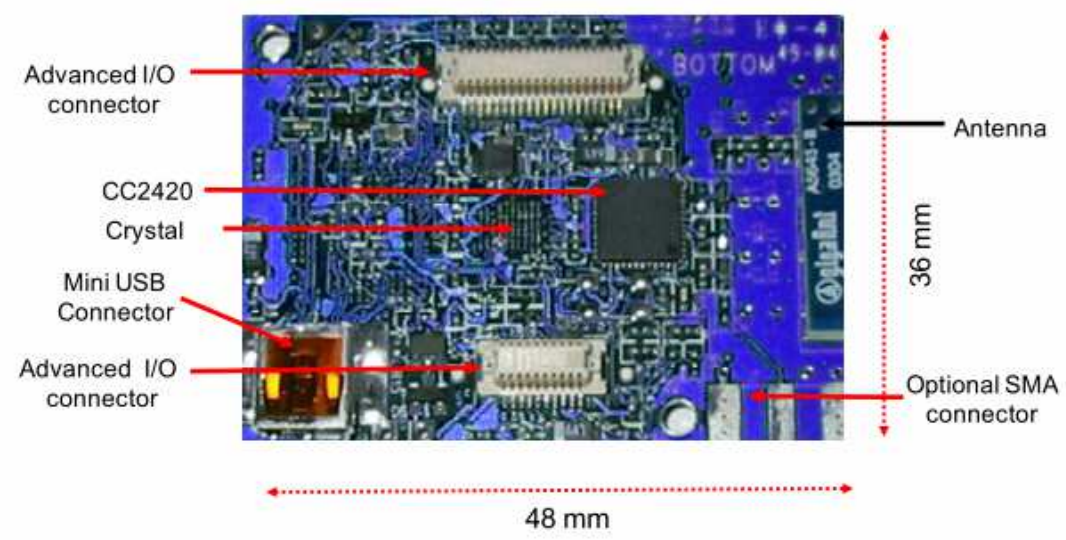

Figura 10 - Vista inferior do *Imote2*

4.1.9 **Waspmote**

A plataforma *Waspmote*, fabricada pela *Libelium*, é um nó sensor, que dispõe de um micro controlador ATmega1281, de 14MHz de frequência, 8KB de SRAM, EEPROM de 4KB, 128KB de FLASH, SD Card de 2GB e pesa 20 gramas. De consumo muito baixo, utilizando 15mA quando ligado e 55uA tanto no modo de repouso, quanto no modo de repouso profundo. Possui 7 entradas analógicas, 8

portas digitais de I/O, 2 UART, 1 I2C, 1 SPI, 1 USB, socket padrão específico para "sensores básicos" tais como temperatura, umidade e luminosidade (LDR).

Possuem uma variedade extensa de possibilidade de comunicação através de interfaces sem fio, desde o IEEE 802.15.4 / Zigbee, passando por *wifi*, além de *6LoWPAN* / IPv6 Radio, Bluetooth, 3G + GPS, GSM / GPRS, Bluetooth PRO, RFID/NFC, Placa de expansão de rádio e *Gateway Waspmote*.

As placas de sensores do *Waspmote*, englobam desde sensoriamento de gases, eventos, cidades inteligentes, estacionamento inteligente, agricultura, câmera de vídeo, radiação, medição inteligente, sensor de prototipagem e podem também utilizar placas customizadas.

A plataforma *Waspmote*, possui o conceito de programação sem fio, comummente conhecida como programação através do ar (*OTAP*), que foi usada nos anos anteriores para a reprogramação de dispositivos móveis como telefones celulares. Entretanto, com os novos conceitos de redes de sensores sem fio, M2M e a internet das coisas, onde as redes consistem de centenas ou milhares de nós, a programação através do ar é levada a uma nova direção, e primeiramente é aplicada usando ambos: tecnologias de telefones móveis, como 3G e GPRS e protocolos não licenciados como *wifi*, 802.15.4 e ZigBee.

Uma das últimas novidades do projeto Waspmote é a *6LoWPAN*. A *6LoWPAN* é um acrônimo para redes pessoais sem fio de baixa potência baseadas no IPv6. Este protocolo oferece mecanismos de encapsulamento e compressão de cabeçalho que permitem que os pacotes IPv6 sejam enviados e recebidos de redes baseadas no IEEE 802.15.4.

A fabricante IBM e a fabricante *Libelium* têm juntado esforços para oferecer uma plataforma única de desenvolvimento IPv6 para redes de sensores e Internet das Coisas. Integrando o SDK do IBM *Mote Runner* com a plataforma de sensores *Waspmote*, temos uma única e poderosa ferramenta para desenvolvedores e pesquisadores interessados na conectividade *6LoWPAN* / IPv6 para a Internet das Coisas.

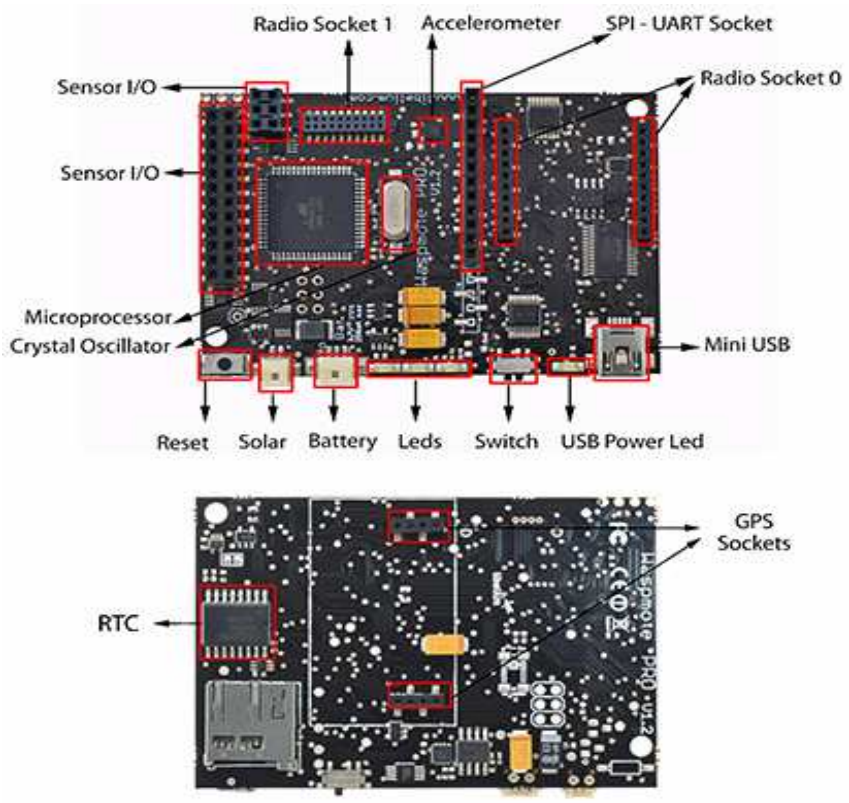

Figura 11 - A placa principal do Waspmote

4.1.10 **SunSpot**

SunSpot ou *Sun Small Programmable Object Technology (SPOT),* é uma plataforma de hardware e de software desenvolvida pela *Sun*, com o objetivo de prover a uma RSSF, todos os componentes necessários para os melhores resultados em sensoriamento. Abaixo seguem alguns detalhes de hardware desta plataforma.

4.1.10.1 Arquitetura

Um *SPOT* é composto de um conjunto de sensores, uma rede sem fio para comunicação e uma bateria recarregável. Sendo quase totalmente programado em Java, permite o desenvolvimento de aplicações no formato Java ME. Além disso, pode ser utilizado em diversas plataformas (Windows, Linux, Macintosh e Solaris).

Como protocolo de rádio, segue o padrão IEEE 802.15.4 para redes WPAN (*Wireless Personal Area Network*). Esse mesmo protocolo serve como base para a implementação de outros dispositivos para redes de sensores, tais como o ZigBee e MiWi. Entretanto, esses são construídos como uma camada em cima do padrão, sendo assim incompatíveis. Nota-se ainda, que nada impede a implementação de uma camada de compatibilidade para a comunicação entre eles.

O kit *Sun SPOT* custa aproximadamente 750 dólares nos Estados Unidos, e está disponível apenas para EUA, Europa, Rússia e Canadá. Nestes países é possível solicitar um desconto para estudantes, ficando o preço em 300 dólares neste caso.

Detalhes técnicos dos SPOTs:

- Dimensões: 41 x 23 x 70 milímetros;
- Peso: 54 gramas;
- Processador: ARM920T 32-bit, 180Mhz *clock*;
- Memória RAM: 512Kb;
- Memória: 4Mb flash;

- Radio: 2.4Ghz, compatível com as normas IEEE 802.15.4 e TI CC2420;
- Conexão USB;
- Bateria: *Lition-ion* 750mAh, 3.7V. Tempo de vida varia entre 3 a 7 horas de operação constante. Até 900 dias com processador e radio desligados.

4.1.10.2 Comunicação

A comunicação é feita através de uma conexão sem fio entre dois nós. Esta pode ser baseada em dois modelos, *streams* ou datagramas.

Streams: Este tipo de conexão é comparável com um *socket*, provendo um protocolo confiável e baseado em *streams* para a troca de mensagens entre os *SPOTs*.

Para estabelecer uma comunicação é necessário que dois componentes abram uma conexão na mesma porta e com endereços complementares. Isto é, abrir uma conexão com o identificador do componente alvo. A porta pode variar entre 0 e 255 e um endereço é representado em 64 bits conforme o padrão IEEE para radio. O código abaixo exemplifica a abertura de conexão.

StreamConnection conn = (StreamConnection);

Connector.open (”radiostream://nnnn.nnnn.nnnn.nnnn:xxx”);

Após a conexão estabelecida, cada um pode criar *streams* para enviar e receber dados.

DataInputStream dis = conn.openDataInputStream ( );

DataOutputStream dos = conn.openDataOutputStream ( );

Datagramas: A conexão é aberta de forma similar aos *streams*, através de um endereço e porta. As portas de 0 a 31 são reservadas para uso do sistema e não devem ser utilizadas.

Da mesma forma, apos aberta a conexão pode-se enviar datagramas entre os pares, conforme ilustrado abaixo:

DatagramConnection conn = (DatagramConnection);

Connector.open (”radiogram://” + targetIEEEAddress + ” :100 ”);

Datagram dg = conn.newDatagram (conn.getMaximumLength ( )) ;

dg.writeUTF (”My message”);

conn.send (dg);

Nota-se que a palavra-chave *radiogram* e *radiostream* identifica qual tipo de protocolo será usado.

Outra possibilidade é enviar pacotes em modo *broadcast* para todos os componentes que escutam em determinada porta, entretanto nesse caso não há garantias sobre a entrega dos dados.

DatagramConnection sendConn = (DatagramConnection);

Connector.open (”radiogram://broadcast:100”);

dg.writeUTF (”My message”);

sendConn.send (dg);

É possível determinar ainda, o número máximo de *hops* utilizado no broadcast, através da primitiva *setMaxBro.*

Figura 12 - Sun Small Programmable Object Technology (SPOT)

4.2 OUTRAS PLATAFORMAS MENOS POPULARES

As plataformas a seguir, seguindo as mesmas definições detalhadas na seção 4.1, são abordadas de uma forma mais resumida, por serem plataformas para RSSF voltadas para aplicações mais específicas, portanto, não sendo plataformas de uso comum e geral pelo mercado, ou pela comunidade acadêmica.

4.2.1 **Shimmer**

Shimmer - (Secure Health with Intelligence, Modularity, Mobility & Experimental Reusability) - (Saúde Segura com Inteligência, Modularidade, Mobilidade e Reusabilidade Experimental) é totalmente integrado com *BioMOBIUS*, um ambiente de aplicação de alto nível que habilita rápida prototipagem. Ele é uma plataforma de sensores sem fio para pesquisas biomédicas não invasivas. O Projeto da plataforma *Shimmer* compreende uma placa base que provê capacidades como: computação de sensores como um sensor de vibração de inclinação passiva, um sensor PIR é utilizado como gatilho de ativação de economia de energia quando o usuário se aproxima. Armazenamento de dados que facilita a gravação de dados para o cartão *MicroSD*. Comunicações incluem um rádio CC2420, Bluetooth e IEEE 802.15.4. E ele também tem incluída uma conexão com uma placa filha. *Shimmer* também suporta TinyOS-2.x.. *Shimmer* é uma plataforma de sensores extremamente flexível. Ela tem a capacidade de expandir de forma transparente para atender várias pesquisas biomédicas.

4.2.2 **Ember EM250**

O EM250 inclui 128 kB de memória flash somente leitura (ROM) embutida. Ele também permite três modos diferentes de operação. A operação ativa vai permitir a execução do código do programa, usando tipicamente 8,5 mA de corrente. A operação ociosa vai permitir que o micro controlador desligue-se até que uma interrupção ocorra, enquanto permite que os periféricos e o transceptor operem normalmente. O EM250 também permite uma operação de repouso profundo que desliga o micro controlador e o transceptor até, tanto uma interrupção externa, quanto um temporizador, acordarem o dispositivo. Na operação de repouso profundo, o EM250 tem quatro conversores analógico-digital, onde dois são usados

para capturar dados analógicos. O EM250 também tem a capacidade de se comunicar através da interface serial.

4.2.3 **TelosB**

O preço unitário do *TelosB* é alto, cerca de $150 cada, e não existe desconto para propósito educacional. A vida útil de sua bateria gira em torno de 3 a 6 meses, dependendo de quão frequentemente o sinal é transmitido de volta para o servidor, que são curtas para aplicações médicas. Seus rádio componentes não podem ser aperfeiçoados (nós não podemos usar uma antena ou transceptor melhor para atingir longas distâncias). Ele é um módulo sem fio de baixíssima potência designado para aplicações de redes de sensores. A plataforma *mote* oferece RAM embutida de 10 kB e também provê um rádio *Chipcon* IEEE 802.15.4 com uma antena integrada, provendo até 125 metros de alcance, estruturada em um micro controlador TI MSP430. O mote *TelosB* também é referenciado como sendo o *Tmote Sky.*

4.2.4 **eZ430-RF2500**

O eZ430-RF2500 é uma ferramenta de desenvolvimento sem fio MSP430 baseada completamente em USB, provendo todo o hardware e software para avaliar o micro controlador MSP430F2274 e transceptor sem fio CC2500 2,4GHz. O custo do software depurador e desenvolvimento da ferramenta é de $ 29, e a placa principal, ou seja, *mote,* é de $ 20.

O depurador é discreto, permitindo que o usuário execute uma aplicação a toda a velocidade tanto com os pontos de interrupção de *hardware* e único passo disponível ao consumir nenhum recurso de hardware extra.

A placa principal eZ430-RF2500 é um sistema sem fios “fora-da-caixa”, que pode ser utilizado com interface de depuração USB, como um sistema isolado ou sem sensores externos, ou pode ser incorporado dentro de um projeto existente.

As características do eZ430-RF2500 incluem depuração USB e interface de programação caracterizando uma instalação sem drivers e retorno da aplicação. Ele tem 21 pinos de desenvolvimento disponíveis. Altamente integrado, o micro controlador de baixíssimo consumo MSP430, com 16 MHz de desempenho, também é encontrado nele. Dois pinos digitais de I/O de uso geral, são conectados aos leds verde e vermelho para retorno visual. Botão interruptível para *feedback* do usuário está presente.

O MSP430F2274 tem como vantagens: 16-MIPS de desempenho, conversor analógico-digital de 10-bit, dois amplificadores operacionais embutidos, relógio “cão-de-guarda”, dois relógios de 16-bits, módulo USCI suportando UART/LIN, 2 SPI, I2C ou infravermelho, cinco modos de economia de energia, usando menos que 700 nA em modo de espera.

O CC2500 está tendo as vantagens de um transceptor de RF de 2,4GHz, taxa programável de até 500 kbps, e consumo de corrente baixa da placa eZ430-RF2500 projetado para otimizar fatores. O eZ430-RF2500 pode ser utilizado como uma ferramenta de desenvolvimento isolada. A placa principal, caracteriza um hardware MSP430F2274, e muitos de seus pinos são facilmente acessíveis.

A interface de depuração USB do eZ430-RF pode ser utilizada como uma ferramenta de emulação de memória flash padrão, através da sua interface Spy-*by*-*Wire*. A interface de depuração USB eZ430-RF suporta as seguintes famílias MSP430:

- MSP430F20xx;
- MSP430F22xx;

4.2.5 **Kit de desenvolvimento CC1110 e CC2510**

O CC1110 e CC2510 são dispositivos Sistemas-em-Chip (SeC) da *Texas Instruments,* projetados para aplicações sem fio de baixo consumo de energia. O CC1110 opera abaixo da banda não-licenciada ISM de 1GHz, enquanto que o CC2510 opera na banda não-licenciada ISM de 2,4GHz. Os CC1110 e CC2510 combinam o excelente desempenho dos transceptores de RF CC1101 e CC2500 respectivamente, com um micro controlador 8051 aperfeiçoado, padronizado de fábrica, com até 32 kB de memória flash programável e até 4 kB de RAM, e muitas outras características poderosas. Isto é assegurado por vários modos de operação de baixa potência avançados.

4.2.6 **Projeto Macro Motes (COTS Dust)**

Os pesquisadores da Universidade de Berkeley desenvolveram nós sensores conhecidos como *Motes*. Um dos principais objetivos do projeto desses dispositivos é o baixo consumo de energia. Os nós sensores *Motes* podem ser encontrados sob diferentes versões, tamanhos e características. A primeira geração, implementada como projeto de tese de Seth Hollar, em 2000, é conhecida como *Macro Motes* ou *COTS Dust Mote*. Existem algumas variações de projeto *Macro Motes*, que são conhecidas como *WeC Mote, RF Mote, Laser Mote, CCR Mote, Mini Motes, MALT Motes e IrDA Motes*. O transceptor RF é o TR 1000, que opera em freqüência 916,5 MHz, com capacidade de transmitir em média 10 kbps. O sistema operacional

destes nós, é o *TinyOS* (ver seção 5.2), que é dirigido a eventos e ocupa apenas 178 bytes de memória.

Em seguida aos projetos *Macro Motes*, os pesquisadores projetaram o *Rene Motes* e finalmente, a última geração de desenvolvimento, formada pelos *MICA Motes* (ver seção 4.1.5) e *Smart Dust* (ver seção 4.2.7).

4.2.7 **Projeto Smart Dust**

O Projeto *Smart Dust* [9] é desenvolvido pela Universidade de Berkeley e tem por objetivo reduzir o tamanho dos nós sensores, para que estes apresentem as dimensões de um grão de poeira, ou seja, um cubo de aproximadamente um milímetro. Os componentes disponíveis para este dispositivo são: um sensor, uma bateria, um circuito analógico, um dispositivo de comunicação óptica bidirecional e um microprocessador programável.

A comunicação através de transceptores de Rádio Freqüência (RF) é bastante inadequada para os nós deste tipo, devido a vários aspectos. Um deles é o fato de que as antenas seriam muito grandes para os *Smart Dust*, e outro é o consumo de energia, que seria alto para a disponibilidade do nó. Assim sendo, a transmissão óptica é a mais adequada, e é utilizada tanto na forma passiva quanto ativa [9].

4.2.8 **Projeto MicroAmps**

Os pesquisadores do *Massachusetts Institute of Technology (MIT),* são os responsáveis pelo desenvolvimento do *µAMPS*. Os nós sensores *µAMPS (µ-Adaptative Multi-domain Power Aware Sensor)* [9] possuem uma política de gerenciamento de energia, conhecida por *power-aware* ou *energy-aware*, que

permite que o nó sensor seja capaz de fazer com que seu consumo de energia se adapte às características e variações do ambiente onde se encontrados recursos que ele próprio dispõe e das requisições dos usuários da rede. Esta metodologia é, portanto, ideal para aplicações onde existem muitas variações no ambiente. O nó sensor *µAMPS* utiliza o rádio transceptor LMX3162 da *National Semiconductor*. Opera na banda ISM na freqüência 2,45 GHz, e consegue um alcance entre 10 e 100 metros e taxa de 1 Mbps, em transmissões sem fio, ponto a ponto. A camada de enlace utiliza TDMA, e está integrada ao rádio PCB, e age como um bloco de memória de armazenamento.

4.2.9 **Projeto WINS**

O *Rockwell Science Center* em colaboração com pesquisadores da Universidade da Califórnia, Los Angeles (UCLA), desenvolveram o protótipo de um nó sensor, chamado *WINS* [9]. O dispositivo combina capacidade de sensoriamento (tais como sísmica, acústica e magnética) com um processador RISC embutido e um rádio de transmissão. O módulo do rádio usa o *Conexant* RDSSS9M que implementa uma comunicação RF *spread spectrum* a uma freqüência de 900 MHz (ISM). O rádio opera em um dos 40 canais, escolhido pelo controlador. O alcance do rádio pode ultrapassar os 100 metros. A camada de enlace (MAC) utiliza TDMA permitindo uma taxa de 100 kbps. Os pesquisadores da *Rockwell* desenvolveram *softwares* para os protocolos básicos de comunicação, um *Kernel runtime*, *drivers* para os sensores, aplicações para processamento de sinais e APIs.

4.2.10 **Projeto JPL**

O *Jet Propulsion Laboratory (JPL)* [9] do *California Institute of Technology* está desenvolvendo um projeto chamado *SensorWeb*. Este projeto consiste em um

sistema sem fio, com nós sensores que comunicam-se entre si, distribuídos espacialmente, que podem ser dispostos para monitorar e explorar novos ambientes. O laboratório JPL foi formado para atender aos interesses da NASA, que tem como meta a exploração do *SensorWeb* em diversas aplicações.

O alcance do rádio de transmissão RF pode chegar a 40 metros, com uma taxa de transmissão de 20 kbps a uma freqüência de 916 MHz.

4.2.11 **Projeto Medusa**

Medusa MK-2 [9] é um nó sensor desenvolvido no Laboratório de Engenharia Elétrica da Universidade da Califórnia, com objetivo de se fazer testes reais de RSSFs, que operam sem a supervisão humana. O rádio possui uma potência de transmissão de 0,75 mW e seu alcance pode chegar aos 20 metros. A taxa de transferência pode variar de 2,4 kbps até 115 kbps. A comunicação é feita através de um rádio TR1000 e um barramento serial RS-485.

4.2.12 **Projeto SCADDS**

O *Scalable Coordination Architectures for Deeply Distributed Systems (SCADDS)* [9] é um projeto de pesquisa da USC/ISI que visa abordar arquiteturas escaláveis de coordenação em sistemas distribuídos e dinâmicos, em especial, RSSFs. Esse projeto envolve localização, sincronização de relógio, auto configuração e comunicação em RSSFs. O projeto não propõe arquitetura de hardware e nem utiliza uma em especial. Dentre as utilizadas como plataforma de teste está a baseada no *Mica Motes*, abordada neste trabalho. Quanto aos protocolos de comunicação utilizados, destacam-se o S-MAC, na camada de enlace, e o *Direct Diffusion*, na camada de rede. O S-MAC adota um esquema TDMA para

comunicação entre vizinhos, o que representa uma vantagem em RSSFs por permitir economia de energia nos instantes onde não há comunicação. Esse protocolo tenta atender alguns requisitos de dinamicidade da rede, como inclusão de nós e tolerância a falhas, porém parece não atender bem redes com nós móveis. O *Direct Diffusion*, tradicional algoritmo de disseminação de dados em RSSFs, propõe um esquema de roteamento centrado em dados, onde não há semântica de endereçamento. Ele também tenta atender redes dinâmicas através de um esquema de negociação, com disseminação de interesses e reforços de caminhos, permitindo à rede convergir perante qualquer alteração topológica. Sua grande desvantagem é o alto custo de comunicação, devido à necessidade de se disseminar interesses periodicamente em toda a rede, e disseminar dados até que um caminho seja reforçado. Em simulações realizadas pelo grupo de pesquisa do projeto *SensorNet* da UFMG [9] (ver seção 4.2.13), foi demonstrado que o algoritmo sofre muito com perdas de pacotes quando a quantidade de nós enviando dados começa a aumentar, e a taxa de envio de dados é alta. Com uma pilha montada desta forma, tem-se a vantagem de trabalhar com uma plataforma projetada especialmente para RSSFs, porém, observa-se a limitação a aplicações de redes estáticas e com disseminação de dados sob requisição (interesses). Logo, redes de tráfego contínuo e, principalmente, orientadas a eventos, devem empregar outros algoritmos da camada de rede.

4.2.13 **Projeto SensorNet**

O *SensorNet* [9] é um projeto do Instituto de Tecnologia da Geórgia com o propósito específico de desenvolver protocolos de comunicação para as RSSFs e suas características particulares. Este projeto, que usa o *Mica Motes* como

plataforma de teste, não propõe uma solução de pilha de protocolos específica, mas apresenta vários protocolos individualmente que poderiam compô-la, como o ESRT [23], para camada de transporte, o SER [23] e o QSR [23], para roteamento e o CMAC [23] na camada MAC. O principal requisito visado por esses protocolos é eficiência energética e confiabilidade na entrega de dados, mas não é apresentado como as pilhas de protocolos são montadas e qual o desempenho do conjunto.

4.2.14 **Projeto BEAN**

Os pesquisadores do projeto *SensorNet* do DCC/UFMG estão desenvolvendo um nó sensor usando componentes de prateleira. No mesmo projeto está em desenvolvimento a plataforma computacional chamada de *BEAN (Brazilian Energy-Efficient Architectural Node)*, que servirá como protótipo de um nó sensor. O micro controlador utilizado é da família MSP430, que tem baixíssimo consumo de energia, além de ser 16bits (8 MIPS), e possuir vários modos de operações, e ser equipado com um conjunto completo de conversor analógico-digital, facilitando a integração dos dispositivos sensores. O rádio utilizado será o CC1000 (o mesmo do *Mica2 Mote*). Uma memória serial *flash* externa (STM25P40) que serve como memória secundária também será utilizada. Outro componente utilizado é o ds2417 que servirá como um relógio de tempo real, além de prover um número identificador único de 48-bits. O sistema operacional deste projeto também está sendo desenvolvido pelos pesquisadores da UFMG e foi batizado de *YATOS (Yet Another Tiny Operating System)*. Ele é dedicado ao nó sensor BEAN e dirigido a eventos. Uma das vantagens em relação ao *TinyOS* (ver seção 5.2) é que o *YATOS* possui prioridade entre tarefas.

4.2.15 **MillennialNet**

A *Millennial* [9] possui uma solução de RSSFs composta de nós com funções especiais. São eles: *Endpoints*, para realizar o sensoriamento; *Routers*, para estender a área monitorada através de *multi-hop*; e *Gateway*, para conexão da RSSF com redes externas. A idéia da rede da *Millennial* é a criação de uma rede auto-organizável, que automaticamente cria e mantém uma topologia de rede mesmo com a ocorrência de alterações topológicas, com tempo de vida longo (anos), exigindo operação com baixíssimo consumo de energia. A solução da *Millennial* é fechada, não permitindo uma análise mais profunda. Porém, algumas características nos dão algumas dicas. Entre as possibilidades de rádio, está o IEEE 802.15.4 [9], para redes pessoais sem fio, prometendo baixo consumo, porém, baixa largura de banda. Já no roteamento, se tem um protocolo com patente pendente com a idéia de usar a arquitetura com nós roteadores para dar maior confiabilidade à entrega de pacotes formando uma infraestrutura em malha. O protocolo opera com baixas taxas de coleta de dados e, ao que tudo indica, simplesmente tem a função de entregar pacotes ao *gateway* pelo menor caminho (menor número de *hops*). Pelas poucas informações obtidas, a arquitetura fechada da *Millennial* se aplica somente a casos de simples coleta de dados com tráfego contínuo, não sendo possível adequá-la a soluções de processamento colaborativo distribuído, porém, já disponibiliza uma rede com longo tempo de vida.

4.2.16 **Projeto PicoRadio**

O *PicoRadio* [9] está sendo projetado na Universidade de Berkeley, e é um tipo de nó micro sensor conhecido como *picoSensor*. Este tipo de dispositivo é projetado com o objetivo de que a dissipação de energia do sensor, tanto em

processamento, quanto em comunicação, seja extremamente baixa. Portanto, os limites aceitáveis em relação à energia são de 10pJ por bit corretamente transmitido ou processado, e em relação à potência, o máximo é de 1 mW.

Para que tais objetivos possam ser alcançados, as seguintes estratégias são utilizadas:

- *Energy scavenging* (técnica cujo objetivo é conseguir que o nó sensor retire o máximo de energia possível do ambiente onde se encontra, como por exemplo, energia solar ou energia de vibrações);

- Baixo consumo de energia na arquitetura do *PicoSensor* e seus circuitos, ou seja, projeto e desenvolvimento de componentes de baixo consumo de energia;

- Projeto de sistema operacional dirigido a eventos (resultados de testes mostraram que este tipo de sistema pode ser mais econômico do que sistemas operacionais de propósito geral, no que diz respeito a energia). A largura de banda é de 5 GHz e a camada de enlace (MAC) utiliza TDMA.

Tabela 6 - Chips MCU de redes de sensores sem fio

Corp.	Chip Type	CPU	RAM (KB)	Flash (KB)
Silicon	80C51/C8051F	CIP-51	8	128
Microchip	PICF18F4620	PIC	4	64
Freescale	MC9S08GT	HCS08	4	60
	MCF5222x	ColdFire®	32	256
Atmel	ATMEGA128L	RISC	4	128
	AT91	ARM	256	1024
Intel	8051	MCS-51	1	16
	PXA27X	XScale®	256	32
TI	MSP430F413	MSP430	10	48`
Samsung	S3C44B0	ARM	8	N/A
OKI	4050/4060	ARM	A6	128

Tabela 7 - Nós da rede de sensores sem fio

Node	MCU	RFModule	Organization
MICAz	Atmega1281	CC2420	UCB
Telos	MSP430	CC2420	Moteiv
M2020	MSP430	CC2420	Dust Inc.
BSN Node	MSP430	CC2420	Imperal
CIT Node	PIC16F877	NordicnRF903	CIT
MIT Node	8051	NordicnRF24	MIT
Pluto	MSP430	CC2420	Harvard
iMote	ARM7TDMI	Bluetooth	Intel
iMote2	Intel PXA	CC2420	Intel
EmberNet	Atmega1281	Ember250	Ember
IP-Link	MSP430	CC2420	Helicomm
Spot	ARM	CC2420	Sun
Zbnode	ARM	CC2420	Taiwan ITRI
XYZ	ARM	CC2420	Yale
WINS	PXA255	802.11b	Sensoria
Embernet	ATmega1281	Ember250	Ember
Cicada1	MC9S08GT60	MC13193	Tsinghua
Cicada2	MC13193		Tsinghua

4.3 ANÁLISE COMPARATIVA DAS PRINCIPAIS PLATAFORMAS

Tabela 8 e a tabela 9 sumarizam as principais características das plataformas de nós sensores. O *Imote2*, a família de motes *Mica* (*MicaZ, Mica2 e Mica2dot*) e o *BTnode*, oferecem as plataformas mais flexíveis do “estado da arte”. Além disso, elas oferecem a vantagem da experiência com vários desenvolvimentos e suporte dos fabricantes e outros usuários.

O *BTnode* tem a vantagem em relação a família *Mica*, de permitir um desenvolvimento de protocolos customizados mais flexível. O *Imote2* e a família *Mica* tem a vantagem sobre o *BTnode*, de suporte garantido da *Crossbow*. O *Imote2* oferece um processador mais eficiente em energia do que a família *Mica*, e flexibilidade ao incorporar múltiplas operações de rádio, enquanto tem um rádio complacente com 802.15.4. Tanto *Imote2* e a família *Mica* são considerados mais eficientes em energia do que o *BTnode*.

Tabela 8 - Resumo das características de plataformas de nós sensores (Parte 1)

	Platform		
Features	MSB430	Tmote Sky	BTnode
MPU@[MHz]	MSP430F1612IPM@11	MSP430 F1611@8	ATmega128L@8
MPU Memory	55 kB Flash, 5 kB RAM	48 kB Flash, 10 kB RAM	128 kB Flash, 64+180 kB SRAM and 4 kB EEPROM
Radio	TI CC1020	TI CC2420 (IEEE 802.15.4)	Zeevo ZV4002 (Bluetooth 1.2), TI CC1000
Freq. Band [MHz]	402 – 470, 804 - 940	2400-2483	2400-2483 (ZV4002) 433-915 (CC1000)
Data Rate [kbs]	153.6 (19 typical)	250	1000 (ZV4002), 76.8 (CC1000)
Software	C/SactterWeb OS, NesC/TinyOS	NesC/TinyOS	C/BTnut OS, NesC/TinyOS
Support Rating	3/5	2/5	4/5
Development Interface	Open access	Constrained access	Open access
Remote Programming	Via wireless USB cards and gateway (in-band)	Via wireless USB cards and gateway (out-of-band)	Via in-built secondary radio (e.g. Bluetooth)
Size [cm]	3.6x4.1	3.2×6.55×0.66	5.8x3.3

Tabela 9 - Resumo das características de plataformas de nós sensores (Parte 2)

	Platform				
Features	EyesIFXv2	MICAz	MICA2	MICA2DOT	Imote2
MPU@[MHz]	MSP430 F1611@8	ATmega128L @8	ATmega128L @8	ATmega128L @8	IPXA271@13 – 416MHz
MPU Memory	48 kB Flash, 10 kB RAM	128+512 kB Flash, 4 kB SRAM	128+512 kB Flash, 4kB SRAM, 4 kB EEPROM	128+512 kB Flash, 4kB SRAM, 4 kB EEPROM	32MB Flash, 256kB SRAM, 32 MB SDRAM
Radio	TDA5250	TI CC2420 (IEEE 802.15.4)	TI CC1000	TI CC1000	TI CC2420 (IEEE 802.15.4)
Freq. Band [MHz]	868–870	2400-2483	868-916 (spectral-agile)	868-916, 433 or 315 (spectral-agile)	2400-2483
Data Rate [kbs]	64	250	38.4	38.4	250
Software	NesC/TinyOS	MoteWorks (TinyOS based)	MoteWorks (TinyOS based)	MoteWorks (TinyOS based)	TinyOS, Linux, SOS
Support Rating	3/5	5/5	5/5	5/5	5/5
Development Interface	Open access	Constrained access	Constrained access	Constrained access	Open access with add-on transceivers
Remote Programming	Via wireless USB cards and gateway	Via 802.15.4 gateway	Via gateway	Via gateway	Via wireless cards and gateway
Size [cm]	3.2×6.55×0.7	5.8x3.2x0.7	5.8x3.2x0.7	2.5x0.6	3.6x4.8x0.9

Tabela 10 - Comparativo entre diversos nós sensores

Mote type	WASP Mote	FireFly	EPIC mote	Mica2	Telos	Imote2	EyesIFX	SUNspot	Shimmer
Micro controller	ATmega1281	Atmel ATmega 1281	Texas Instruments MSP430 microcontroller	ATmega 128	TIMSP430	PXA271 XScale® Processor	TI MSP430	ARM920T core	MSP430F1611
Radio	8MHz 802.15.4 modules (2.4GHz, 868MHz, 900MHz	916 MHz	2.4 GHz IEEE 802.15.4	315 MHz, 433 MHz, and 868/916 MHz options	8MHz, 2.4 to 2.4835 GHz	13–416 MHz	868 MHz	180 MHz	8MHz, 2.4GHz
Data rate	156.25 kbps	250 kpbs	250 kbps	38.4 kbps	250 kbps	250 kbps	64 kbps	250 kbps	80 Kbps
RAM	8KB SRAM	8K RAM	10k RAM	4KB	10Kbyte RAM	256kB SRAM, 32MB SDRAM	10k RAM	512K RAM	10Kbyte RAM
Flash	128KB	128 KB	48k Flash	128 KB	48 Kbyte	32 MB	48 KB	4M Flash	48 Kbyte
Modulation type	Frequency Hopping	O-QPSK	O-QPSK (MSK)	BPSK	O-QPSK	O-QPSK	ASK/FSK	O-QPSK	O-QPSK
	Spread Sequence (FHSS)								
Sensors	Chemical, temperature, pressure, humidity, luminosity, magnetic, Soil Temperature / Moisture	light, temperature, audio, passive infrared motion, dual axis acceleration and voltage sensing	Temperature, pressure, humidity, luminosity, magnetic, home energy tracking, solar-powered environmental monitoring	Light (Photo), Temperature, Acceleration, Magnetometer, Microphone, Tone Detector, Sound	Temperature, humidity and light sensors	Wireless MMX DSP Coprocessor, I2S, AC97,for audio applications, Camera Interface for image and video applications	Temperature and light sensors	2G/6G three-axis accelerometer, Temperature and Light sensors	3-Axis Accelerometer, Gyro, Magnetometer, ECG, EMG, GSR, GPS/Temperature, or Strain Gauge modules
Radio	XBee-802.15.4	CC2420	Chipcon CC1000 Wireless Transceiver	CC1000	CC2420	CC2420 IEEE 802.15.4 radio transceiver	Infineon TDA5250	CC2420	CC2420

5 **SISTEMAS OPERACIONAIS PARA REDES DE SENSORES SEM FIO**

Este capítulo aborda os principais sistemas operacionais para RSSF, que levam em consideração diversos aspectos: processamento, economia de energia, otimização do uso da memória, número de tarefas executadas simultaneamente, entre outras, que são questões muito relevantes no que concerne à adequação do SO para o hardware utilizado, bem como o ambiente à ser monitorado.

5.1 PRINCIPAIS PREOCUPAÇÕES NO PROJETO DE SO PARA RSSF

Esta seção discute os detalhes de maiores questões relacionadas ao projeto de Sistemas Operacionais para RSSF.

5.1.1 **Arquitetura**

A organização de um SO constitui sua estrutura. A arquitetura de um sistema operacional tem uma influência sobre o tamanho do núcleo do SO, bem como sobre a forma como provê serviços para os programas da aplicação. Algumas das arquiteturas de SO bem conhecidas, tem uma arquitetura monolítica, arquitetura de micro-núcleo, arquitetura de máquina virtual e arquitetura em camadas.

Uma arquitetura monolítica, de fato, não tem nenhuma estrutura. Serviços providos por um SO são implementados separadamente e cada serviço provê uma interface para outros serviços. Como uma arquitetura permite agregação de todos os serviços requeridos juntos dentro de uma imagem de sistema única, assim resulta em um menor consumo de memória pelo SO. Uma vantagem da arquitetura monolítica é que os custos da interação dos módulos são baixos. Desvantagens associadas com esta arquitetura são: dificuldade no entendimento e alterações no sistema, não confiabilidade, e dificuldade na manutenção. Estas desvantagens

associadas com núcleos monolíticos fazem dele uma escolha pobre de projeto de SO para nós sensores contemporâneos.

Uma escolha alternativa é uma arquitetura de micronúcleo em que funcionalidade mínima é provida dentro do núcleo. Assim, o tamanho do núcleo é reduzido significativamente. Muitas das funcionalidades do SO são providas via servidores a nível de usuário, como um servidor de arquivos, um servidor de memória, um servidor de tempo, etc. Se um servidor falha, o sistema inteiro não cai. A arquitetura de micronúcleo provê melhor confiabilidade, facilidade de extensão e customização. A desvantagem associada com um micronúcleo é seu desempenho pobre por causa de usuário frequente para travessias de fronteira de núcleo. Um micronúcleo é a escolha de projeto para muitos SOs embarcados devido ao seu tamanho pequeno do núcleo e o número de trocas de contexto numa aplicação de RSSF típica, é considerada bem pequena. Assim, poucas travessias de fronteira são requeridas, comparadas à sistemas tradicionais.

Uma máquina virtual é outra escolha arquitetural. A idéia principal é exportar máquinas virtuais para programas de usuário, que se assemelham a *hardware*. Uma máquina virtual tem todas as características de *hardware* necessárias. A principal vantagem é sua portabilidade e a principal desvantagem é tipicamente um desempenho de pobre de sistema.

Uma arquitetura de SO em camadas implementa serviços em forma de camadas. Vantagens associadas com a arquitetura em camadas são: gerenciamento, facilidade de compreensão, e confiabilidade. A principal desvantagem é que ela não é uma arquitetura muito flexível de uma perspectiva de projeto de SO.

Um sistema operacional para redes de sensores sem fio, precisa ter uma arquitetura que resulte em um núcleo de tamanho pequeno, portanto consumo pequeno de memória. A arquitetura tem que permitir extensões ao núcleo se requeridas. A arquitetura tem que ser flexível, isto é, apenas serviços requeridos pela aplicação são carregados dentro do sistema.

5.1.2 **Modelo de programação**

O modelo de programação suportado por um SO tem um impacto significante no desenvolvimento da aplicação. Existem dois modelos de programação populares providos por SOs para RSSF, são eles: programação orientada a eventos, e programação multitarefa. Multitarefa é o modelo de desenvolvimento de aplicação mais familiar ao programador, mas em seu verdadeiro sentido, em vez de recurso intenso, portanto não é considerado bem adequado para dispositivos de recursos limitados como nós sensores. A programação orientada a eventos é considerada mais útil para dispositivos de computação equipados com poucos recursos, mas não é considerada conveniente para desenvolvedores tradicionais de aplicações. Entretanto, pesquisadores têm focado sua atenção no desenvolvimento de um modelo de programação multitarefa leve para SOs para RSSF. Muitos SOs para RSSF contemporâneos hoje provêem suporte para o modelo de programação multitarefa e discutir-se-á em detalhes mais tarde.

5.1.3 **Escalonamento**

A seleção de um algoritmo de escalonamento apropriado para RSSFs depende tipicamente da natureza da aplicação. Em sistemas de computação tradicionais, o objetivo de um escalonador é minimizar a latência para maximizar a saída e utilização de recursos, e garantir equidade. O escalonamento da unidade

central de processamento (CPU) faz isso determinando a ordem que as tarefas são executadas numa CPU.

Para aplicações que tem requisitos de tempo real, como as redes de sensores sem fio, o algoritmo de escalonamento de tempo real tem que ser utilizado. Para outras aplicações, algoritmos de escalonamento de tempo não-real são suficientes.

As RSSFs estão sendo utilizadas tanto em ambientes de tempo real, quanto de tempo não-real, por isso, um SO para RSSF tem que prover algoritmos de escalonamento que possam acomodar os requisitos da aplicação. Além disso, um algoritmo de escalonamento adequado tem que ser eficiente em memória e energia.

5.1.4 **Gerenciamento e proteção de memória**

Em um sistema operacional tradicional, o gerenciamento de memória se refere à estratégia utilizada para alocar e desalocar memória para processos e threads diferentes. Duas técnicas de gerenciamento de memória normalmente utilizadas são: o gerenciamento estático de memória e o gerenciamento dinâmico de memória. O gerenciamento estático de memória é simples e é uma técnica útil quando se trata de recursos escassos de memória. Ao mesmo tempo, isto resulta em sistemas inflexíveis porque a alocação de memória em tempo de execução não pode ocorrer. Por outro lado, o gerenciamento dinâmico de memória, produz um sistema mais flexível porque a memória pode ser alocada e desalocada em tempo de execução. A proteção da memória do processo se refere a proteger um espaço do processo do outro. Em sistemas operacionais para redes de sensores como o *TinyOS*, não existe gerenciamento de memória disponível. Os sistemas operacionais para RSSF iniciais, assumiram que apenas uma única aplicação executa num mote

sensor, entretanto, não há necessidade para proteção da memória. Com a emergência de novos domínios de aplicações para RSSF, RSSFs contemporâneas provêem suporte para execução de múltiplas tarefas, consequentemente o gerenciamento de memória se torna questão para SO para RSSF.

5.1.5 **Suporte a protocolos de comunicação**

No contexto do SO, a comunicação refere-se à comunicação entre processos e o sistema bem como com outros nós na rede. RSSF operam num ambiente distribuído, onde nós sensores se comunicam com outros nós na rede. Todos os SOs para RSSF provêem uma Interface de Programação da Aplicação (API) que permite os programas se comunicarem. É possível que uma RSSF seja composta de nós sensores heterogêneos, portanto, o protocolo de comunicação provido pelo SO tem também que considerar heterogeneidade. Em comunicação baseada em rede, o SO provê implementações de transporte, rede, e protocolo de camada MAC.

5.1.6 **Compartilhamento de recursos**

A responsabilidade de um SO inclui alocação e compartilhamento de recursos, que é de imensa importância quando múltiplos programas estão executando simultaneamente. A maioria dos SOs para RSSF hoje provêem algum tipo de multitarefa, requerendo um mecanismo de compartilhamento de recursos. Isto pode ser realizado em tempo, por exemplo, agendamento de um processo / *thread* na CPU, e no espaço, por exemplo, a gravação de dados na memória do sistema. Em alguns casos, precisamos de acesso serializado para recursos e isso é feito através do uso de primitivas de sincronização.

5.1.7 **Suporte para Aplicações de Tempo Real**

Uma RSSF pode ser usada para monitorar um sistema de missão crítica. Entretanto, um SO para RSSF pode prover implementações de algoritmos de escalonamento de tempo real para atingir os prazos de entrega de tarefas de tempo real pesadas. Com o advento das redes de sensores multimídia sem fio (RSMSF), um SO para RSSF pode prover implementações de protocolos de comunicação que suportam fluxos de multimídia de tempo real. Por exemplo, um SO pode prover uma implementação de um protocolo MAC que reduz a latência fim a fim dos fluxos multimídia, além disso, projetistas de SO devem se preocupar em prover implementações de protocolos de comunicação de tempo real nas camadas de rede e transporte. Além disso, um SO para RSSF poderia prover uma Interface de programação para aplicação (API) para os programadores de aplicação, que os possibilita implementar protocolos de comunicação customizados no topo da pilha de protocolos de comunicação suportada pelo SO. Acima de tudo, o SO tem que prover uma implementação de arquitetura de *QoS* para segregação de tráfego na camada de rede.

O custo decrescente do hardware, como o das câmeras *CMOS* e microfones, tem resultado em novas variantes de RSSFs chamados nós sensores de multimídia sem fio ou NSMSF. Os Nós NSMSF são equipados com câmeras integradas, microfones e sensores escalares. Tais nós sensores são capazes de capturar e comunicar fluxos de áudio e vídeo através de um canal sem fio. NSMSFs são mais sofisticados quando comparados a nós sensores comuns, mas ainda tem recursos limitados, comparados às plataformas de computação contemporâneas. Primeiramente, RSSFs e NSMSFs pretendem capturar e transmitir informação de

forma ubíqua para os nós sorvedouros presentes na rede. Assim, o protocolo de comunicação desempenha um papel fundamental para a funcionalidade correta de tais redes. Recursos escassos e a mídia de comunicação sem fio inibem o uso de arquitetura em camadas tradicional como a pilha de protocolo TCP/IP em RSSF. Em segundo lugar, o TCP foi projetado para redes cabeadas e seu desempenho na comunicação sem fio é relatado como pobre. Além disso, no caso de RSMSF, o TCP não é um protocolo recomendado para aplicações multimídia principalmente por seus mecanismos de controle de fluxo e congestionamento. O UDP pode ser uma alternativa para aplicações multimídia TCP mas ele não provê qualquer informação sobre o status da rede que pode ser requerido para transmissão apropriada de dados multimídia. Além disso, tanto TCP, quanto UDP, não servem como protocolos de camada de transporte ideais para RSMSF.

O paradigma do projeto de arquitetura camada sobre camada está emergindo como uma técnica promissora para redes usando comunicação sem fio. No projeto de camada sobre camada, dependendo da condição do enlace sem fio, a camada MAC pode escolher técnicas de codificação de erro apropriadas. Da mesma forma, a camada de rede pode escolher um caminho, tendo a entrada das camadas física e de aplicação. A arquitetura camada sobre camada pode adaptar o comportamento de uma aplicação às condições do enlace físico. Como resultado, pesquisadores têm desenvolvido arquiteturas camada sobre camada que pode trabalhar eficientemente para redes de sensores sem fio de aplicação única.

À luz da discussão acima, isto pode ser visto como novas áreas de aplicação emergentes de demandas adicionais de RSSFs no SO. Devido às limitações de recursos e a natureza do enlace sem fio, a abordagem de projeto de protocolo

camada sobre camada tem sido relatada por trabalhar bem. Entretanto, um SO contemporâneo para RSSF poderia prover uma implementação da pilha de protocolos, que permite a interação camada sobre camada, e otimização dos parâmetros da pilha de protocolos em acordo com os requisitos da aplicação. Um SO tem que suportar um protocolo de camada de transporte que suporte aplicações de tempo real. O protocolo de transporte de tempo real deve monitorar as condições da rede e minimizar o congestionamento dentro da rede para que fluxos de tempo real experimentem qualidade aceitável. Além disso, o SO precisa prover uma implementação de um protocolo de roteamento que construa a rota de acordo com os requisitos de *QoS* das aplicações. Mais ainda, ele deve suportar um algoritmo MAC que escalone pacotes de acordo com sua prioridade.

5.2 TINYOS

O *TinyOS* [18] é um sistema operacional de código aberto, flexível, baseado em componentes e aplicações específicas projetado para redes de sensores. O *TinyOS* pode suportar programas concorrentes com muito poucos requerimentos de memória. O SO tem um consumo que cabe em 400 bytes. A biblioteca de componentes do *TinyOS* inclui protocolos de rede, serviços distribuídos, drivers de sensores e ferramentas de aquisição de dados. As seguintes subseções examinam o projeto do *TinyOS* com mais detalhes.

5.2.1 **Arquitetura**

O *TinyOS* se enquadra na classe de arquitetura monolítica. O *TinyOS* usa um modelo orientado a componentes e, de acordo com os requisitos da aplicação, componentes diferentes são adicionados ao escalonador para compor uma imagem estática que roda no mote. Um componente é uma entidade computacional

independente que expõe uma ou mais interfaces. Componentes têm três abstrações computacionais: comandos, eventos e tarefas. Mecanismos para comunicação entre componentes são comandos e eventos. Tarefas são utilizadas para expressar concorrência intra-componentes. Um comando é uma requisição para executar algum serviço, enquanto que o evento sinaliza a conclusão do serviço. *TinyOS* provê uma pilha compartilhada única e não há separação entre o espaço do núcleo e o espaço do usuário.

5.2.2 **Modelo de programação**

Versões anteriores do *TinyOS* não provinham qualquer suporte à *multithread*, com desenvolvimento da aplicação seguindo estritamente o modelo de programação orientada a eventos. *TinyOS* versão 2.1 provê suporte para *multithread* e estas *threads TinyOS* são *chamadas threads TOS*. Em [18], os autores pontuaram o problema que, dadas as limitações de recursos dos motes, um SO baseado em eventos que permite maior concorrência. Entretanto, *threads* preemptivas oferecem um paradigma de programação intuitivo. O pacote de *threading* do *TOS* provê facilidade no modelo de programação de *threads* junto com a eficiência de um núcleo dirigido à eventos. As *Threads TOS* são retro compatíveis com o código *TinyOS* existente. As *Threads TOS* usam uma abordagem de *threading* cooperativa, isto é, as *threads TOS* confiam em aplicações para explicitamente render o processador. Isto adiciona um encargo adicional para o programador para gerenciar explicitamente a concorrência. *Threads* em nível de aplicação no *TinyOS* podem preemptar outras *threads* de nível de aplicação, mas elas não podem preemptar tarefas e manipuladores de interrupção. Uma *thread* de núcleo de alta prioridade é dedicada a executar o escalonador do *TinyOS*. Para a comunicação entre as *threads*

de aplicação e o núcleo, o *TinyOS* 2.1 provê passagem de mensagens. Quando um programa de aplicação faz uma chamada de sistema, ele não pode executar diretamente o código. Antes ele posta uma mensagem para a *thread* de núcleo postando uma tarefa. Posteriormente, a *thread* do núcleo preempta a *thread* que está rodando, e executa a chamada de sistema. Este mecanismo garante que apenas o núcleo executa diretamente o código *TinyOS*. Chamadas de sistema como: Criar, Destruir, Pausar, Continuar e Unir, são providas pela biblioteca de *threading TOS*.

As *Threads TOS* alocam dinamicamente Blocos de controle de *threads* (TCB) com espaço para uma pilha de tamanho fixo que não cresce com o tempo. Trocas de contexto de *Threads TOS* e chamadas de sistema introduzem um custo adicional menor que 0,92%

Versões anteriores do *TinyOS* impõem atomicidade desabilitando as interrupções, isto é, dizendo ao *hardware* para atrasar os eventos externos até que a aplicação tenha completado uma operação atômica. Este esquema funciona bem em sistemas de único processador. Seção crítica pode ocorrer em *threads* de nível de usuário e o projetista do SO não quer que o usuário desabilite as interrupções devido ao desempenho do sistema e questões de usabilidade. Este problema é escondido no *TinyOS* versão 2.1. Ele provê suporte a sincronização com a ajuda de variáveis de condição e exclusões mútuas. Estas primitivas de sincronização são implementadas com a ajuda de instruções especiais de *hardware*, por exemplo, instruções de testes e configuração.

5.2.3 **Escalonamento**

Versões anteriores do *TinyOS* suportavam um algoritmo de escalonamento "*First-In-First-Out*" (*FIFO*) não-preemptivo. Portanto, aquelas versões do *TinyOS* não suportavam aplicações de tempo real. O núcleo do modelo de execução do *TinyOS* é composto por tarefas que executam até completar, de uma maneira *FIFO*. Desde que o *TinyOS* suporta apenas escalonamento não-preemptivo, a tarefa tem que obedecer a semântica de executar até terminar. As tarefas executam até a conclusão em relação a outra tarefa, mas elas não são atômicas em relação a manipuladores de interrupção, comandos e eventos que elas invocam. Desde que o *TinyOS* usa escalonamento *FIFO*, desvantagens associadas com escalonamento *FIFO* também são associadas com o escalonador *TinyOS*. O tempo de espera para uma tarefa depende do tempo de chegada da tarefa. O escalonamento *FIFO* pode ser injusto para as últimas tarefas, especialmente quando as tarefas curtas estão esperando atrás das mais longas.

Em [18], os autores reivindicam que eles adicionaram suporte para o algoritmo de *Earliest Deadline First (EDF)* no *TinyOS*, para facilitar aplicações de tempo real. O algoritmo de escalonamento *EDF* não produz uma agenda viável quando as tarefas pedem por recursos. Assim, o *TinyOS* não fornece um algoritmo de escalonamento em tempo real sólido se *threads* diferentes pedirem por recursos.

5.2.4 **Gerenciamento e Proteção de Memória**

Em [18], a segurança eficiente de memória para o *TinyOS* é apresentada. Em nós sensores, a proteção de memória baseada em *hardware* não está disponível e os recursos são escassos. Limitações de recursos necessitam do uso de linguagens não seguras, e de baixo nível como *nesC*. No *TinyOS* versão 2.1, a segurança da

memória é incorporada. Os objetivos para segurança de memória dados em [18] são: apontar todos os erros de ponteiro e matriz, fornecer diagnósticos úteis e fornecer estratégias de recuperação. As implementações do *TinyOS* baseadas em memória segura, exploram o conceito de *Deputy*. O *Deputy* é um compilador “recurso a recurso”, que garante segurança de tipo e de memória para código C. O código compilado pelo *Deputy* confia numa mistura de checagens compiladas e em tempo real, para garantir segurança da memória. O *TinyOS* seguro é retro compatível com versões anteriores do *TinyOS*. A cadeia de ferramenta do *TinyOS* seguro insere checagens dentro do código da aplicação, para garantir segurança em tempo de execução. Quando uma checagem detecta que a segurança está pra ser violada, o código inserido pelo *TinyOS* seguro toma ações curativas. O *TinyOS* usa uma abordagem de gerenciamento de memória estática.

5.2.5 **Suporte a protocolos de comunicação**

Versões anteriores do *TinyOS* provêem dois protocolos de múltiplo salto: disseminação e *TYMO*. O protocolo de disseminação, confiavelmente entrega dados para qualquer nó na rede. Este protocolo provê duas interfaces: *DisseminationValue* e *DisseminationUpdate*. Um produtor chama *DisseminationUpdate*. O comando *DisseminationUpdate.change()* pode ser chamado a cada vez que os produtores quiserem disseminar um novo valor. Por outro lado, a interface *DisseminationValue* é provida para o consumidor. O evento *DisseminationValue.changed()* é sinalizado a cada vez que o valor de disseminação é mudado. *TYMO* é a implementação do protocolo *DYMO*, um protocolo de roteamento para redes móveis e *ad hoc*. Em *TYMO*, os formatos de pacotes foram mudados e isto tem sido implementado no topo da pilha de mensagens.

Em [11] foi apresentado o DIP, um protocolo novo de disseminação para redes de sensores. DIP é um protocolo de descoberta de dados e disseminação que escala a centenas de valores. O *TinyOS* versão 2.1.1 também provê suporte para o *6lowpan*, uma camada de rede IPv6 dentro de uma rede *TinyOS*.

Na camada MAC, o *TinyOS* provê uma implementação dos seguintes protocolos: um protocolo TDMA de único salto; um protocolo híbrido TDMA/CSMA que implementa as otimizações Z-MAC e B-MAC; e uma implementação opcional do IEEE 802.15.4 aderente ao MAC.

5.2.6 **Compartilhamento de Recursos**

O *TinyOS* usa dois mecanismos para gerenciamento de recursos compartilhados: virtualização e eventos. Um recurso virtualizado aparece como uma instância independente, isto é, a aplicação o usa independente de outras aplicações. Recursos que não podem ser virtualizados são manipulados através de eventos de conclusão. A pilha de comunicação *GenericComm* do *TinyOS* é compartilhada entre *threads* diferentes e não pode ser virtualizada. *GenericComm* pode apenas mandar um pacote por vez, mandar operações de outras *threads* falhas durante este tempo. Como recursos compartilhados são manipulados através de eventos de conclusão que informam *threads* em espera sobre a conclusão de uma tarefa particular.

5.2.7 **Suporte para aplicações de tempo real**

O *TinyOS* não provê qualquer suporte explícito para aplicações de tempo real. Como já discutimos na seção de escalonamento, as tarefas no *TinyOS* observam a semântica executar até concluir numa maneira *FIFO*, consequentemente em sua forma original, o *TinyOS* não é uma boa escolha para redes de sensores que estão

sendo aplicadas em fenômenos de monitoria de tempo real. Fez-se um esforço para implementar um algoritmo de escalonamento de processos *EDF* e disponibilizou-se isto em novas versões do *TinyOS*. Entretanto, mostrou-se que o algoritmo *EDF* não pode produzir uma escala viável quando tarefas pedem por recursos. No *nutshell*, o *TinyOS* não é uma forte escolha para aplicações de tempo real.

O *TinyOS* não provê qualquer MAC específico, rede, ou implementações de protocolo de camada de transporte que suporte requisitos de qualidade de serviço de fluxos de multimídia em tempo real. Na camada MAC, *TinyOS* suporta TDMA, que pode ser otimizado, dependendo dos requisitos da aplicação para suportar fluxos de tráfego multimídia.

5.2.8 **Características adicionais**

Nesta seção, serão discutidas algumas características adicionais providas pelo *TinyOS*.

5.2.8.1 Sistema de arquivos

O *TinyOS* provê um sistema de arquivos de nível único. A lógica por trás, fornecendo um arquivo de sistema de arquivos de nível único, é o pressuposto de que apenas uma única aplicação é executada no nó em qualquer ponto em tempo. Como a memória do nó é escassa, ter um sistema de arquivos de nível único é, portanto, suficiente.

5.2.8.2 Suporte a banco de dados

O propósito de redes sensores é sensoriar, executar computações, armazenar e transmitir dados, mas, além disso, o *TinyOS* provê suporte a banco de dados na forma do *TinyDB*.

5.2.8.3 Suporte a segurança

A segurança de comunicação numa mídia de difusão sem fio é sempre requisitada. O *TinyOS* provê sua solução de segurança de comunicação na forma do *TinySec.*

5.2.8.4 Suporte a simulação

O *TinyOS* provê suporte a simulação na forma do *TOSSIM* [15]. O código de simulação é escrito em *nesC* e consequentemente também pode ser implantado para *motes* atuais.

5.2.8.5 Suporte a linguagem

O *TinyOS* suporta desenvolvimento de aplicação na linguagem de programação *NesC*. O *NesC* é um dialeto da linguagem C.

5.2.8.6 Plataformas suportadas

O *TinyOS* suporta as seguintes plataformas de sensoriamento: *Mica, Mica2, Micaz, Telos, Tmote* e alguns outros.

5.2.8.7 Suporte a documentação

O *TinyOS* é um SO bem documentado e documentação extensa pode ser encontrada na página do *TinyOS* em: http://www.tinyos.net .

5.3 CONTIKI

Contiki, é um SO leve, de código aberto, escrito em C, para nós sensores de RSSF. *Contiki* é um SO altamente portável, e ele é construído em torno de um *kernel* orientado a eventos. *Contiki* provê preempção multitarefa, que pode ser usada num nível de processos individual. Em uma configuração *Contiki* típica, são consumidos 2 kb de RAM e 40 kb de ROM. Uma instalação *Contiki* completa inclui características como: *kernel* multitarefa, *multithread* preemptivo, *protothreads*, rede TCP/IP, IPv6, uma interface de usuário gráfica, um navegador web, um servidor web pessoal, um cliente de telnet simples, uma proteção de tela, e computação de rede virtual.

5.3.1 **Arquitetura**

O SO *Contiki* segue uma arquitetura modular. No nível do *kernel* ele segue o modelo orientado à eventos, mas provê facilidades de *threading* opcionais a processos individuais. O *kernel* do *Contiki* é composto de um escalonador de eventos simples, que direciona os eventos para os processos em execução. A execução de processos é disparada por eventos direcionados pelo *kernel* aos processos, ou por um mecanismo de *polling*. O mecanismo de *polling* é utilizado para evitar condições de concorrência. Qualquer evento agendado vai executar até a conclusão, entretanto, manipuladores de eventos podem usar mecanismos internos para preempção.

Dois tipos de eventos EW são suportados pelo SO *Contiki*: eventos assíncronos e eventos síncronos. A diferença entre os dois é que eventos síncronos são direcionados imediatamente para o processo alvo, fazendo com que ele seja agendado. Por outro lado, eventos assíncronos são mais parecidos com chamadas

de procedimentos adiadas, que são enfileiradas e enviadas posteriormente para o processo de destino.

O mecanismo de *polling* usado no *Contiki* pode ser visto como eventos de alta prioridade que são agendados entre cada evento assíncrono. Quando um voto é agendado, todos os processos que implementam um manipulador de votos são chamados em ordem de sua prioridade.

Todas as facilidades do SO, por exemplo, manipulação de dados, comunicação, e drivers de dispositivos são providos na forma de serviços. Cada serviço tem sua interface e implementação. Aplicações usando um serviço particular precisam conhecer a interface de serviço. Uma aplicação não está preocupado com a implementação de um serviço.

5.3.2 **Modelo de programação**

Contiki suporta *multithreading* preemptiva. *Multithreading* é implementada como uma biblioteca no topo do *kernel* orientado a eventos. A biblioteca pode ser linkada com aplicações que requerem *multithreading*. A biblioteca de *multithreading* do *Contiki* é dividida em duas partes: uma parte independente da plataforma e uma parte específica da plataforma. A parte independente da plataforma faz interface com o *kernel* de eventos, e a parte específica de plataforma da biblioteca implementa comutação de pilha e primitivas de preempção. Como a preempção é suportada, a preempção é implementada usando a interrupção do timer e o estado da thread é guardado na pilha.

Para *multithreading*, o *Contiki* usa *protothreads* [18]. *Protothreads* são desenvolvidas para dispositivos limitados de memória variados porque eles não têm

pilha e são leves. As características principais dos *protothreads* são: custo adicional de memória muito pequeno (apenas dois bytes por *protothread*), não tem pilha adicional para uma thread, altamente portável (isto é, eles são totalmente escritos em C, e por isso não há código montado para arquitetura específica. Desde que os eventos executam até concluírem e o *Contiki* não permite que manipuladores de interrupção postem novos eventos, a sincronização de processos não é provida no *Contiki.*

5.3.3 **Escalonamento**

Contiki é um SO orientado a eventos, portanto ele não emprega nenhum algoritmo de escalonamento sofisticado. Eventos são disparados para a aplicação de destino assim que eles chegam. No caso de interrupções, manipuladores de interrupção de uma aplicação executam de acordo com sua prioridade.

5.3.4 **Gerenciamento e proteção de memória**

Contiki suporta gerenciamento dinâmico de memória. Além disso, ele também suporta ligação dinâmica com os programas. A fim de se proteger contra a fragmentação da memória, *Contiki* usa um alocador de memória gerenciado. A responsabilidade primária do alocador de memória gerenciado é manter a memória alocada livre de fragmentação, compactando a memória quando os blocos estão livres. Portanto, um programa usando um módulo alocador de memória não pode ter certeza que a memória alocada está no lugar.

Para o gerenciamento dinâmico da memória, *Contiki* também provê funções de gerenciamento de blocos de memória. Esta biblioteca provê simples mas poderosas funções de gerenciamento de memória para os blocos de tamanho fixo.

Um bloco de memória é declarado estaticamente usando a macro *MEMB ().* Blocos de memória são alocados da memória declarada pela função *memb_alloc (),* e são desalocadas usando a função *memb_free ().*

É importante notar aqui que *Contiki* não fornece qualquer mecanismo de proteção de memória entre diferentes aplicações.

5.3.5 **Suporte a protocolos de comunicação**

Contiki suporta um conjunto rico de protocolos de comunicação. No *Contiki*, uma aplicação pode usar qualquer versão de IP, isto é, IPv4 ou IPv6. *Contiki* provê uma implementação do *uIP*, uma pilha de protocolos para micro controladores de 8 bits pequenos. O *uIP* não requer que seus pares tenham uma pilha de protocolos completa, mas ele pode se comunicar com os pares executando uma pilha leve similar. A implementação *uIP* tem o mínimo conjunto de características necessárias para uma pilha TCP/IP completa. O *uIP* foi escrito em C, e ele pode suporta apenas uma interface de rede, suportando protocolos TCP, UDP, ICMP e IP.

Contiki provê outra pilha de protocolos em camadas leve, chamada *Rime*, para comunicação baseada em rede. *Rime* provê *unicast* de único salto, broadcast em único salto, e suporte a comunicação de múltiplos saltos. *Rime* suporta tanto *best-effort,* quanto transmissão confiável. Na comunicação de múltiplo salto, *Rime* permite que as aplicações executem seus próprios protocolos de roteamento. Aplicações são permitidas a implementar protocolos que não estão presentes na pilha *Rime*.

Contiki não suporta *multicast*. Portanto, *Contiki* não provê qualquer implementação de protocolos de gerenciamentos de grupo, como o *Internet Group Management Protocol* (ICMP), ou protocolo *Multicast Listener Discovery* (MLD).

Como a memória é um recurso escasso em dispositivos embarcados, *uIP* usa a memória de forma eficiente, usando mecanismos de gerenciamento de memória. A pilha *uIP* não usa alocação de memória dinâmica explícita, ela usa um *buffer* global para armazenar os pacotes de dados que chegam. Sempre que um pacote é recebido, *Contiki* o coloca no *buffer* global e notifica à pilha TCP/IP. Se é um pacote de dados, o TCP/IP notifica a aplicação apropriada. A aplicação precisa copiar os dados no *buffer* secundário ou ela pode imediatamente processar os dados. Uma vez que a aplicação está pronta com os dados recebidos, *Contiki* sobrescreve o *buffer* global com novos dados que chegam. Se uma aplicação atrasa o processamento de dados, então os dados podem ser sobrescritos por novos pacotes de dados que chegam.

Contiki provê uma implementação do *RPL (IPv6 routing protocol for low power and lossy networks)* pelo nome *ContikiRPL*. *ContikiRPL* opera através de enlaces sem fio de baixa potência e enlaces de linhas de energia com perdas.

5.3.6 **Compartilhamento de recursos**

Desde que os eventos que executam até a conclusão e o *Contiki* não permitem que manipuladores de interrupção postem novos eventos, *Contiki* provê acesso serializado a todos os recursos.

5.3.7 **Suporte a aplicações de tempo real**

Contiki não provê qualquer suporte para aplicações de tempo real, por isso, não há implementação de qualquer algoritmo de escalonamento de processos em tempo real no *Contiki*. No lado da pilha de protocolos de rede, *Contiki* não provê qualquer protocolo que considere os requisitos de QoS das aplicações de multimídia. Além disso, desde que *Contiki* provê uma implementação da micro pilha IP, interações entre camadas diferentes da pilha de protocolos não são possíveis.

5.3.8 **Características adicionais**

Nesta seção, serão discutidas características adicionais providas pelo SO Contiki.

5.3.8.1 Sistema de Arquivos Coffee

Contiki provê suporte a sistema de arquivos para dispositivos sensores baseados em flash na forma do sistema de arquivos *Coffee*. O propósito do sistema de arquivos *Coffee* é prover uma interface de programação para construir abstrações de armazenamento eficientes e portáveis. *Coffee* provê uma abstração de armazenamento independente de plataforma através de uma interface de programação expressiva. *Coffee* usa um consumo pequeno e constante por arquivo, tornando-o escalável. Na configuração padrão, *Coffee* requer 5 kb de ROM para o código e 0,5 kb de RAM em tempo de execução. Uma estrutura de página sequencial simples está sendo utilizada. *Coffee* também introduz o conceito de *micro logs* para manipular modificações de arquivo sem usar uma estrutura de log abrangente. Por causa da estrutura da página contígua de metadados dos arquivos, o *Coffee* usa uma consumo pequeno e constante para cada arquivo. A memória

flash é dividida em páginas lógicas, e o tamanho da página geralmente coincide com as páginas de memória *flash* subjacentes. Se o tamanho do arquivo não é conhecido de antemão, *Coffee* aloca um tamanho predefinido de páginas para o arquivo. Posteriormente, se o tamanho reservado torna a ser insuficiente, *Coffee* cria um novo arquivo maior e copia os dados do arquivo antigo dentro dele. Para aumentar o desempenho do sistema de arquivos, por padrão, *Coffee* usa um cache de metadados de 8 entradas na memória RAM. *Coffee* também provê uma implementação de um coletor de lixo que recupera páginas obsoletas quando um pedido de reserva do arquivo não pode ser satisfeito. Para alocar páginas para um arquivo, *Coffee* usa um algoritmo de primeiro ajuste. As memórias *flash* sofrem de desgaste, isto é, cada vez que uma página é apagada ela aumenta as chances de corrupção de memória. *Coffee* utiliza nivelamento de desgaste e o seu objetivo é espalhar rasuras de setor uniformemente para minimizar o risco de danificar alguns setores muito mais cedo do que outros. *Coffee* provê as seguintes APIs para os programadores de aplicações: *Open(), read(), modify(), seek(), append(), close().* A descrição detalhada destas APIs pode ser encontrada na documentação do *Contiki.*

5.3.8.2 Suporte a Segurança

Contiki não provê suporte para comunicação segura. Uma proposta de implementação de um protocolo de comunicação segura com o nome *ContikiSec* foi provida em [25].

5.3.8.3 Suporte a simulação

Contiki provê simulações de redes de sensores através do *Cooja.*

5.3.8.4 Suporte a Linguagem

Contiki suporta desenvolvimento de aplicações na linguagem C.

5.3.8.5 Plataformas suportadas

Contiki suporta as seguintes plataformas de sensoriamento: *Tmote, AVR series MCU.*

5.3.8.6 Suporte a documentação

Documentação do *Contiki* pode ser encontrada na *home page* do *Contiki* em: http://www.sics.se/contiki .

5.4 MANTIS

O sistema multimodal para redes de sensores sem fio (*MANTIS*) provê um novo sistema operacional *multithreaded* para RSSFs. *MANTIS* é um sistema operacional leve e eficiente em termos de energia. Ele tem um consumo de 500 bytes, que inclui *kernel*, escalonador, e pilha de rede. A principal característica do sistema operacional *MANTIS* (MOS), é que ele é portável através de múltiplas plataformas, isto é, pode-se testar aplicações MOS em um PDA, ou num PC. Depois, a aplicação pode ser portada para o nó sensor. MOS também suporta gerenciamento remoto dos nós sensores através de programação dinâmica. MOS é escrito em C e ele suporta desenvolvimento de aplicações em C. As subseções seguintes discutem as características de projeto do MOS em maior detalhe.

5.4.1 **Arquitetura**

MOS segue um projeto arquitetural em camadas. Em uma arquitetura em camadas, serviços providos por um OS são implementados também em camadas. Cada camada atua como uma máquina virtual avançada para as camadas acima. A seguir, são os serviços diferentes implementados por cada camada.

- Camada 3: Pilha de Rede, servidor de comandos, e *threads* de nível de usuário;
- Camada 2: API do sistema *MANTIS;*
- Camada 1: *Kernel*/escalonador, camada de comunicação (MAC e PHY) e drivers de dispositivo;
- Camada 0: Hardware;

O *kernel* do MOS apenas manipula a interrupção de *timer*, e todas as outras interrupções são diretamente enviadas para os *drivers* de dispositivos associados. Quando um *driver* de dispositivo recebe uma interrupção, ele posta um semáforo a fim de ativar uma *thread* em espera, e esta *thread* manipula o evento que causou a interrupção.

5.4.2 **Modelo de programação**

MOS suporta multitarefa preemptiva. A equipe do MOS projetou um SO *multithreaded* por causa dos fatos apresentados em [18], como por exemplo "Um sistema orientado a *threads* pode alcançar o alto desempenho de aplicações de concorrência intensa, com modificações apropriadas para o *threading package*." A memória do nó sensor é um recurso escasso, portanto, o MOS mantém duas seções

logicamente distintas de RAM: o espaço para variáveis globais que estão alocadas em tempo de compilação, enquanto o resto da RAM é gerenciada como um *heap*. Sempre quando uma *thread* é criada, espaço na pilha é alocado pelo *kernel* pelo *heap*. O espaço da pilha é retornado ao *heap* uma vez que a *thread* sai. A tabela de *thread* é a principal estrutura de dados que está sendo gerenciada pelo *kernel* MOS. Na tabela de *threads*, existe uma entrada por *thread*. O MOS aloca memória estaticamente para a tabela de *thread*, portanto, só poderá ser corrigido o número máximo de *threads.* Por outro lado, o custo adicional da tabela de *thread* é fixo.

O número máximo de *threads* pode ser ajustado no tempo de compilação, que por padrão é 12. A entrada na tabela de *thread* compreende 10 bytes e contém: ponteiro da pilha atual, informações de limite de pilha (ponteiro base e tamanho), ponteiro para a função inicial da *thread*, nível de prioridade da *thread* e ponteiro para próxima *thread*. Uma vez que a *thread* é suspensa, seu contexto é salvo na pilha. A partir que cada entrada na tabela de *thread* compreende 10 bytes e por padrão 12 *threads* podem ser criadas, o *overhead* associado em termos de memória é de 120 bytes. Por default, cada *thread* recebe uma fatia de tempo de 10 ms e uma mudança de contexto acontece com a ajuda de interrupções do temporizador. Chamadas de sistema e postagem de uma operação de semáforo também podem desencadear um mudança de contexto.

Suporte a *multithreading* no MOS vem com o custo de troca de contexto e *overhead* de memória da pilha. Em [18], o argumento apresentado em favor do *overhead* da troca de contexto, é que ele é apenas uma questão moderada em RSSFs. Foi observado que, cada troca de contexto incorre em *overhead* de 60 microssegundos. Em comparação à isso, a fatia de tempo *default* é muito maior, isto

é, 10 ms, então o *overhead* de troca de contexto é menor que 1%. Um segundo custo é a alocação da memória da pilha. A pilha de *thread* padrão no MOS é 128 bytes e *motes MICA2* tem 4 kb de RAM. Desde que o *kernel* do MOS ocupe apenas 500 bytes, um espaço considerável está disponível para suportar *multithreading*.

O MOS evita condições de concorrência, usando exclusões mútuas binárias e contando semáforos. Um semáforo no MOS é uma estrutura de 5 bytes, e isto é declarado por uma aplicação como necessário. A estrutura do semáforo contém um bloqueio ou contagem de byte, junto com ponteiros de início e fim.

5.4.3 **Escalonamento**

O MOS usa escalonamento preemptivo baseado em prioridades. MOS usa um escalonador semelhante ao *UNIX* com classes múltiplas de prioridade e usa a abordagem *round robin* dentro de cada classe de prioridade. O tamanho da fatia de tempo é configurável, por *default* ela é setada para 10 milissegundos. O escalonador usa uma interrupção de temporizador para trocas de contexto. Trocas de contexto são também disparadas por chamadas de sistema e operações de semáforo. A eficiência em energia é alcançada pelo escalonador do MOS, mudando o micro controlador para o modo de repouso quando as threads da aplicação estão ociosas.

A fila disponível do escalonador do MOS tem cinco prioridades, do nível alto até o baixo: *Kernel, sleep*, alta, normal, e ociosa. O escalonador agenda a tarefa de maior prioridade na fila disponível. A tarefa tanto executa até a conclusão ou é preemptada se sua fatia de tempo expira. Para fatiar o tempo, o escalonador do MOS usa um temporizador de 16 bits. Quando não existe *thread* na fila disponível, o sistema vai para o modo de repouso. Se o sistema está suspenso em I/O, o sistema entra no modo de repouso ocioso moderado. Se as *threads* de aplicação chamaram

a chamada de sistema *sleep* (), o sistema entra em um modo de repouso profundo, de economia de energia. Uma fila separada mantém a lista ordenada de *threads* que chamaram o *sleep* (), e é ordenada pelo tempo de repouso, de baixo até alto. A prioridade *sleep* na fila disponível permite *threads* recém despertadas, ter a mais alta prioridade, para que possam ser atendidas primeiro depois de acordar.

O *kernel* do MOS mantém pronta uma lista de ponteiros de início e fim para cada nível de prioridade. Existem cinco níveis de prioridade e estes ponteiros consomem 20 bytes no total. Estes dois ponteiros ajudam na rápida adição e deleção de *threads* de uma lista disponível, consequentemente aumenta o desempenho em manipular a lista de *threads*. O MOS também usa um ponteiro de *thread* atual de 2 bytes, um byte de estado de interrupção, e um byte de *flags*. O *overhead* total estático para escalonamento é 144 bytes.

O escalonador do MOS usa um escalonamento *round robin* dentro de cada classe de prioridade. Isto significa que *threads* de classes de maior prioridade podem causar que *threads* de menor prioridade fiquem em *starvation*. O MOS usa escalonamento prioritário, que pode suportar tarefas de tempo real melhor que os escalonadores do *TinyOS* ou *Contiki*. Mas ainda requer escalonadores em tempo real como *Rate Monotonic* e *Earliest Deadline First (EDF)*, a fim de verdadeiramente acomodar tarefas em tempo real.

5.4.4 **Proteção e gerenciamento de memória**

MANTIS permite gerenciamento dinâmico de memória, mas desencoraja seu uso, porque o uso de gerenciamento dinâmico de memória incorre em muito *overhead*. Em segundo lugar, a memória é um recurso escasso num nó sensor.

MANTIS gerencia a memória de diferentes *threads* usando a tabela de *threads* que já foi discutida. *MANTIS* não provê qualquer mecanismo para proteção de memória.

5.4.5 **Suporte a protocolos de comunicação**

MOS implementa a pilha de rede em duas partes. A primeira parte da pilha de protocolos de rede é implementada no espaço do usuário. Esta parte contém a implementação de protocolos da camada 3 (e acima). Uma segunda parte contém a implementação de operações de camada MAC e PHY. A lógica por trás da implementação de funcionalidades da camada 3 e superiores no espaço do usuário, é prover flexibilidade. Se uma aplicação quer usar seu próprio protocolo de roteamento dirigido à dados, então ele pode implementar seu protocolo de roteamento no espaço do usuário. O outro lado da abordagem é o desempenho, isto é, a pilha de protocolos de rede tem que usar APIs providas pelo *MANTIS* ao invés de se comunicar diretamente com um *driver* e *hardware* de dispositivo. Isto resulta em muitas trocas de contexto, resultando em *overheads* computacionais e de memória.

A segunda parte da pilha de protocolos é implementada numa camada *COMM*. A camada *COMM*, primeiramente implementa funcionalidades de sincronização e de camada MAC. A camada *COMM* provê uma interface unificada para comunicação com *drivers* de dispositivo, para interfaces como conexões seriais, USB e dispositivos de rádio. A camada *COMM* também faz *bufferização* de pacotes. É possível que os pacotes que chegam a partir da rede para uma *thread* que não está programada atualmente. Nestes cenários, a camada *COMM* vai *bufferizar* pacotes. Uma vez que a *thread* está agendada, a camada *COMM* passa um ponteiro para os dados na *thread* interessada.

O *SO MANTIS* não provê suporte para aplicações *multicast*, além disso não provê uma implementação para protocolos de gerenciamento de grupo. *MANTIS* também não provê suporte para aplicações multimídia de tempo real em sua pilha de protocolos de comunicação. Por outro lado, *MANTIS* provê uma facilidade para implementar roteamento customizado e protocolos de camada de transporte no topo da camada MAC. Assim, pode-se implementar protocolos de transporte e roteamento em tempo real para redes de sensores multimídia em *MANTIS*.

5.4.6 **Compartilhamento de recursos**

MANTIS realiza compartilhamento de recursos com a ajuda de semáforos. Ao mesmo tempo, ele não aborda o fenômeno da inversão de prioridades, onde um processo de maior prioridade espera em um processo de menor prioridade.

5.4.7 **Suporte para aplicações de tempo real**

MANTIS provê suporte muito pequeno para aplicações de tempo real no nível de processos. Foi discutido acima que o MOS usa escalonamento de prioridades dentro de cada classe de prioridade. Assim, processos executando tarefas de prioridade mais alta, podem ser mapeados para a classe de prioridade mais alta, e dentro dessa classe de processos, podem ainda ser atribuídos numa prioridade elevada. O MOS não fornece uma implementação de um algoritmo de escalonamento que pode cumprir prazos fáceis e difíceis de processos. Portanto, MOS não é um SO de tempo real para RSSFs. Para aplicações de multimídia em tempo real, funcionalidades adicionais são requeridas na pilha de protocolos de rede, com por exemplo, alocar largura de banda de rede para aplicações multimídia, achando rotas que satisfaçam requisitos de *QoS* dos fluxos, etc. MOS não provê qualquer funcionalidade em sua pilha de protocolos de rede.

5.4.8 **Características adicionais**

Nesta seção discute-se as características adicionais providas pelo SO *MANTIS*.

5.4.8.1 Suporte a simulação

MANTIS suporta simulação de rede de sensores sem fio através do *AVRORA*.

5.4.8.2 Suporte a linguagem

MANTIS suporta as seguintes plataformas de sensoriamento: *Mica2, MicaZ* e *Telos*.

5.4.8.3 Shell

Uma implementação do shell parecido com *UNIX*, vem com junto com o *MANTIS*, que roda no nó sensor.

5.4.8.4 Suporte a documentação

Documentação do *MANTIS* pode ser encontrada na *home page* do *MANTIS* em: http://mantisos.org.

5.5 NANO-RK

Nano-RK é um SO de tempo real multitarefa preemptivo fixo para RSSFs. O objetivo do projeto para o *Nano-RK* é suporte à multitarefa, suporte para rede de múltiplos saltos, suporte para escalonamento baseado em prioridades, pontualidade e escalonabilidade, vida útil das RSSFs prolongada, limites de uso de recursos do aplicativo e baixo consumo de energia. O *Nano-RK* usa 2 kb de RAM e 18 kb de ROM. *Nano-RK* provê suporte para CPU, sensores e reservas de largura de banda

da rede. *Nano-RK* suporta aplicações de tempo real leves e pesadas, pelos significados dos diferentes algoritmos de escalonamento em tempo real, como por exemplo, escalonamento de taxa monotônica e escalonamento de taxa harmonizada. Nano-RK provê suporte para rede, através da abstração do tipo *socket*. Nano-RK suporta plataformas de sensoriamento *FireFly* e *MicaZ*.

5.5.1 **Arquitetura**

O *Nano-RK* segue o modelo de arquitetura de kernel monolítico. Devido à sua natureza de tempo real, os autores do *Nano-RK* enfatizam o uso de *framework* de tempo de projeto estático, isto é, prioridades de tarefa, prazos, períodos, e suas reservas podem ser atríbuídas *off-line*, então estes procedimentos de controle de admissão podem ser aplicados eficientemente. Ao escolher esta abordagem estática, pode-se determinar se os prazos das tarefas podem ser cumpridos na concepção geral do sistema ou não. Programadores de aplicação podem mudar diferentes parâmetros (prazo, período, reserva de CPU, e reserva de largura de banda de rede) associados com as tarefas para chegar a uma configuração que atinja os objetivos globais. *Nano-RK* também provê APIs através das quais parâmetros de tarefa podem ser configurados em tempo real, mas seu uso é desencorajado, especialmente quando uma tarefa representa trabalhos pesados de tempo real.

5.5.2 **Modelo de programação**

Um dos objetivos para o *Nano-RK* foi facilitar os programadores de aplicação permitindo que eles trabalhem em um paradigma de multitarefa familiar. Estes resultados são obtidos em uma curva de aprendizado curta, tendo desenvolvimento rápido de aplicações e produtividade aumentada. Desde que o *Nano-RK* é um SO

multitarefa preemptivo, ele precisa salvar o contexto da tarefa corrente antes de agendar uma nova tarefa. Salvando o estado de cada tarefa resulta em consumo grande de memória de trocas de contexto frequentes resultando em uma performance reduzida e maior consumo de energia.

No *Nano-RK*, cada tarefa tem um bloco de controle de tarefas (TCB). É recomendável que o TCB possa ser inicializado durante a inicialização e criação da imagem do sistema. O TCB armazena os conteúdos do registro, prioridade, período de recorrência, (CPU, rede, sensores) tamanhos de reserva, e identificadores de porta da tarefa. Baseado no período da recorrência, o *Nano-RK* mantem duas listas ligadas aos ponteiros do TCB para ordenar o conjunto de tarefas ativas e suspensas.

Para prover semântica em tempo real, o *Nano-RK* provê multitarefa totalmente preemptiva, isto é, ele garante que o processo de maior prioridade na fila pronta sempre vai rodar no micro controlador.

Discutiu-se que cada tarefa tem um TCB associado, que contém o registro e conteúdo da pilha da tarefa, a prioridade da tarefa, a tarefa da CPU, rede, e reserva de sensores, o identificador da porta da tarefa, e seu período. Uma simples TCB requer uma quantidade significativa de memória, por isso, se houver um grande número de tarefas no sistema, o sistema pode rodar sem espaço de memória.

Cada SO *multithreaded* precisa fornecer suporte para primitivas de sincronização, para que estado correto dos dados compartilhados ou outros recursos possam ser mantidos. O *Nano-RK* fornece suporte a sincronização na forma de exclusões mútuas e semáforos.

5.5.3 **Escalonamento**

Nano-RK provê escalonamento por prioridades em dois níveis: escalonamento por prioridades no nível de processo e escalonamento por prioridades ao nível da rede. Nesta seção, discute-se apenas os algoritmos de escalonamento que são usados no *Nano-RK* para escalonamento de processos. Para suportar aplicações de tempo real, *Nano-RK* usa um algoritmo de escalonamento dirigido a prioridades totalmente preemptivo, isto é, a qualquer instância dada, a tarefa de maior prioridade é escalonada pelo sistema operacional. Um algoritmo de escalonamento por taxa monotônica é usado para tarefas periódicas de tempo real, e a prioridade da tarefa é configurada estaticamente baseada no período da tarefa: quanto menor período da tarefa maior é sua prioridade. Desde que o algoritmo de escalonamento de taxa monotônica atribui prioridades as tarefas, o *Nano-RK* recomenda configurar parâmetros de tarefa *off-line*.

Nano-RK também provê uma implementação do escalonamento de taxa harmonizada para economia de energia [18]. A idéia principal por trás desse algoritmo de escalonamento é eliminar períodos ociosos da CPU, agrupando a execução de tarefas diferentes. A partir do tempo de chegada inicial, periodicidade e prazos das tarefas são conhecidos *a priori*, escalonamento de taxa harmonizada pode ser usado para futuramente economizar energia eliminando ciclos ociosos.

O *Nano-RK* também fornece uma implementação do algoritmo de maior prioridade, para ligar o tempo bloqueado encontrado por um processo de prioridade maior, devido à inversão de prioridades, isto é, o recurso compartilhado necessário

pelo processo de prioridade mais elevado está a ser usado por um processo de prioridade mais baixa.

Para contornar a inversão de prioridade, cada exclusão mútua está associada com um teto de prioridade. Sempre que uma exclusão mútua é adquirida por uma tarefa, a prioridade da tarefa é definida igual ao limite máximo de prioridade da exclusão mútua. Uma vez, que a exclusão mútua é liberada, a prioridade da tarefa é definido igual a sua prioridade inicial.

5.5.4 **Proteção e gerenciamento de memória**

Nano-RK apenas provê suporte para gerenciamento estático de memória, ele não suporta gerenciamento dinâmico de memória. No *Nano-RK*, tanto o SO, quanto as aplicações, residem em um único espaço de endereços, e para o melhor conhecimento dos autores, *Nano-RK* não provê nenhum suporte para salvaguardar os SO co-localizados e espaços de endereços de processos.

5.5.5 **Suporte a protocolos de comunicação**

Nano-RK provê uma pilha de protocolos de rede leve, que provê uma abstração de comunicação similar a *sockets*. Como numa programação de rede tradicional, uma aplicação que quer enviar dados, pode criar um *socket*, e então pode começar a comunicar através deste *socket*. Da mesma forma, uma aplicação pode ligar e ouvir dois bytes particulares de um número de porta específico para receber dados. Para manipular a memória mais eficientemente, *buffers* de transmissão e recepção não são gerenciados pelo SO, em vez disso, eles são gerenciados pela aplicação. A razão apresentada para isso é que ele desperdiça memória se o SO reserva um espaço grande considerável para uma aplicação que

apenas envia e recebe poucos bytes de dados. Entretanto, é mais apropriado permitir que as aplicações gerenciem seus *buffers* de envio e recepção. O SO identifica os *buffers* de aplicação usando o número de porta presente no cabeçalho do pacote. O SO copia os dados recebidos dentro do *buffer* da aplicação usando semântica de cópia zero. Uma vez que os dados são colocados dentro do *buffer* da aplicação, a aplicação é notificada de acordo. Novos dados recebidos não são copiados dentro do *buffer* de aplicação até dados previamente colocados serem lidos pela aplicação ou se a aplicação explicitamente permite o SO a fazê-lo.

Um protocolo de enlace de tempo sincronizado para redes sem fio de múltiplos saltos de energia limitada (*RT-Link*) foi implementado no *Nano-RK*. O objetivo primário do *RT-Link* é prolongar a vida útil da rede de sensores e prover garantias no atraso fim a fim. *RT-Link* fornece suporte para aplicações em tempo real, por meio de atraso fim-a-fim delimitado através de múltiplos saltos e transmissão livre de colisão. *RT-Link* é implementado sobre um protocolo de camada de enlace TDMA, onde cada transmissão do nó ocorre em slots de tempo predefinidos, permitindo economias de energia. Um ciclo *RT-Link* consiste em 32 frames, e cada frame consiste em 32 slots de tempo. A duração de cada slot de tempo é igual a 5 ms, suficiente para transmitir um pacote de tamanho máximo. A duração de um único pacote é 5.12s. Existem dois tipos de slots no *RT-Link*: Pacotes agendados (SS) e Pacotes de contenção (CS). SS são alocados para aqueles nós membros que requerem atraso fim a fim limitado e a alocação dos slots e trazida pelo *gateway*, isto é, uma entidade de controle central. Cada nó membro manda sua lista de vizinhos para o *gateway* e o *gateway* atribui slots aos nós, dependendo da topologia de rede construída através da lista de vizinhos. Um novo nó entra na rede como nó convidado, e opera em slots de contenção. Um nó faz sua

requisição de reserva a um *gateway* usando CS. Uma vez que o SS é atribuído, o nó se torna um nó membro de opera durante o período de SS. Um nó móvel sempre opera no período de CS, porque sua adesão muda rapidamente com o tempo, perturbando o plano de agendamento definido pelo *gateway*.

A partir de que o *Nano-RK* é um SO baseado em reservas, ele provê APIs às aplicações para então que eles possam reservar largura de banda de rede de acordo com os requisitos da aplicação. O *Nano-RK* provê dois conjuntos de APIs: uma para a largura de banda reservada do lado emissor, e outra para a largura de banda reservada do lado receptor. Num dado período, uma aplicação pode apenas enviar ou receber dados de acordo com o status de reserva de largura de banda de sua rede. Esta restrição pode ser aliviada num sistema de tempo real leve, se houver alguma folga na largura de banda disponível no sistema. Em todo novo período, as reservas de largura de banda de cada aplicação são renovadas. *Nano-RK* não provê uma implementação de um algoritmo de roteamento *multicast* nem provê uma implementação de um protocolo de gerenciamento de grupo.

5.5.6 **Compartilhamento de recursos**

Para recursos compartilhados como memória, *Nano-RK* provê exclusões múltiplas e semáforos para acesso serializado. Para contornar o problema de inversão de prioridades, *Nano-RK* provê uma implementação do algoritmo de maior prioridade. Adicionalmente, *Nano-RK* provê APIs para reservar recursos do sistema como ciclos de CPU, sensores e largura de banda.

5.5.7 **Suporte para aplicações de tempo real**

Nano-RK é um sistema operacional de tempo real, por isso, ele provê suporte rico para aplicações de tempo real. Ele suporta processos de tempo real, e seu procedimento de controle de admissão *off-line*, garante alcançar os prazos associados com cada processo de tempo real admitido. *Nano-RK* provê uma implementação de algoritmos de escalonamento preemptivos de tempo real, e tarefas que são agendadas, usando um algoritmo de escalonamento de taxa monotônica. Além disso, *Nano-RK* provê reservas de largura de banda para fluxos sensíveis a atraso, e isto pretende prover garantias de atraso fim a fim numa rede de sensores sem fio de múltiplos saltos. *Nano-RK* é o SO adequado para uso em redes de sensores multimídia devido ao seu suporte extenso provido por aplicações de tempo real.

5.5.8 **Características adicionais**

Nesta seção discute-se características adicionais providas pelo SO *Nano-RK.*

5.5.8.1 Suporte à linguagem

Nano-RK suporta desenvolvimento de aplicações na linguagem C.

5.5.8.2 Plataformas suportadas

Nano-RK suporta as seguintes plataformas de sensoriamento: *MicaZ* e *FireFly*.

5.5.8.3 Suporte a documentação

A documentação *Nano-RK* pode ser encontrada na home page do *Nano-RK* em: http://www.nano-rk.org.

5.6 LITEOS

LiteOS é um sistema operacional do tipo UNIX projetado para RSSFs na Universidade de *Illinois* em *Urbana-Champaign*. As motivações por trás do projeto de um novo SO para RSSF são: prover um SO do tipo unix para RSSF, prover aos programadores de sistema um paradigma de programação familiar (modo de programação baseado em *threads*, embora ele ofereça suporte para registrar manipuladores de eventos usando callbacks), um sistema de arquivos hierárquico, suporte para programação orientada a objeto na forma do LiteC++, e um shell do tipo *UNIX*. O consumo do *LiteOS* é pequeno o suficiente para ser executado em nós *MicaZ* que possuem uma CPU de 8 MHz, 128 bytes de *flash* programável e 4 kb de RAM. *LiteOS* é composto primariamente por três componentes: *LiteShell*, *LiteFS* e o *kernel*. Nas subseções seguintes, discute-se estes três componentes e outras características do *LiteOS* em detalhes.

5.6.1 **Arquitetura**

LiteOS segue um projeto de arquitetura modular. *LiteOS* é particionado dentro de três subsistemas: *LiteShell*, *LiteFS* e o *kernel*. *LiteShell* é um shell do tipo *UNIX*, que provê suporte para comandos de shell para gerenciamento de arquivos, gerenciamento de processos, depuração e dispositivos. Um aspecto interessante do *LiteOS* é seu *LiteShell*, que reside numa estação base ou um PC. Isto influencia a suportar comandos mais complexos, porque a estação base tem recursos

abundantes. O *LiteShell* pode apenas ser usado quando há um usuário presente numa estação base. Algum processamento local é feito no comando do usuário (a análise de um comando) pelo shell e, em seguida, ele é transmitido sem fio para o nó sensor pretendido. O nó sensor requer processamento no comando e manda uma resposta de retorno, que é então mostrada ao usuário. Quando um *mote* não executa os comandos, um código de erro é retornado. Vale a pena notar que *Contiki* e *Mantis* também fornecem um shell para interagir com os motes, mas o código de implementação reside no mote, limitando a capacidade do shell devido a limitações de recursos. O segundo componente arquitetural do *LiteOS* é seu sistema de arquivos *LiteFS*. *LiteFS* monta toda a vizinhança de nós sensores como um arquivo. *LiteFS* monta uma rede de sensores como um diretório e então lista todos nós sensores de único salto como um arquivo. Usuário na estação base pode usar essa estrutura de diretório como a estrutura de diretório *UNIX* normal e um usuário também pode usar comandos legítimos nesses diretórios. Discute-se *LiteFS* em mais detalhes mais tarde. O terceiro maior componente do *LiteOS* é o kernel que reside no nó sensor. O kernel do *LiteOS* provê concorrência na forma de *multithreading*, provê suporte para carregamento dinâmico, usa escalonamento *round robin* e prioridade, permite aos programadores registrar manipuladores de eventos através de funções de *callback*, e provê suporte a sincronização.

5.6.2 **Modelo de programação**

LiteOS é um SO multitarefa que suporta *multithreading*. No *LiteOS*, processos executam aplicações como *threads* separadas. Cada *thread* tem seu próprio espaço de memória, para evitar erros potenciais de semântica que podem ocorrer durante leitura e escrita no espaço compartilhado de memória. *LiteOS* também provê suporte

para manipulação de eventos. Programadores da aplicação podem registrar manipuladores de eventos usando uma facilidade chamada *call-back,* provida pelo *LiteOS.*

Para evitar potenciais condições de concorrência, *LiteOS* provê funções *atomic_start ()* e *atomic_end ()*. Sempre que dados compartilhados entre *threads* diferentes são acessadas ou modificadas, é altamente recomendável usar estas funções. A documentação *LiteOS* não detalha como estas funções são implementadas internamente, isto é, se elas estão desabilitando interrupções ou usando exclusões mútuas.

5.6.3 **Escalonamento**

LiteOS provê uma implementação do escalonamento *round Robin,* e escalonamento baseado em prioridades. Sempre que uma tarefa é adicionada à fila disponível, a próxima tarefa a ser executada é escolhida através do escalonamento baseado em prioridades. As tarefas executam até a conclusão, ou até elas requisitarem um recurso que não está disponível atualmente. Quando uma tarefa requer um recurso que não está disponível, a tarefa habilita interrupções e vai para o modo de repouso. Uma vez que os recursos requisitados estejam disponíveis, a interrupção apropriada é sinalizada e a tarefa continua sua execução de onde parou. Quando uma tarefa completa sua operação ela deixa o *kernel*.

Quando não existem tarefas ativas no sistema, o nó sensor vai para o modo de repouso. Antes de ir para o modo de repouso, o nó habilita sua interrupção para que então ele possa acordar no tempo ou evento apropriado.

Como o escalonador do *LiteOS* permite que as tarefas sejam executadas até a conclusão, existe uma chance de que a tarefa de prioridade mais alta entre na fila disponível quando uma tarefa de baixa prioridade está completando sua execução. Neste cenário, a tarefa de maior prioridade pode errar seu prazo, portanto o *LiteOS* não é um SO apropriado para redes de sensores de tempo real.

5.6.4 **Proteção e gerenciamento da memória**

Dentro do *kernel*, o *LiteOS* suporta alocação de memória dinâmica através do uso de *malloc* e funções livres do tipo da linguagem C. Aplicações de usuário podem usar estas APIs para alocar e desalocar memória em tempo de execução. A memória dinâmica cresce na direção oposta da pilha *LiteOS*. A memória dinâmica é alocada da área inutilizada entre o final das variáveis do *kernel* e o começo do bloco de memória das aplicações do usuário. Isto permite ajustar o tamanho da memória dinâmica requisitada pela aplicação.

O *kernel* do *LiteOS* compila separadamente da aplicação, portanto o espaço de endereçamento não é compartilhado entre o *kernel* e a aplicação. Similarmente, cada aplicação de usuário tem seu espaço de endereço separado. A segurança da memória dos processos e do *kernel* é reforçada através de espaços de endereçamento separados.

5.6.5 **Suporte a protocolos de comunicação**

LiteOS provê suporte a comunicação na forma de arquivos. *LiteOS* cria um arquivo correspondendo a cada dispositivo no nó sensor. Similarmente, ele cria um arquivo correspondendo com a interface de rádio. Sempre que há algum dado que precise ser enviado, o dado é colocado dentro do arquivo de rádio e é

posteriormente transmitido via *wireless*. Da mesma forma, sempre que algum dado chega ao nó ele é colocado no arquivo de rádio e ele é entregue para a aplicação correspondente usando o número da porta presente nos dados.

Ao nível da camada de rede o *LiteOS* suporta encaminhamento geográfico. Cada nó contém uma tabela que pode apenas conter 12 entradas. Este protocolo de roteamento é suportado no *LiteOS* versão 0.3. Infelizmente, a documentação detalhada dos protocolos de comunicação suportados pelo *LiteOS* não está disponível, assim, não foi feita indicação sobre os protocolos suportados nas camadas MAC, de rede e de transporte.

5.6.6 **Compartilhamento de recursos**

LiteOS sugere o uso de APIs providas para sincronização sempre que uma *thread* quer recursos de acesso que são compartilhado por múltiplas threads. A documentação do LiteOS não provê qualquer detalhe, ou como recursos de sistema são compartilhados entre *threads* múltiplas em execução.

5.6.7 **Suporte para aplicações de tempo real**

LiteOS não provê qualquer implementação de protocolos de rede que suportam aplicações de multimídia de tempo real. Ele provê um algoritmo de escalonamento de processos baseados em prioridades, mas uma vez que é agendado, ele executa até a conclusão. Isto pode resultar num prazo errado de um processo de maior prioridade que entra na fila disponível uma vez que processo de baixa prioridade foi agendado.

5.6.8 **Características adicionais**

Nesta seção, discute-se características adicionais providas pelo *LiteOS.*

5.6.8.1 Sistema de arquivos LiteOS

LiteOS provê suporte para um sistema de arquivos hierárquico chamado *LiteFS*. *LiteFS* provê suporte para tanto arquivos e diretórios. *LiteFS* é particionado em três módulos. Ele mantém descritores de arquivos abertos, vetores de bit de alocação de memória e informação sobre o *layout* da memória *flash* na RAM. Um segundo módulo reside na EEPROM e este módulo contém informação sobre a estrutura hierárquica de diretórios. Em terceiro lugar, ele usa a *flash* para guardar dados. Como no sistema de arquivos *UNIX*, arquivos no *LiteOS* representam diferentes entidades, como dados, binários da aplicação e drivers de dispositivo.

A última versão do *LiteFS* suporta oito manipuladores de arquivo na RAM, e cada manipulador consome oito bytes. Entretanto, pelo menos oito arquivos podem ser abertos simultaneamente. Dois vetores de bit vetores são usados para manter o controle das alocações atuais em EEPROM e *flash*. O vetor de bit correspondente a EEPROM consome oito bytes e o vetor de bit correspondente a memória *flash* consome 32 bytes. No total, *LiteFS* requer 104 bytes de da RAM.

LiteFS monta todos os nós de único salto para o sistema de arquivos, assim como a montagem de um dispositivo USB. É importante notar que a partir que todos os vizinhos de único salto são montados num PC executando alguma versão do SO *Linux*, o usuário com acesso ao PC pode copiar ou deletar arquivos nestes nós. O usuário pode copiar um novo binário executável do PC para um nó em particular e

então ele pode pedir um comando executável para instruir o nó a iniciar a execução do arquivo.

5.6.8.2 Suporte a Simulação

LiteOS suporta simulação de redes de sensores sem fio através do *AVRORA.*

5.6.8.3 Suporte a linguagem

LiteOS suporta desenvolvimento de aplicações na linguagem LiteC++.

5.6.8.4 Plataformas suportadas

LiteOS suporta as seguintes plataformas de sensoriamento: *MicaZ* e *AVR series MCU.*

5.6.8.5 Suporte a documentação

Documentação *LiteOS* pode ser encontrada na *home page* do *LiteOS* em: http://www.liteos.net .

5.7 SISTEMAS NÃO-ESPECÍFICOS PARA RSSF

5.7.1 **uC/OS-II**

uC/OS-II é um kernel de SO embarcado multitarefa preemptivo de tempo real, que é popularmente portável, escalável e de fácil uso. O *uC/OS-II* provê um número de funcionalidades principais necessárias para aplicações embarcadas de rede, como multitarefa, sincronização, gerenciamento de *timer*, gerenciamento de memória. A pilha de protocolos de rede do *uC/OS-II* é composta por camadas MAC e de rede.

A camada MAC do *uC / OS-II* tem um papel importante controlando o meio de acesso para evitar colisões.

A camada de rede do *uC / OS-II* suporta uma variedade de algoritmos de roteamento em redes de sensores sem fio. *O uC / OS-II* também permite aos nós microssensores, nativamente intercalar tarefas complexas, com tarefas sensíveis ao tempo, assim mitigando o problema do *buffer* limitado do produtor/consumidor. Adicionalmente, a segurança e a confiabilidade é útil para construir redes de sensores sem fio robustas.

5.7.2 **LIMOS**

LIMOS (*Lightweight Multithreading Operational System*) é um sistema operacional híbrido nativamente configurável, que pode operar tanto no modo orientado a eventos ou no modo *multithreading* para minimizar requisições de recursos e aumentar a eficiência do sistema de acordo com ambientes de aplicações práticas. *LIMOS* adota uma arquitetura de sistema multinível baseada em componentes: ação, *thread* e evento. Em *LIMOS*, todos os tipos de trocas de dados, não importando interações interiores (entre componentes, ou seja, evento e *thread*) ou interações exteriores (geralmente com periféricos externos, incluindo sensores, atuadores e uma variedade de dispositivos de interface, etc), foram implementados via espaço da tupla. Assim, não há interação entre os *threads* e eventos.

LIMOS é um sistema de *microkernel* de tempo real inteligente, consciente de recursos, de baixo consumo de energia, e distribuído. Ele adota a arquitetura de sistema em multinível baseada em componente ação/evento/*thread*. Como resultado da estrutura multinível, LIMOS adota uma política de escalonamento em dois níveis: escalonamento de alto nível de "prioridade a não preempção" para eventos, e

escalonamento de baixo nível "prioridade a preempção" para *threads*. Desde que o *LIMOS* está dedicado a aplicações embarcadas com limitações estritas de recursos, especialmente para nós de RSSF, ele consome pouquíssimos recursos, como por exemplo, memória, energia e CPU. *LIMOS* pode operar em diferentes modos de operação (dirigido à eventos, *multithreading*), tendo muito poucos requisitos de memória (<5 kbytes) comparados a muitos dos sistemas operacionais de tempo real.

5.7.3 **Nano-Qplus**

Nano-Qplus é um novo sistema operacional de redes de sensores *multithreaded*, leve, e de baixo consumo de energia, integrado com uma plataforma de *hardware* de única placa, de propósito geral, para permitir prototipagem flexível e rápida de RSSF. *Nano-Qplus* usa a noção de tarefa para descrever um pedaço do código que precisa ser executado.

Normalmente, existem várias atividades de tarefas ao mesmo tempo e um escalonador de tarefas decide a ordem de execução. *Nano-Qplus* opcionalmente provê uma variedade de escalonadores como um escalonador FIFO não-preemptivo simples, e um escalonador FIFO temporizado. *Nano-Qplus* contém o algoritmo de protocolo MAC baseado no IEEE 802.15.4, formulando uma formação de redes ponto a ponto de múltiplo salto.

5.7.4 **PAVENET OS**

PAVENET OS provê *multithreading* híbrida: *multithreading* preemptiva e *multithreading* cooperativa. Ambas as *multithreadings* são otimizadas para os dois tipos de tarefa em RSSFs, e os tipos são: tarefas de tempo real, e tarefas de melhor esforço. *PAVENET OS* pode eficientemente realizar tarefas pesadas de tempo real,

que não podem ser feitas pelo *TinyOS*. Para realizar a característica pesada de tempo real, *PAVENET OS* é projetado com um modelo de *thread*, habilitando preempção. A preempção habilitada causa dois problemas. Primeiro, a preempção induz um alto *overhead* por checar prioridades de tarefas, e salvando o contexto da CPU. Segundo, a preempção induz um problema de conflito de gerenciamento entre tarefas. Para reduzir o *overhead* da preempção, *PAVENET OS* usa uma característica dos nós sensores sem fio: tarefas podem ser categorizadas como tarefas de tempo real ou tarefas *best-effort.*

PAVENET OS provê dois tipos de *multithreading*, que são *multithreading* preemptiva e *multithreading* cooperativa. A *multithreading* preemptiva é otimizada para tarefas de tempo real com um projeto de CPU específico, e *multithreading* cooperativa, é otimizada para tarefas *best-effort*. Para mitigar o problema do conflito de gerenciamento, *PAVENET OS* fornece uma pilha de comunicação sem fio para esconder os controles exclusivos para usuários. *PAVENET OS* sacrifica a portabilidade, porque o *PAVENET OS* tem um projeto específico para o *microchip* PIC18. A falta de portabilidade é um problema significante.

5.7.5 **SunSPOT**

Além da plataforma *SPOT* que já foi abordada anteriormente, o *SunSpot* também provê uma plataforma de software apropriada, para que se possa fazer comunicação com o sensor, bem como gerenciamento do mesmo. A seguir, nas seções que se seguem, falaremos sobre alguns aspectos sobre a plataforma de *Software* do *SunSpot*.

5.7.5.1 Instalação

A instalacão do *SunSPOT Manager* é extremamente simples, basta baixar o arquivo descritor do manager que o *Java Web Start* o instala automaticamente na máquina alvo. Entretanto, alguns pré-requisitos devem ser checados:

- Java SE 1.5 ou superior;
- Variável *JAVA HOME* setada corretamente (deve indicar o diretório da instalacão da JDK);
- Variável *PATH* deve possuir o diretório onde o executável java esta instalado;
- *Java Web Start* instalado.

Além da SDK básica, o sistema procura por instalações do *NetBeans*, versões superiores da 5.5, a fim de instalar o plug-in necessário para o desenvolvimento facilitado das aplicações para os *SPOTs*.

Maiores informações sobre a instalação estão disponíveis no arquivo *InstallationInstructions*.pdf, que é fornecido junto com o pacote.

5.7.5.2 Programas no *SunSPOT*

Enquanto numa aplicação Java normal, um programa é iniciado através do método estático main, numa aplicação Java ME a execução é definida através de uma classe que estende *MIDlet*. *MIDlet* é o padrão de programas Java para dispositivos embarcados, como por exemplo telefones celulares. Um *MIDlet* é

geralmente empacotado num *Java Archive (JAR*), para depois ser implantado nos *SPOTs*. Esse arquivo Jar é composto de:

- Arquivos *.class*: Arquivos com a implementação do programa conforme o padrão *MIDlet*;
- *Manifest file*: Arquivo que descreve o conteúdo do JAR;
- Recursos (imagens, dados, etc) relativos ao JAR.

Mais detalhes sobre o empacotamento dos *MIDlet* pode ser encontrado em [40].

A máquina virtual utilizada nos *SPOTs* é a Squawk a qual implementa a *Java ME*. Entretanto, há uma diferença fundamental entre uma *Java ME* e a *Squawk*.

Enquanto numa *Java ME* somente uma aplicação Java pode rodar ao mesmo tempo (independente do número de *threads*), na *Squawk* é permitido que mais de um programa rode junto. Este recurso é disponibilizado graças à classe de Isolação. Esta controla o acesso aos recursos concorrentes do *SPOT*, criando seções críticas e permitindo o acesso de somente um *MIDlet* a cada vez.

5.7.5.3 SPOT Emulator

O pacote da *SDK SunSPOT* disponibiliza um emulador de *SPOTs* [40], para o teste e desenvolvimento das aplicações no caso onde não se possui os aparelhos reais. Esse emulador fornece todas as características de um *SPOT* real, tais como, nível de luminosidade, temperatura, as cores dos LEDs, etc. Além disso, cada *SPOT* possui um identificador único que permite a troca de mensagens entre eles. A figura 13 mostra um exemplo do uso do emulador.

Para iniciar um *SPOT* virtual é suficiente adicioná-lo no *SPOTWorld*, então é possível associar um arquivo jar e iniciar sua execução. Dentre as opções de manipulação disponíveis, cita-se:

- Escolher um nome para identificar o *SPOT*;
- Reiniciar o *SPOT*;
- Mostrar a saída de texto da aplicação;
- Manipular os sensores (luminosidade, temperatura, pinos analógicos e digitais, aceleração);
- Requisitar Informações sobre o *SPOT* ou deletá-lo.

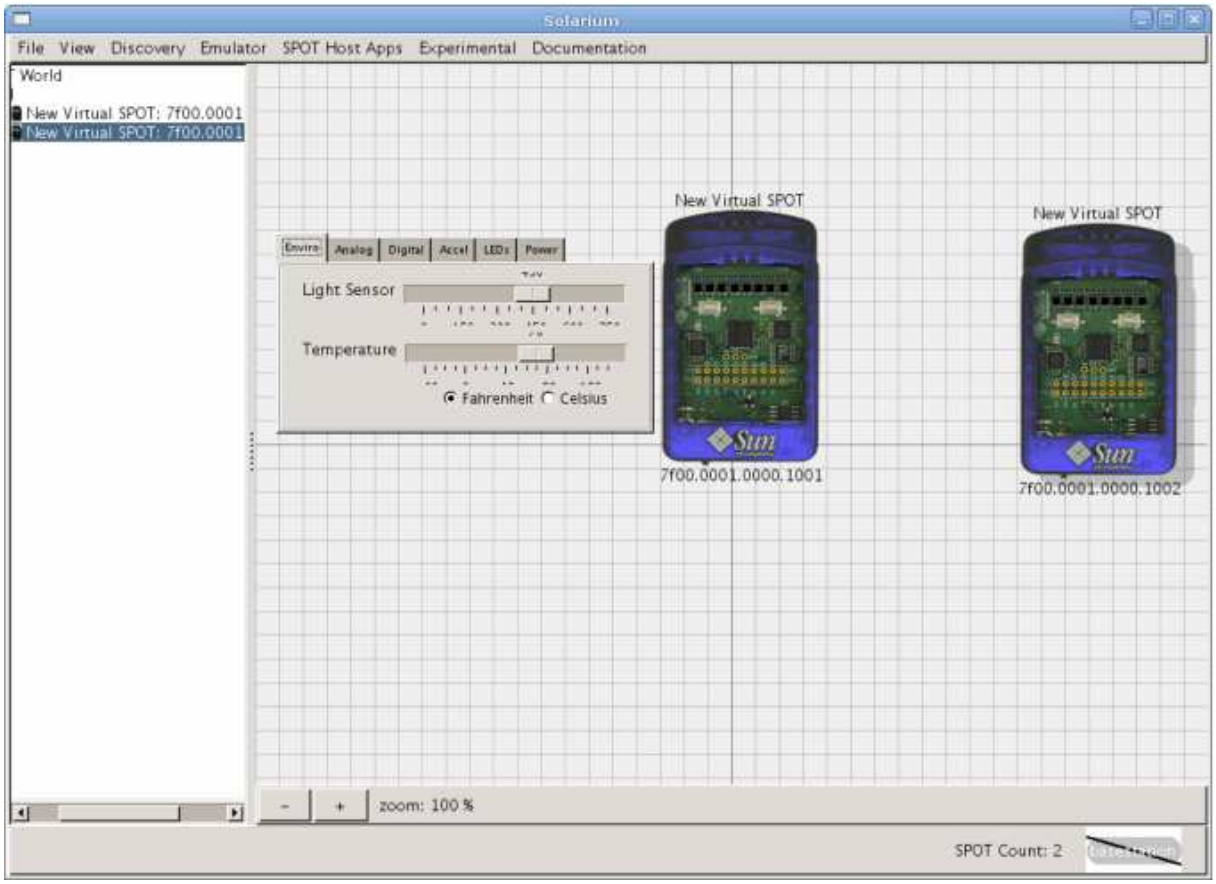

Figura 13 - Exemplo de utilização do ambiente virtual de simulação dos SPOTs

Um fator importante a salientar é o uso de memória pelo emulador. A medida que o número de *SPOTs* simulado cresce o uso de memória cresce linearmente, já que para cada aplicação rodada num *SPOT* ele cria uma nova máquina virtual *Squawk*. Cada VM criada usa aproximadamente 30Mb de memória RAM. Logo, podemos alcançar o limite de memória do computador usado, limitando o tamanho dos sistemas simulados.

5.7.6 **IBM Mote Runner**

IBM Mote Runner, é tanto uma especificação quanto uma implementação de ambiente em tempo de execução para sistemas pequenos embarcados, primariamente RSSF, que visa combater todos os problemas relacionados à estas mesmas redes. Por um lado, isso acontece diretamente, provendo um ambiente de execução unificada e virtualizada para configurações flexíveis. Por outro lado, isto é feito indiretamente, provendo a infraestrutura e mecanismos para a implementação ou soluções de alto nível ou de domínio específico da aplicação. Da perspectiva da engenharia, as especificações do *Mote Runner* são projetadas para ser implementadas de forma portável, escalável e eficiente, com base e levar adiante a nossa experiência com JCOP, a implementação *IBM JavaCard* [46].

A implementação *Mote Runner* baseia-se num hardware “fora-da-caixa” embarcado, com uma leve camada de abstração de *hardware* escrita em C e a*ssembler,* encapsulando qualquer funcionalidade específica de *hardware*. Na próxima camada, existe uma máquina virtual (também escrita em C), biblioteca em tempo de execução (escrita em C e C#) e camada MAC 802.15.4 (escrita em C). Tanto a biblioteca em tempo de execução e a camada MAC expõem APIs para desenvolvimento de aplicações em linguagens de alto nível como Java e C#. Pilhas

de protocolo de comunicação de rede de alto nível são implementadas no topo da biblioteca em tempo de execução e na camada MAC, por sua vez, expondo novas APIs para programadores de aplicação.

5.7.6.1 Portabilidade

No *Mote Runner*, a portabilidade geralmente trabalha em vários níveis. Em primeiro lugar, pela introdução de uma máquina virtual, as aplicações estão protegidas contra as particularidades do *hardware* de uma ampla gama de diferentes *motes*. As características específicas de *motes* individuais podem ainda ser disponibilizadas através de interfaces opcionais, enquanto a sua presença ou ausência pode ser listada em um perfil que descreve uma configuração do mote em um dado ponto no tempo. O desenvolvimento de aplicações, em seguida, torna-se independente de configurações de hardware de motes individuais, sendo mais ligada ao conjunto de recursos de um mote e, potencialmente, as suas características não-funcionais em vez disso.

Em segundo lugar, como as especificações do código de byte da máquina virtual, o formato de carga de aplicação, bem como a interface de programação da aplicação (API) será disponibilizada abertamente, aplicações desenvolvidas contra essas especificações podem ser executadas em diferentes implementações das especificações da plataforma de tempo de execução do *Mote Runner*. O desenvolvimento de aplicações então, torna-se independente de uma implementação específica do *Mote Runner*, que permite que os desenvolvedores não apenas escolham a implementação que fornece a melhor escolha, mas também mudem graciosamente para uma melhor escolha da implementação *Mote Runner* se

qualquer tipo de implementação se torna disponível ou requisitos da aplicação mudam com o tempo.

Em alguns casos específicos de implementação do *Mote Runner*, foca-se em um terceiro nível de portabilidade, nomeado como a habilidade de implementar a plataforma em tempo de execução em um caminho que pode se tornar disponível em alguns novos *hardwares* de *motes,* com um pequeno esforço possível, ainda sem sacrificar a eficiência. Isto é alcançado pela combinação de trocas de tempo de compilação para configurar as características tais como ordenação da plataforma ou *layout* de memória, e as decisões de *design*, como a introdução de uma pequena camada de abstração de *hardware*. Além disso, o código de mais alto nível é desenvolvido em C# e traduzido para o código de byte do *Mote Runner* para compactação e fácil manutenção.

5.7.6.2 Escalabilidade e Eficiência

Mesmo que *motes* das RSSF variem amplamente em termos de poder de processamento e quantidade de memória disponível, pequenos dispositivos com recursos escassos são predominantes. *Motes* poderosos geralmente servem apenas como *hubs* ou *gateways*, interligando *clusters* de *motes* menos poderosos ou integrando-os em uma infraestrutura maior. Assim, ambientes em tempo de execução para RSSF devem ser projetados a partir do zero para funcionar de forma eficiente em *motes* com muito poucos recursos, ou seja, micro controladores de 8 bits com apenas alguns kilobytes de memória transitória e dezenas de kilobytes de memória persistente, e ainda fazer bom uso dos recursos adicionais, se disponíveis. Em geral, isso tende a ser significativamente mais fácil de escalonar e fazer uso de mais recursos do que o contrário.

5.7.6.3 Linguagens de alto nível

De um ponto de vista linguagem de alto nível, a única exigência do código de byte do *Mote Runner* é a capacidade de mapear a partir de uma linguagem superior, com um comportamento padrão razoável e sem sacrificar a eficiência. Isto é, dentro destes limites, o *Mote Runner* é independente de linguagem. Isto é conseguido através da especificação de um código de byte digitado que é compacto e pode ser executado de forma eficiente em plataformas muito pequenas. Enquanto isso restringe o suporte para linguagens de alto nível, para aqueles que fornecem informações de tipo em tempo de compilação, evitando pesquisas de tempo de execução de informações de tipo, que são obrigatórias para executar eficientemente em micro controladores de 8 bits de baixo custo. Além disso, muitas linguagens como *Javascript*, que geralmente suportam informações de tipo dinâmico, hoje em dia, permitem que informações de tipo opcional a ser fornecido pelo programador.

5.7.6.4 Modelo de programação orientada a eventos

Sistemas embarcados em geral e, em particular, as RSSFs, operam reativas inerentemente, orientados à eventos como mudanças no ambiente capturado pelos sensores, as interrupções de tempo, ou mensagens recebidas de *motes* vizinhos. Em resposta, algum processamento é acionado, outros eventos podem ser gerados, e certos atuadores podem ser ativados. O *Mote Runner*, portanto, se baseia em um modelo de programação orientada a eventos, em vez de fornecer *threads*, que em primeiro lugar leva a uma forma mais natural de aplicações de estruturação e, em segundo lugar, é mais eficiente e amigável ao recurso. Naturalmente, como mencionado antes, as *threads* são então também deixadas de fora dos subconjuntos de linguagem do *Mote Runner*, definidos para Java e C #. Ao nível do C#, as APIs

do *Mote Runner* fazem uso extensivo de delegados para representar e chamar manipuladores de eventos. Em Java, os delegados são emulados por alguns padrões de codificação específicos, que podem ser identificados durante o tempo de compilação, e mapeados para o código de byte delegado correspondente da máquina virtual do *Mote Runner*.

5.7.6.5 Gerenciamento Remoto

Um dos principais objetivos do *Mote Runner* é fornecer uma infraestrutura que permite que as aplicações sejam dinamicamente distribuídas, carregadas e atualizadas, bem como deletadas de uma RSSF operante. A máquina virtual é um componente chave, uma vez que fornece uma base, que facilita a execução do mesmo código binário em todos os *motes* de uma RSSF heterogênea.

Em conjunto com um formato de arquivo de carga adequado para fluxo de ligação, novas aplicações também podem ser distribuídas em fluxo através da RSSF para economizar recursos. Para distribuir uma aplicação através de toda a RSSF, ela pode ser injetada via praticamente qualquer *mote* e de lá inundada para outros *motes*. Cada pacote recebido por um *mote* podem ser encaminhado para os *motes* vizinhos imediatamente, sem a necessidade de toda a aplicação ser temporariamente armazenada em formato de distribuição.

5.8 ANÁLISE COMPARATIVA

Nesta seção, sumariza-se nosso exame dos SOs para RSSF. A tabela 11 sumariza as discussões anteriores baseadas nas características comuns: Arquitetura, Modelo de Programação, Escalonamento, Gerenciamento e Proteção de

memória, Suporte a protocolos de comunicação, Compartilhamento de recursos, e Suporte para aplicações de tempo real.

Tabela 11 - Resumo dos sistemas operacionais para RSSF

OS/ Feature	Architecture	Programming model	Scheduling	Memory Management and Protection	Communication Protocol Support	Resource Sharing	Support for Real-time Applications
TinyOS	Monolithic	Primarily event Driven, support for TOS threads has been added	FIFO	Static Memory Management with memory protection	Active Message	Virtualization and Completion Events	No
Contiki	Modular	Protothreads and events	Events are fired as they occur. Interrupts execute w.r.t. priority	Dynamic memory management and linking. No process address space protection.	*u*IP and Rime	Serialized Access	No
MANTIS	Layered	Threads	Five priority classes and further priorities in each priority class.	Dynamic memory management supported but use is discouraged, no memory protection.	At Kernel Level COMM layer. Networking Layer is at user level. Application is free to use custom routing protocols.	Through Semaphores.	To some extent at process scheduling level (Implementation of priority scheduling within different processes types)
Nano-RK	Monolithic	Threads	Rate Monotonic and rate harmonized scheduling	Static Memory Management and No memory protection	Socket like abstraction for networking	Serialized access through mutexes and semaphores. Provide an implementation of Priority Ceiling Algorithm for priority inversion.	Yes
LiteOS	Modular	Threads and Events	Priority based Round Robin Scheduling	Dynamic memory management and it provides memory protection to processes.	File based communication	Through synchronization primitives	No

Da tabela acima pode ser visto que poucos SOs provêem suporte para aplicações de tempo real. Alguns SOs provêem suporte para escalonamento de prioridades enquanto muitos outros não provêem suporte para isto. Exceto o *Nano-RK*, nenhum dos SOs examinados provêem suporte para aplicações de tempo real na pilha de protocolos de comunicação. Também pode ser visto que SOs anteriores para RSSF enfatizaram o uso do paradigma de programação orientada à eventos. Entretanto, como os programadores são mais familiarizados com o paradigma de programação baseado em *threads*, os SOs contemporâneos para RSSF suportam o modelo de programação baseado em *threads*. A tabela 12, sumariza as diversas características dos SOs para RSSF examinados.

Tabela 12 - Resumo de recursos diversos dos Sistemas Operacionais para RSSF

OS/Feature	Communication Security	File System Support	Simulation Support	Programming Language	Shell
TinyOS	TinySec	Single level file system	TOSSIM	NesC	Not available
Contiki	ContikiSec	Coffee file system	Cooja	C	Unix-like shell runs on sensor mote
MANTIS	Not available	Not available	Through AVRORA	C	Unix-like shell runs on sensor mote
Nano-RK	Not available	Not available	Not available	C	Not available
LiteOS	Not available	LiteFS	Through AVRORA	LiteC++	Shell that runs on base station

Devido à semântica *multithreaded*, todo programa *MANTIS* precisa ter espaço da pilha alocada da pilha do sistema (*heap*), e mecanismos de bloqueio que devem ser usados para alcançar exclusões mútuas de variáveis compartilhadas. Em contraste, *Contiki* usa um escalonador baseado em eventos sem preempção, assim evitando alocação de múltiplas pilhas e mecanismos de bloqueio. *Multithreading* preemptiva é provida por uma biblioteca que pode ser linkada com programas que explicitamente requerem isto.

O *Kernel* de eventos do C*ontiki* é significantemente maior que o *kernel* do *TinyOS*, porque diferentes serviços são providos. Enquanto o *kernel* de eventos *TinyOS* provê apenas um escalonador *FIFO* de filas de eventos, o *kernel* do *Contiki* suporta tanto eventos *FIFO* e manipuladores de votação com prioridades.

Os resultados experimentais mostram que *MANTIS* é mais previsível do que o *TinyOS*. Especificamente, o pacote enviando tempo de execução de tarefa no *MANTIS* tem uma pequena variação e é independente de outras atividades como a tarefa de sensoriamento. Assim, *MANTIS* poderia ser preferível em situações que precisam processamento determinístico e temporizado. Entretanto, como uma mostra de experimentos, o sistema *MANTIS* não é eficiente em energia como o

TinyOS. Assim, o *TinyOS* poderia ser preferível se o consumo de energia é escolhido como sendo de primeira importância.

O mecanismo de carregamento dinâmico do *LiteOS* segue a linha de vários esforços prévios que não envolveram memória virtual como o *TinyOS* (usando XNP), *SOS* e *Contiki*.

A Latência do *TinyOS* é muito menor do que outras, porque a criação de tarefas do *TinyOS* simplesmente significa atribuir um ponteiro de função de uma tarefa a uma fila pronta. Para isto, não precisa que a memória seja alocada ou copiada porque o escalonador do *TinyOS* é *FIFO* (não preemptivo). Entretanto, sistemas operacionais *MANTIS* e *Nano-Qplus* requerem alocação de memória para o bloco de controle de tarefas.

Nano-RK suporta técnicas de gerenciamento de energia e provê várias APIs sensíveis a energia para uso do sistema. Enquanto sistemas operacionais de baixo consumo como o *uC/OS* suportam escalonamento em tempo real, eles não dão suporte para redes sem fio.

O *Nano-RK* é o trabalho relacionado mais próximo ao *PAVENET OS*. *Nano-RK* é um sistema operacional multitarefa preemptivo, que suporta tarefas de tempo real. Adicionalmente, *Nano-RK* é mais portável do que o *PAVENET OS*. Entretanto, o *Nano-RK* tem maior *overhead* de trocas de contexto do que *PAVENET OS*, porque o *Nano-RK* preserva a troca de contexto por *software*. *Nano-RK* precisa de várias dúzias de microssegundos para comutação de tarefas enquanto que o *PAVENET OS* precisa de vários microssegundos.

Como o *PAVENET OS*, *MANTIS* é um sistema operacional de modelo de *thread*. A diferença entre *MANTIS* e *PAVENET OS* é a implementação do modelo de *thread*. *MANTIS* usa *multithreading* de tempo fatiado enquanto *PAVENET OS* não é de tempo fatiado.

Tabela 13 - Tabela comparativa entre sistemas operacionais de RSSF

Features	TinyOS	SOS	MANTIS	Nano-Qplus	Nano-RK	Improved uC/OS-II
Low power mode	●	●		●	●	●
Multimodal sensing/tasking		●	●	●	●	●
Dynamic reprogramming	●	●				●
Priority-based scheduling			●	●		
Real-time guarantee			●	●	●	●
Execution model	Component based	Module based	Thread based	Thread based	Thread based	Thread based

6 **FERRAMENTAS DE DESENVOLVIMENTO**

Esta seção apresenta uma breve descrição das ferramentas de *software* para desenvolvimento de aplicações para RSSFs, incluindo uma descrição da linguagem de programação *NesC* e ferramentas de simulação *TOSSIM* e *TinyViz*.

Uma aplicação é descrita na linguagem *NesC*. Em seguida, o código fonte desta aplicação, junto com o *kernel* do *TinyOS* e as bibliotecas do *TinyOS* são compilados por um compilador *NesC*, resultando no código de uma aplicação com *TinyOS* em C. Este código é compilado por um compilador C, gerando um executável de uma aplicação.

Os tipos de arquivos em cada uma das etapas do desenvolvimento de aplicação possuem extensões diferentes, que servem para identificar o formato do arquivo. Arquivos escritos em *NesC* possuem extensão .nc. Quando compilados através do compilador da linguagem *NesC*, o *ncc*, geram arquivos em C que possuem extensão .c

Um compilador C (no caso o *avr-gcc*) gera código de máquina (extensão .s) que serve de entrada para um montador (no caso o *avr-as*). O montador irá gerar o código objeto (extensão .o). Por fim, o compilador *avr-gcc* é utilizado para gerar o código executável (extensão .exe). Este pode ser carregado no hardware dos *Mica Motes* no formato *S-Records* (arquivos com extensão .srec). A conversão do executável para o formato *S-Records* é feita pelo *avr-objcopy*.

Esclarecido o processo de geração de código, é preciso compreender os conceitos da linguagem para poder desenvolver aplicações. Dessa forma, o *NesC* é brevemente explicado a seguir.

6.1 NESC

A linguagem *NesC* [9] é uma extensão da linguagem de programação C projetada para incluir os conceitos estruturais e modelos de execução do *TinyOS*. Conforme mencionado anteriormente, o *TinyOS* é um sistema operacional dirigido a eventos, voltado para RSSF que possui recursos limitados (por exemplo, 8 Kbytes de memória de programa, 512 bytes de RAM). O manual da linguagem [24], assim como o código fonte, são abertos e podem ser encontrados no site: http://nescc.sourceforge.net/.

6.1.1 **Conceitos Básicos do NesC**

O *NesC* permite separar construção de composição. Aplicativos escritos em *NesC* são compostos por componentes, que podem ser construídos e combinados para formar uma aplicação, aumentando a modularidade e reusabilidade de código. Em *NesC*, o comportamento de um componente é especificado em termos de um conjunto de interfaces. Interfaces são bidirecionais, e informam o que um componente usa e o que ele provê.

Componentes são estaticamente ligados um ao outro via interfaces. O fluxo de informação pode ocorrer com camadas inferiores (via comandos) ou com camadas superiores (via eventos). Conceitos do *TinyOS*, como eventos, tarefas e comandos, estão embutidos no *NesC*. Relembrando os pontos importantes destes conceitos:

a) Tarefas

- Realizam computação (conjunto de comandos);

- Não são críticas com relação a tempo;

b) Eventos

- Críticas em relação a tempo;
- Sinalizam interrupções externas;
- Geram um “*Signal*”;
- Receptor recebe/aceita um “*Event*”;

c) Comandos

- Funções de procedimentos para outros componentes;
- Não podem sinalizar eventos;

O *TinyOS* provê outras ferramentas para facilitar o desenvolvimento de aplicações, como simuladores e depuradores. A seguir são descritos os simuladores do *TinyOS*.

6.2 O SIMULADOR TOSSIM

O *TOSSIM* (*The TinyOS simulator*) [9] é um simulador discreto de eventos para RSSFs que usam o *TinyOS*. Ao invés de compilar a aplicação do *TinyOS* para o *mote*, usuários podem compila-lá para o ambiente do *TOSSIM*, que executa em um PC. O *TOSSIM* permite que usuários depurem, testem, e analisem algoritmos em ambientes controlados. Como o *TOSSIM* executa em um PC, usuários podem examinar seus códigos *TinyOS*, utilizando depuradores e outras ferramentas de desenvolvimento utilizadas em programas para PC.

O objetivo primário do *TOSSIM* é prover uma simulação com alta fidelidade das aplicações para o *TinyOS*. Por esta razão, *TOSSIM* prefere focar na simulação do *TinyOS* e na execução deste a simular o mundo real. Enquanto o *TOSSIM* pode ser usado para compreender as causas de comportamentos observados no mundo real, ele não captura todos estes, e por isso não deve ser usado para avaliações absolutas. O código que é executado no *TOSSIM* é compilado diretamente do código *TinyOS*. Este código pode ser executado nativamente em um desktop ou laptop. *TOSSIM* permite simular milhares de nós sensores simultaneamente. Na simulação, cada nó sensor executa o mesmo programa *TinyOS*.

6.3 O AMBIENTE TINYVIZ

TinyViz é um programa em Java com interface gráfica que permite visualizar e controlar a simulação enquanto ela é executada, inspecionando mensagens de depuração, rádio e pacotes *UART* (*Universal Asynchronous Receiver Transmitter*).

TinyViz provê vários mecanismos de interação com a rede. O *TinyViz* provê suporte à monitoração de tráfego de pacotes e injeção de pacotes na rede de forma dinâmica.

6.4 DESENVOLVENDO UMA APLICAÇÃO

Nesta seção é desenvolvida uma aplicação simples que utiliza os conceitos, componentes e ferramentas descritos nas seções anteriores. O *NesC* provê sintaxe para o modelo do *TinyOS*, criando comandos, eventos e tarefas. O *NesC* utiliza o conceito de Interfaces para aumentar o reuso. Interfaces especificam a funcionalidade de um componente para o mundo exterior, elas identificam quais comandos podem ser chamados e quais eventos precisam ser tratados.

Convencionalmente, arquivos de interface são nomeados “*.nc”. Os componentes de software são formados por módulos e configurações. Os arquivos “*M.nc” possuem a e a definição de interface dos módulos. Os arquivos de configuração são nomeados “*C.nc”, e conectam componentes (módulos e configurações) em unidades. Opcionalmente, os arquivos de configuração especificam as interfaces que usam e provem. O mais importante é que eles não possuem código C. Quando representarem a aplicação no nível mais alto, podem perder a letra C do nome do arquivo.

Um exemplo simples de programa que faz um LED do *Mote* piscar e que é executado com o *TinyOS* será explicada nesta seção.

Trata-se da aplicação chamada “*Blink*” que pode ser encontrada no caminho “*apps/Blink*” da árvore do *TinyOS*. Esta aplicação causa o LED vermelho do mote acender e apagar em uma freqüência de 1 Hz.

A aplicação *Blink* é composta de dois componentes: um módulo denominado “*BlinkM.nc*”, e uma configuração chamada “*Blink.nc*”. É importante ressaltar que todas as aplicações requerem um arquivo de configuração no nível mais alto, que tipicamente possui o nome da própria aplicação. Neste caso, o arquivo “*Blink.nc*” é a configuração para a aplicação *Blink*, que é utilizada pelo compilador *NesC* para gerar o arquivo executável. Este arquivo também conecta o módulo *BlinkM.nc* aos outros componentes que aplicação *Blink* utiliza. “*BlinkM.nc*” provê a implementação da aplicação.

A distinção entre os módulos e configurações, permite a um desenvolvedor de sistema, o desenvolvimento de aplicações através da conexão de módulos, o que permite uma programação ágil. Por exemplo, um desenvolvedor pode prover uma

configuração que simplesmente conecta um módulo a outros módulos, nenhum dos quais ele desenvolveu. Da mesma forma, outro desenvolvedor pode prover um novo conjunto de bibliotecas de módulos que podem ser usados em diversas aplicações.

Em alguns casos (como é no caso do *Blink* e *BlinkM*), uma configuração e um módulo podem ser unidos. Quando este é o caso, a convenção usada na árvore fonte do *TinyOS* é que "*Foo.nc*" representa uma configuração e "*FooM.nc*" representa o módulo correspondente.

Um resumo das convenções de nomes utilizados no código do *TinyOS* se encontra no site http://today.cs.berkeley.edu/tos/tinyos-1.x/doc/tutorial/naming.html.

6.4.1 **Exemplo: Arquivo de Configuração Blink.nc**

O compilador da linguagem *NesC*, o *ncc*, compila uma aplicação escrita em *NesC*, quando o arquivo dado for de uma configuração de nível mais alto. Tipicamente, aplicações do *TinyOS* apresentam um "*Makefile*" padrão, que permite escolher a plataforma alvo e invocar o *ncc* com as opções apropriadas. A seguir é mostrado o arquivo de configuração da aplicação *Blink*.

```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;

  Main.StdControl -> BlinkM.StdControl;
  Main.StdControl -> SingleTimer.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

Figura 14 - O arquivo de configuração Blink.nc

A primeira observação é a palavra chave “*configuration*”, que indica que este arquivo é de configuração. As duas primeiras identificam que esta é a configuração da aplicação chamada *Blink*. Dentro do par de chaves vazia é possível especificar cláusulas “*uses*” e “*provides*”, assim como em um módulo. Esta é uma observação importante: uma configuração pode usar e prover interfaces.

A configuração é implementada dentro do par de chaves seguido da palavra chave “*implementation*”. A linha “*components*" especifica o conjunto de componentes que a configuração referencia, no nosso caso *Main*, *BlinkM*, *SingleTimer* e *LedsC*. O restante da implementação consiste em conectar as interfaces usadas pelo componente com a interface provida pelos outros componentes.

Em uma aplicação *TinyOS*, o componente que é executado primeiro é o *Main*. Mais precisamente, o comando *Main.StdControl.init()* é o primeiro a ser executado no *TinyOS*, seguido do *Main.StdControl.start().* Uma aplicação *TinyOS* deve possuir um componente *Main* na sua configuração. *StdControl* é a interface comum usada para inicializar os componentes do *TinyOS*. O próximo passo é investigar o arquivo “*tos/interfaces/ StdControl.nc”:*

```
interface StdControl {
  command result_t init();
  command result_t start();
  command result_t stop();
}
```

Figura 15 - O arquivo StdControl.nc

StdControl define três comandos, *init(), start(), e stop(). “init()”* é chamado quando um componente é inicializado pela primeira vez, e “*start()”* é chamado

quando o componente é posto a executar pela primeira vez. “*stop()”* é chamado quando um componente é parado, por exemplo, com o objetivo de desligar o dispositivo que está sendo controlado para economizar energia*. “init()”* pode ser chamado múltiplas vezes, mas nunca depois de chamar *“start()”* ou “*stop()”.* Especificamente, a expressão regular que representa o padrão válido de chamadas do *StdControl* é *init*(start*|*stop)*.* As duas linhas seguintes no arquivo de configuração *Blink* conectam a interface *StdControl* no *Main* com a interface *StdControl* em ambos *BlinkM* e *SingleTimer*.

Main.StdControl -> SingleTimer.StdControl;

Main.StdControl -> BlinkM.StdControl;

SingleTimer.StdControl.init()

e

BlinkM.StdControl.init()

serão chamados pelo **Main.StdControl.init().**

A mesma regra se aplica aos comandos *start()* e *stop().*

A respeito das interfaces usadas (palavra chave “*uses”*), é importante ressaltar que as funções de inicialização de subcomponentes devem ser explicitamente chamadas pelo componente usado. Por exemplo, o módulo *BlinkM* usa a interface Leds, portanto *Leds.init()* deve ser chamada explicitamente em *BlinkM.init().*

NesC usa setas para determinar relacionamentos entre interfaces. Pense na seta a direita (− >) como “ligando a”. O lado esquerdo da seta liga uma interface a uma implementação no lado direito, ou seja, o componente que usa (palavra chave uses) uma interface está na esquerda, e o componente que fornece (palavra chave provides) a interface está na direita.

A linha abaixo é usada para conectar a interface Timer usada pelo BlinkM a interface Timer fornecida pelo SingleTimer.

BlinkM.Timer -> SingleTimer.Timer;

Conexões também são implícitas. Por exemplo, *BlinkM.Leds -> LedsC*; é equivalente a *BlinkM.Leds -> LedsC.Leds*;

6.4.2 **O Módulo BlinkM.nc**

```
module BlinkM {
  provides {
    interface StdControl;

  }
  uses {
    interface Timer;
    interface Leds;
  }
}//Continua abaixo
```

Figura 16 - O módulo BlinkM.nc

A primeira parte do código indica que este é um módulo chamado BlinkM e declara as interfaces fornecidas e utilizadas. O módulo BlinkM fornece a interface StdControl, o que significa que BlinkM implementa a interfaces de StdControl. Como explicado anteriormente, isto é necessário para iniciar o componente Blink e

executá-lo. O módulo BlinkM também utiliza duas interfaces: Leds (diodos emissores de luz) e Timer (temporizador). Isto significa que BlinkM pode chamar qualquer comando declarado nestas interfaces, e deve também implementar quaisquer eventos declarados por estas interfaces.

A interface Leds define alguns comandos, como redOn() e redOff(), que acendem ou desligam os diferentes LEDs (vermelho, verde, ou amarelo) do mote. BlinkM pode invocar qualquer um desses comandos porque ele usa a interface Leds.

Timer é um temporizador. O comando start() é usado para especificar o tipo de temporizador e o seu intervalo, especificados em milissegundos. O tipo TIMER REPEAT indica que o temporizador continua enquanto não for parado pelo comando stop(). Quando o timer expira, ele gera um evento que é recebido pela aplicação. A interface Timer fornece o evento:

event result_t fired();

Eventos podem ser vistos como uma função de retorno que a implementação de uma interface irá invocar. Um módulo que usa uma interface deve implementar os eventos que esta interface usa.

BlinkM.nc, continuação

```
implementation {
  command result_t StdControl.init() {
    call Leds.init();
    return SUCCESS;
  }

  command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, 1000) ;
  }

  command result_t StdControl.stop() {
    return call Timer.stop();
  }

  event result_t Timer.fired()  {
    call Leds.redToggle();

    return SUCCESS;
  }
}
```

Figura 17 - Continuação do módulo BlinkM.nc

O módulo BlinkM implementa os comandos *StdControl.init(), StdControl. start() e StdControl.stop()*. Ele também implementa o evento *Timer.fired()*, que é necessário uma vez que *BlinkM* deve implementar todos os eventos das interfaces utilizadas.

O comando "*init()"* inicializa o sub-componente *Leds*. O comando *"start()"* invoca *Timer.start()* no intuito de criar um temporizador cíclico que expira a cada 1000 ms. O comando *"stop()"* termina o temporizador. Cada vez que o temporizador expira, um evento é gerado e *Timer.fired()* é acionando, modificando o estado do

diodo vermelho. A linha *Leds.redToggle()* apaga o led se ele está acesso ou o acende se ele está apagado.

6.4.3 **Compilando a Aplicação**

Com o nó sensor conectado ao computador e utilizando um terminal aberto no diretório de instalação do *TinyOS* é possível compilar e carregar a aplicação no hardware.

Para compilar a aplicação *Blink* para o *Mica Motes* digite:

make mica

O makefile irá executar automaticamente os aplicativos descritos acima. Estes também podem ser invocados na linha de comando.

O compilador *NesC* é acionando quando se digita:

ncc -o main.exe -target=mica Blink.nc

A aplicação *Blink* é compilada, e é gerado um arquivo executável *“main.exe”* para o *Mica motes*.

Antes de carregar o código no *mote*, use

avr-objcopy --output-target=srec main.exe main.srec

para produzir o arquivo *main.srec*, que é o arquivo usado para programar o *mote*.

6.4.4 **Gerando a Documentação**

Também é possível gerar documentação e diagramas que representam a conexão entre os componentes.

O comando:

make <plataform> docs

gera documentação e o diagrama de componentes da aplicação.

7 CONCLUSÃO

As redes de sensores sem fio se tornaram parte do nosso mundo conectado, e sua importância tem crescido tremendamente. Como elas nasceram como sistemas autônomos de monitoramento e vigilância, têm evoluído em um componente chave das redes de próxima geração e para a futura internet. Têm sido a ponte entre o mundo digital e o mundo físico. A flexibilidade, tolerância a falhas, alta fidelidade de sensoriamento, o baixo custo e rápida implantação, são características das redes de sensores, e vão criar muitas áreas e aplicações novas para sensoriamento remoto. Apesar destas áreas de aplicação existentes, as redes de sensores sem fio continuam a ser um campo interessante e aberto de pesquisa. Várias questões técnicas ainda têm que ser totalmente tratadas e resolvidas. Tais abordagens bem sucedidas exigem avançados paradigmas, conhecimento interdisciplinar e soluções de *cross-layer*.

Algumas áreas potenciais onde a pesquisa pode ser tomada em redes de sensores sem fio são auto-configuração, tolerância a falhas, adaptação, flexibilidade, eficiência energética, projeto e operação de protocolos eficientes, escalabilidade e heterogeneidade, segurança, privacidade e confiança, questões de arquitetura, a interface com o sistema global, a interoperabilidade com protocolos padrões existentes, endereçamento, nomeação ,localização, operação de dados centralizada e mobilidade e tolerância a atraso.

As redes de sensores sem fio têm grande potencial econômico a longo prazo, a capacidade de transformar as nossas vidas, e representam muitos novos desafios de construção do sistema.

REFERÊNCIAS

[1] SARASWAT, Lalit; YADAV, Pankaj S. **A Comparative Analysis of Wireless Sensor Network Operating Systems**. Ghaziabad, India: Department Of Computer science, Raj Kumar Goel Institute of Technology, 2010. 7p

[2] BHATTACHARYYA, Debnath; KIM, Tai-Hoon; PAL, Subhajit. **A Comparative Study of Wireless Sensor Networks and Their Routing Protocols**. Daejeon, Korea: Department of Multimedia Engineering, Hannam University, 2010. 18p

[3] CHIEN, Thang V.;CHAN, Hung N.;HUU, Thanh N. **A Comparative Study on Operating System for Wireless Sensor Networks**. Quyet Thang Commune, Thai: Thai Nguyen University of Information and Communication Technology, 2011. 6p

[4] PARK, Seungmin; KIM, Jin W.; SHIN, Kee-Young; KIM, Daeyoung. **A Nano Operating System for Wireless Sensor Networks**. Daejeon, South Korea: Electronics and Telecommunications Research Institute, [20??]. 4p

[5] TALEGHAN, Majid A.; TAHERKORDI, Amirhosein; SHARIFI, Mohsen,. **A Survey of System Software for Wireless Sensor Networks**. Daejeon, Korea: Hannam University, [20??]. 6p

[6] HASAN, Shabbir; HUSSAIN, Md. Zair; SINGH, R.K. **A Survey of Wireless Sensor Network**. International Journal of Emerging Technology and Advanced Engineering. Vol 3, Issue 3, Mar/2013. 6p

[7] OMIYI, Eitan; KAAN, Bür; YANG, Yang . **Technical Survey of Wireless Sensor Networkk Platforms, Devices and Testbeds**. London, UK: University College London, 2008. 26p

[8] ROCA, Ivan; MORAES, Fernando de; KOFUJI, Sergio T. **Aplicações das Redes de Sensores sem fio (RSSF) na Engenharia Naval e Oceânica**. São Paulo: Escola Politécnica da Universidade de São Paulo, 2007. 35p

[9] RUIZ, Linnyer B.; CORREIA, Luiz H. A.; VIEIRA, Luiz F. M.; VIEIRA, Marcos A. M.; MECHELANE, Eduardo H.; CAMARA, Daniel; LOUREIRO, Antonio A. F.; NOGUEIRA, José M. S.; JÚNIOR, Diógenes C. da S. **Arquiteturas para Redes de Sensores Sem Fio**. Belo Horizonte: DCC/UFMG, [20??]. 52p

[10] VIEIRA, Marcos A. M. **BEAN: Uma Plataforma Computacional para Rede de Sensores Sem Fio**. Orientador: Diógenes Cecílio da Silva Júnior. Belo Horizonte, 2004. xp. Dissertação. (Mestrado em Ciência da Computação)- Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, 2004. 143p

[11] UGLIARA, Fellipe A. **Desenvolvimento de Plataformas para Redes de Sensores sem Fio Baseadas no Sistema Operacional TinyOS**. Lavras: UFLA, 2010. 92p

[12] RODRIGUES, Taniro C. **Abordagem Dirigida a Modelos para Redes de Sensores sem Fio**. Natal: UFRN, 2011. 103p

[13] CONDEMINE, C. **Energy-driven and event-driven WSN platforms**. Minatec Campus, 2012. 73p

[14] ROSANES, Pedro P. C.; ROSSETTO, Silvana. Extensão dos mecanismos de gerência de tarefas do sistema operacional TinyOS. **Anais do Congresso de iniciação científica do Inatel** - Incitel 2012. p. 161 a 169. 2012.

[15] HANDZISKI, Vlado; POLASTRE, Joseph; HAUER, Jan-Hinrich; SHARP, Cory; WOLISZ, Adam; CULLER, David. Flexible Hardware Abstraction for Wireless Sensor Networks . **In Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN 2005).** 2005.

[16] MARGI, Cintia B.; OLIVEIRA, Bruno T. de; SOUSA, Gustavo T.; JÚNIOR, Marcos A.S.; BARRETO, Paulo S. L. M.; CARVALHO, Tereza C. M. B.; NÄSLUND, Mats; GOLD, Richard. **Impact of Operating Systems on Wireless Sensor Networks (Security) Applications and Testbeds**. São Paulo: Escola Politécnica Universidade de São Paulo, [20??]. 6p

[17] BERNDT, Alexandre. **Introdução a Redes de Sensores sem Fio (RSSF).** Barra do Bugres: Departamento de Ciência da Computação, UNEMAT, [20??] . 15p

[18] FAROOQ, Muhammad; KUNZ, Thomas . **Operating Systems for Wireless Sensor Networks: A Survey**. Ottawa: Department of Systems and Computer Engineering, Carleton University Ottawa, Canada, 2011. 31p

[19] PORTER, Barry; COULSON, Geoff . **Lorien: A pure dynamic component-based Operating System for Wireless Sensor Networks**. Lancaster: Computing Department of Lancaster University. Lancaster, England, [20??]. 6p

[20] JÚNIOR, Arliones H. ; FRÖHLICH, Antonio A. **Redes de sensores sem-fio sob a perspectiva do EPOS**. Florianópolis: LISHA/UFSC, 2010. 39p

[21] SCHILLER, Jochen; LIERS, Achim; RITTER, Hartmut . **ScatterWeb: A wireless sensornet platform for research and teaching**. Berlin: Institute of Computer Science, Freie Universita¨t Berlin, 2005. 7p

[22] ALMEIDA, Vinícius C. de; VIEIRA, Luiz F. M.; VITORINO, Breno A. D.; VIEIRA, Marcos A. M.; NACIF, José A. FERNANDES, Antônio O.; JUNIOR, Diogenes C. da S.; COELHO, Claudionor N. **Sistema Operacional YATOS para Redes de Sensores sem Fio**. Belo Horizonte: DCC/UFMG, [20??]. 10p

[23] AKYILDIZ, I. F.; SU, W.; SANKARASUBRAMANIAM, Y.; CAYRICI, E. **Wireless sensor networks: a survey**. Atlanta: Broadband and Wireless

Networking Laboratory, School of Electrical and Computer Engineering, Georgia Institute of Technology, USA, 2002. 30p

[24] GAY, David; LEVIS, Philip; CULLER, David; BREWER, Eric . **NesC 1.3 Language Reference Manual** . 2009. 46p

[25] FAROOQ, Muhammad O.; AZIZ, Sadia; DOGAR, Abdul B. **State of the Art in Wireless Sensor Networks Operating Systems: A Survey**. Bremen: Transmission Systems Research Group, Jacobs University Bremen, Germany, [20??]. 16p

[26] HEMPSTEAD, Mark; LYONS, Michael J.; BROOKS, David; WEI, Gu-Yeon . Survey of Hardware Systems for Wireless Sensor Networks. **Journal of Low Power Electronics**. Vol. 4, 1–10. American Scientific Publishers, 2008. 10p

[27] VIEIRA, Marcos A. M.; JUNIOR, Claudionor N. C.; JUNIOR, Diógenes C. da S.; MATA, José M. da . **Survey on Wireless Sensor Network Devices**. Belo Horizonte: DCC/UFMG, [20??]. 8p

[28] HILL, Jason; HORTON, Mike; KLING, Ralph, KRISHNAMURTHY, Lakshman . The Platforms Enabling Wireless Sensor Networks. **Communications of the ACM**. Vol 47, No 6, Jun/2004. 6p

[29] LEVIS, Philip . **TinyOS Programming** . 2006. 139p

[30] KARANI, Manish; KALE, Ajinkya; KOPEKAR, Animesh . Wireless Sensor Network Hardware Platforms and Multi-channel Communication Protocols: A Survey. **International Journal of Computer Applications® (IJCA), 2nd National Conference on Information and Communication Technology (NCICT).** Department of Electronics Engineering, Vishwakarma Institute of Technology PUNE, 2011. 4p

[31] BISHCHOFF, Reinhard; MEYER, Jonas; FELTRIN, Glauco . **Wireless Sensor Network Platforms.** Dübendorf: Structural Engineering Research Laboratory, Empa, Swiss Federal Laboratories for Materials Testing and Research, Dübendorf, Switzerland, [20??]. 10p

[32] TRENKAMP, Peter . **Wireless Sensor Network Platforms − Datasheets versus Measurements**. Bremen, ITG Fachgruppen 5.2.1/5.2.4. University Bremen, Germany, 2011. 21p

[33] YICK, Jennifer; MUKHERJEE, Biswanath; GHOSAL, Dipak . **Wireless sensor network survey.** California: Department of Computer Science, University of California, 2008. 39p

[34] YADAV, S.G. Shiva P.; CHITRA, A. Wireless Sensor Networks - Architectures, Protocols, Simulators and Applications: a Survey. **International Journal of Electronics and Computer Science Engineering**, Vol 1, No 4, p. 1941-1953, [20??].

[35] SHAIKH, Faisal K. **WSN Platforms, Hardware & Software**. Dependable Embedded Wired/Wireless Networks, [20??] . 25p

[36] Loureiro, A. A., Ruiz, L. B., Nogueira, J. M. S., and Mini, R. A. Rede de sensores sem fio. In Porto, I. J., editor, **Simpósio Brasileiro de Computação**, Jornada de Atualização de Informática, p 193–234. 2002

[37] Badrinath, B. R., Srivastava, M., Mills, K., Scholtz, J., and Sollins, K. **Special issue on smart spaces and environments**. IEEE Personal Communications. 2000.

[38] National Science Foundation. **Report of the National Science Foundation Workshop on Fundamental Research in Networking**. 2004. Disponível em: http://www.cs.virginia.edu/\~{}jorg/workshop. Acesso em 27 de fevereiro de 2014

[39] CAVALCANTE, André; ALLGAYER, Rodrigo; MÜLLER, Ivan; BALBINOT, Jovani; PEREIRA, Carlos E. **Analise da Plataforma SunSPOT para Programacao de Sistemas de Controle Distribuido em Rede de Sensores sem Fio**. Porto Alegre: Departamento de Engenharia Elétrica – Universidade Federal do Rio Grande do Sul (UFRGS), 2010. 6p

[40] DONASSOLO, Bruno. **Analise do Middleware para Rede de Sensores Sun SPOT**. Disponível em: https://saloon.inf.ufrgs.br/twiki-data/Disciplinas/CMP167/TF08BrunoDonassolo/TF08_SunSPOT.pdf. Acesso em 12 mar. 2014

[41] AVELAR, Edson A. M.; AVELAR, Lorena M.; SILVA, Diego dos P. S.; DIAS, Kelvin L. **Arquitetura de Comunicação para Cidades Inteligentes: Uma proposta heterogênea, extensível e de baixo custo**. Recife: Centro de Informática – Universidade Federal de Pernambuco (UFPE), 2012. 12p

[42] ARAUJO, Rodrigo P. M. de; PORTOCARRERO, Jesús M. T.; DELICATO, Flávia C.; PIRES, Paulo F.; PIRMEZ, Luci; BATISTA, Thais; ROSSETO, Silvana; OLIVEIRA, Ana L. S. **Middleware Baseado em Componentes e Orientado a Recursos para Redes de Sensores sem Fio**. XXX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos. P 219-232. 2011.

[43] CAICEDO, Ângela. **The Sun Small Programmable Object Technology (Sun SPOT): Java(tm) Technology-Based Wireless Sensor Networks**. Sun Tech Days 2006-2007. Disponível em: http://www.austinjug.org/presentations/SunSpots.pdf. Acesso em 12 mar. 2014.

[44] HORAN, Bernard. **Sun SPOTs**. Disponível em: http://academic.fuseyism.com/ambassador/slides/sunspot.pdf. Acesso em 12 mar. 2014.

[45] VAQUERO, J. M. **Historical Sunspot Observations A Review**. Departamento de Física Aplicada, Universidad de Extremadura, Cáceres, Spain. Disponível em: http://arxiv.org/ftp/astro-ph/papers/0702/0702068.pdf. Acesso em 11 mar. 2014.

[46] CARACAS, A.; KRAMP, T.; BAENTSCH, M; OESTREICHER, M.; EIRICH, T.; ROMANOV, I. **Mote Runner: A Multi-Language Virtual Machine for Small Embedded Devices**. IBM Zurich Research Laboratory. Disponível em: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5210937. Acesso em 11 mar. 2014.

[47] CARACAS, A.; LOMBRISER, C.; PIGNOLET, Y. A.; KRAMP, T.; EIRICH, T.; ADELSBERGER, R.; HUNKELER, U.; **Energy-Efficiency Through Micro-Managing Communication and Optimizing Sleep**. Disponível em: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5984943. Acesso 11 mar. 2014

[48] LIBELIUM. Waspmote**: Technical Guide**. Disponível em: http://www.libelium.com/v11-files/documentation/waspmote/waspmote-technical_guide_eng.pdf. Acesso em: 11 mar. 2014.

[49] BRAGA, Tiago C. **Monitorização Ambiental em Espaços Florestais com Rede de Sensores Sem Fios**. Orientador: Joaquim Amândio Rodrigues Azevedo. Funchal, 2010. 164p. Dissertação. (Mestrado em Engenharia de Telecom e Redes) - Centro de Competência de Ciências Exactas e da Engenharia, Universidade da Madeira, 2010.

[50] LIBELIUM. **Waspmote Hardware Overview**. Disponível em: http://www.libelium.com/products/waspmote/hardware/. Acesso em 11 mar. 2014-03-14

[51] IBM. **IBM Mote Runner White Paper**. Disponível em: http://www.zurich.ibm.com/pdf/csc/Mote_Runner_WP.pdf. Acesso em 10 mar. 2014-03-14