**UNIVERSIDADE FEDERAL DE SANTA CATARINA**
**PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA**

Cládio José Martins Júnior

# RIVERSENSE: UM SISTEMA PARA MONITORAMENTO DE RIOS ATRAVÉS DE REDES DE SENSORES SEM FIO

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina para a obtenção do Grau de Mestre em Engenharia Elétrica.
Orientador: Prof. Dr. Eduardo Augusto Bezerra
Co-orientador: Prof. Dr. Jorge Luis Victória Barbosa

Florianópolis
2013

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Martins Júnior, Cládio José
RIVERSENSE : Um Sistema para Monitoramento de Rios através de Redes de Sensores Sem Fio [dissertação] / Cládio José Martins Júnior ; orientador, Eduardo Augusto Bezerra ; coorientador, Jorge Luis Victória Barbosa. - Florianópolis, SC, 2013.
125 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Engenharia Elétrica.

Inclui referências

1. Engenharia Elétrica. 2. sensoriamento remoto. 3. redes de sensores sem fio. 4. arquitetura. 5. monitoramento ambiental. I. Bezerra, Eduardo Augusto. II. Barbosa, Jorge Luis Victória. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia Elétrica. IV. Título.

Cládio José Martins Júnior

**RIVERSENSE:**
**UM SISTEMA PARA MONITORAMENTO DE RIOS ATRAVÉS DE REDES DE SENSORES SEM FIO**

Esta Dissertação foi julgada adequada para obtenção do Título de Mestre em Engenharia Elétrica, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.

Florianópolis, __ de ___________ de 2013.

________________________
Prof. Patrick KuoPeng, Dr.
Coordenador

**Banca Examinadora:**

________________________________
Prof. Eduardo Augusto Bezerra, Dr.
Presidente (Orientador)
Universidade Federal de Santa Catarina

________________________________
Prof. Roberto Alexandre Dias, Dr.
Instituto Federal de Santa Catarina

________________________________
Prof. Djones Vinicius Lettnin, Dr.
Universidade Federal de Santa Catarina

________________________________
Prof. Rafael Luiz Cancian, Dr.
Universidade Federal de Santa Catarina

*Dedico este trabalho à minha família que sempre acreditou no meu potencial.*

## AGRADECIMENTOS

Agradeço a todos que de alguma forma, direta ou indiretamente, participaram da minha caminhada no desenvolvimento deste trabalho. Ao professor Eduardo Bezerra, orientador deste trabalho, pelos seus significantes conhecimentos, sua atenção e sua boa vontade. Jamais esquecerei pela oportunidade dada a mim no Grupo de Sistemas Embarcados (GSE). Agradeço a cooperação do Laboratório de Integração de Software e Hardware (LISHA) e do Laboratório de Hidrologia (LABHIDRO), sendo fundamentais para o desenvolvimento deste trabalho.

Agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo suporte financeiro durante esses dois anos e a companhia dos meus colegas de laboratório que me proporcionaram os momentos dos mais variados tipos possíveis. Agradeço a minha família pelo apoio incondicional desde o início da minha caminhada na carreira acadêmica, pelo carinho nos momentos difíceis, e claro, pelo apoio financeiro.

Dedico este trabalho, a todos aqueles que participaram do desenvolvimento deste projeto.

Aos integrantes do GSE, Eduardo Bezerra, Djones Lettnin, Paulo Villa, Frederico Ferlini, Roberto de Matos, Wagner Guadanin, Tomás Grimm, William Jamir e Pedro Ceriotti.

Aos integrantes do LISHA, Peterson de Oliveria, Arliones Höeller Jr., Renato Bock, Alexandre Okazaki e Antônio Augusto Fröhlich.

Aos integrantes do LABHIDRO, Fernando Grison e Masato Kobiyama.

Aos integrantes do LAMAN (Laboratório de Manutenção), aos integrantes do LMM (Laboratório de Montagem Mecatrônica) e ao USICON (Laboratório de Usinagem e Comando Numérico).

Meus mais sinceros agradecimentos a todas estas pessoas.

"Nothing is impossible, the word itself says 'I'm possible'!"

Audrey Hepburn

# RESUMO

As inundações representam um importante problema a nível mundial, que tem crescido acentuadamente, provavelmente, devido às mudanças climáticas no planeta, impactando principalmente em microclimas sensíveis. Este problema é considerado no estado de Santa Catarina, Brasil, onde as inundações frequentes durante a estação chuvosa causam mortes e significantes prejuízos financeiros. Esforços para minimizar estes problemas têm sido realizados nas áreas de monitoramento ambiental, modelagem hidrológica e gerenciamento de desastres. Na área de monitoramento ambiental uma tecnologia promissora para dar suporte a sua aplicação são as redes de sensores sem fio (RSSFs). As RSSFs são compostas por pequenos computadores embarcados conhecidos como *motes*, que basicamente possuem um processador de baixo consumo de energia, comunicação sem fio e sensores. Estes *motes* podem formar redes, que são capazes de se auto-organizar e reagir ao ambiente físico. Em áreas como hidrologia e micro meteorologia, as RSSFs possibilitam a disponibilidade de dados espaciais e temporais, que tem sido um grande desafio, devido a custos e o tamanho de estações meteorológicas tradicionais. Além disso, o governo brasileiro, através do projeto CIA², identifica as RSSFs como uma possível solução para o monitoramento de rios em busca de alertar a população sobre possíveis inundações. Este trabalho visa a concepção de uma RSSF de baixo custo e consumo de energia para monitoramento de rios, objetivando melhorar coletas de dados e disponibilizar o acesso público a dados ambientais. Ao final da pesquisa foi possível propor e implementar uma arquitetura de rede de sensores, e todo um sistema de monitoramento de inundações. Os resultados quantitativos obtidos com a utilização desse sistema concluem a sua viabilidade.

**Palavras-chave**: sensoriamento remoto, redes de sensores sem fio; arquitetura; implantação; monitoramento ambiental.

# ABSTRACT

Flooding is a worldwide critical problem, which has been growing increasingly due to climate changes on the planet, mainly impacting in sensible microclimates. This problem is pointed out in the state of Santa Catarina, Brazil, where the floods during the rainy season cause significant human and financial costs. Efforts to minimize these problems have been aimed in the areas of environmental monitoring, hydrological modeling and disaster management. In the environmental monitoring area a promising technology for supporting its application is the Wireless Sensor Networks (WSN). WSN are composed of tiny embedded computers known as motes, which basically features a low-power processor, wireless communication and sensors. These motes can form networks, which are capable of self-organize and react to the physical environment. In areas such as hydrology and micrometeorology, the WSN can help with the availability of spatial and temporal data, which has generally been lacking, due to the costs and size of traditional sensing stations. Furthermore, the Brazilian Government, through the CIA² project, identifies WSNs as a possible solution for river monitoring in pursuit to alert the population about possible river flooding events. This work concerns the design of a low-cost and low-power WSN for river monitoring, aiming to improve data collection and provide public access to environmental data. At the end of the research it was possible to propose and implement a sensor network architecture, and a whole system for floods monitoring. The quantitative results conclude the system feasibility.

**Keywords**: remote sensing; wireless sensor networks; architecture; deployment; environmental monitoring.

# LISTA DE FIGURAS

Figura 1 – Ocorrências de desastres naturais entre 1960 e 2008...................... 25
Figura 2 – Principais componentes de um nodo sensor sem fio. ...................... 30
Figura 3 - Processo de integração e adaptação dos componentes para produzir uma imagem de sistema EPOS ...................................................................... 33
Figura 4- Diagrama UML ilustrando as relações entre os principais conceitos do sistema EPOS ......................................................................................... 34
Figura 5 – Kit de desenvolvimento do EPOSMoteII ....................................... 35
Figura 6 – Visão geral de funcionamento do Terminal Serial GPRS............... 37
Figura 7 - Princípio de funcionamento do sensor de pressão por coluna da água ............................................................................................................... 38
Figura 8 - Esquemático de conexão e interface de leitura do sensor de nível... 39
Figura 9 - Diagrama de funcionamento do pluviômetro de báscula ................ 40
Figura 10 - Procedimento de leitura através da SPI......................................... 41
Figura 11 - Gráfico de comportamento do sensor de radiação solar................ 42
Figura 12 - Gráfico de comportamento do sensor de radiação ultravioleta....... 43
Figura 13 - Diagrama de funcionamento do sensor de velocidade do vento..... 43
Figura 14 - Funcionamento do sensor de direção do vento.............................. 44
Figura 15 - Resultado de pesquisa do termo "Remote Sensing" no ieeexplore. 47
Figura 16 – Visão geral da arquitetura do *Sistema de Alerta Temprana*........... 48
Figura 17 – Visão geral dos componentes de software do middleware DisSeNT ............................................................................................................... 49
Figura 18 - Visão geral da arquitetura do SensorScope .................................. 50
Figura 19 - Visão geral da arquitetura do framework *Ecology-Hidrology Wireless Sensor Network* ........................................................................... 51
Figura 20 – Visão geral da arquitetura do RiverSense..................................... 54
Figura 21 - Arquitetura do RiverSense Server ................................................ 55
Figura 22 - Tipos de mensagens trocadas entre a RSSF e o Servidor.............. 55
Figura 23- Banco de dados do RiverSense Server .......................................... 56
Figura 24 - Elementos que compõem o RiverSense RSSF .............................. 57
Figura 25 – Diagrama de implantação do protótipo RiverSense...................... 58
Figura 26 - Diagrama de componentes do RiverSense Server.......................... 60
Figura 27 - Diagrama de sequência - Recebimento de mensagens .................. 61
Figura 28 - Página web para visualização gráfica........................................... 63
Figura 29 - Visão geral dos componentes do RiverSense RSSF...................... 64
Figura 30 - Visão geral das classes do nodo sincronizador e suas dependências ............................................................................................................... 65
Figura 31 - Fluxo de programa simplificado do nodo sincronizador ............... 67
Figura 32 - Interface de expansão disponibilizada pelo Terminal Serial GPRS 68
Figura 33- Visão geral das classes do nodo estação meteorológica e suas dependências................................................................................................ 69
Figura 34 - Fluxo de programa simplificado do nodo estação meteorológica .. 71
Figura 35 - Placa de interface para sensores da estação meteorológica ........... 72

Figura 36 - Visão geral das classes do nodo estação de nível e suas dependências ....................................................................................... 73
Figura 37 - Gráfico de comportamento do sensor de nível ............................... 74
Figura 38 - Placa de interface para sensor da estação de nivel .......................... 75
Figura 39 - Cenário proposto para realizar a avaliação do sistema .................... 77
Figura 40 - Teste do sensor de nível realizado em laboratório .......................... 78
Figura 41 - Teste do sensor de temperatura realizado em laboratório ............... 78
Figura 42 - Leituras inesperadas realizadas pelo sensor de temperatura ........... 79
Figura 43 - Teste do sensor de radiação solar realizado em laboratório ............ 79
Figura 44 - Leituras inesperadas realizadas pelo sensor de radiação ................. 80
Figura 45 - Round-Trip Time medido da conexão GPRS no nodo sincronizador ....................................................................................................... 81
Figura 46 - Variação da vazão na exutória da bacia do campus da UFSC com base na precipitação local ........................................................................ 82
Figura 47 - Comportamento da exutória da bacia do campus da UFSC ............ 82
Figura 48 - Nível de bateria durante o teste em laboratorio ............................... 84
Figura 49 - Nível de bateria no momento em que os sensores pararam de funcionar como esperado ........................................................................ 84
Figura 50 – Diagrama de circuito da interface para a estação meteorológica .... 96
Figura 51 – Diagrama de circuito da interface para o sensor de pressão ........... 97

# LISTA DE TABELAS

Tabela 1- Formato dos dados de leitura do sensor TMP123 ............................. 40
Tabela 2 - Sensores disponíveis para o desenvolvimento do sistema ............... 45
Tabela 3 – Características dos trabalhos relacionados ...................................... 51
Tabela 4 – Web Services criados para consulta ................................................ 62

## LISTA DE ABREVIATURAS E SIGLAS

EPOS – Embedded Parallel Operating System
GPRS – General Packet Radio Service
GPS – Global Positioning System
GSE – Grupo de Sistemas Embarcados
IP – Internet Protocol
ISM – Industrial, Scientific and Medical radio bands
LCS – Laboratório de Comunicações e Sistemas Embarcados
LISHA – Laboratório de Integração de Software e Hardware
LABHIDRO – Laboratório de Hidrologia
MAC – Medium Access Control
MEMS – Micro Electrical Mechanical Systems
MySQL – My Structured Query Language
RSSF – Rede de Sensores Sem Fio
RTC – Real Time Clock
SICS – Swedish Institute of Computer Science
TSG – Terminal Serial GPRS
WMO – World Meteorological Organization

# SUMÁRIO

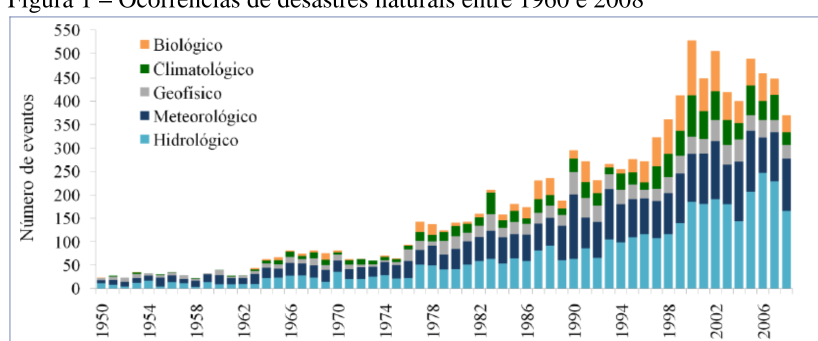
**1 INTRODUÇÃO ........ 25**
1.1 MOTIVAÇÃO ........ 26
1.2 OBJETIVOS ........ 26
1.3 ESTRUTURA DO TRABALHO ........ 27
**2 FUNDAMENTAÇÃO TEÓRICA ........ 29**
2.1 REDES DE SENSORES SEM FIO ........ 29
**2.1.1 Evolução da RSSF ........ 30**
**2.1.2 O Sistema Operacional EPOS ........ 32**
2.2 TECNOLOGIAS UTILIZADAS ........ 34
**2.2.1 EPOSMote ........ 34**
**2.2.2 Terminal Serial GPRS ........ 35**
**2.2.3 Sensores ........ 37**
2.2.3.1 Nível do rio ........ 37
2.2.3.2 Pluviosidade ........ 39
2.2.3.3 Temperatura ........ 40
2.2.3.4 Radiação solar ........ 41
2.2.3.5 Radiação ultravioleta ........ 42
2.2.3.6 Velocidade e direção do vento ........ 43
2.3 CONSIDERAÇÕES SOBRE O CAPÍTULO ........ 45
**3 TRABALHOS RELACIONADOS ........ 47**
3.1 SISTEMA DE ALERTA TEMPRANA ........ 47
3.2 REDE ........ 48
3.3 SENSORSCOPE ........ 49
3.4 ECOLOGY-HIDROLOGY ........ 50
3.5 CONSIDERAÇÕES SOBRE O CAPÍTULO ........ 51
**4 SISTEMA PROPOSTO ........ 53**
4.1 METODOLOGIA ........ 53
4.2 ARQUITETURA ........ 53
**4.2.1 RiverSense Server ........ 54**
**4.2.2 RiverSense RSSF ........ 57**
4.3 IMPLEMENTAÇÃO ........ 58
**4.3.1 RiverSense Server ........ 59**
4.3.1.1 Fluxo de mensagens ........ 60
4.3.1.2 Serviços ........ 62
4.3.1.3 Página web ........ 62
**4.3.2 RiverSense RSSF ........ 63**
4.3.2.1 Nodo Sincronizador ........ 64
4.3.2.2 Nodo Estação Meteorológica ........ 69
4.3.2.3 Nodo Estação de Nível ........ 72
4.4 CONSIDERAÇÕES SOBRE O CAPÍTULO ........ 75
**5 RESULTADOS OBTIDOS ........ 77**
5.1 IMPLANTAÇÃO ........ 77

5.2 AVALIAÇÃO ........................................................................................ 83
5.3 CONSIDERAÇÕES SOBRE O CAPÍTULO ............................................ 84
**6 CONCLUSÃO ........................................................................ 85**
**REFERÊNCIAS ........................................................................ 89**
**APÊNDICE A – (DIAGRAMA DE CIRCUITOS) ............................ 95**
**APÊNDICE B – (CÓDIGO FONTE) .................................................. 99**
**APÊNDICE C – (TUTORIAL DE SETUP) ......................................... 125**

# 1 INTRODUÇÃO

A situação mundial das inundações resulta em um grande impacto na sociedade, principalmente, em ambientes com microclimas sensíveis (DOUGLAS, 1997; TASCA et al., 2009). A Figura 1 ilustra a distribuição anual dos cinco principais desastres naturais que ocorrem no mundo, sendo que os desastres hidrológicos possuem a maior taxa de crescimento, o qual tem grande participação, de maneira intensa e severa, na causa de perdas de vidas e custos financeiros para a sociedade (MACIEL et al., 2009). Este problema é particularmente acentuado no estado de Santa Catarina, Brasil, mais especificamente no rio Itajaí-Açu no Vale do Itajaí, onde as inundações durante a estação chuvosa causam mortes e prejuízos significativos (ROCHA; KOBIYAMA, 2009).

Figura 1 – Ocorrências de desastres naturais entre 1960 e 2008




Adaptado de (TASCA et al., 2009)

Esforços para minimizar estes problemas têm sido realizados nas áreas de monitoramento ambiental, modelagem hidrológica e gerenciamento de desastres. As redes de sensores sem fio (RSSFs) são consideradas uma tecnologia promissora para aplicações na área de monitoramento ambiental (FROHLICH; STEINER; RUFINO, 2011; KERKEZ; GLASER, 2011; TRUBILOWICZ; CAI; WEILER, 2009). As RSSFs são compostas por pequenos computadores embarcados conhecidos como *motes*, que basicamente possuem um processador de baixo consumo de energia, comunicação sem fio e sensores para monitoramento. Estes *motes* podem formar redes que são capazes de se auto-organizar e reagir ao ambiente físico, permitindo as RSSFs serem altamente flexíveis e versáteis (OKAZAKI, 2011; ROMER, 2004). O uso de RSSFs em vários domínios de aplicação tem sido largamente estudado, desde o monitoramento de animais (WANG et al., 2005), na agricultura de precisão, (LANGENDOEN; BAGGIO; VISSER, 2006)

até no monitoramento ambiental (INGELREST et al., 2010), por exemplo, o estudo do comportamento de vulcões (VARAPRASAD, 2009).

Em áreas como hidrologia e micro meteorologia, as RSSFs possibilitam a disponibilidade de dados espaciais e temporais (LUNDQUIST; CAYAN; DETTINGER, 2003), o que tem sido um grande desafio devido aos custos e tamanho de estações meteorológicas tradicionais. Além disso, o governo brasileiro, através do projeto CIA² (Construindo Cidades Inteligentes da Instrumentação dos Ambientes ao desenvolvimento de Aplicações) (RNP, 2011), identifica as RSSFs como uma solução para o monitoramento de rios em busca de alertar a população sobre possíveis inundações.

## 1.1 MOTIVAÇÃO

O monitoramento de rios é essencial para as populações ribeirinhas e para várias aplicações ao longo do rio. Aplicações típicas são o controle da capacidade energética para hidrelétricas e sistemas de aviso de inundações. Além disso, as informações coletadas podem ser utilizadas por sistemas que informam a possibilidade de atividades de lazer, e.g., atividades de pesca ou canoagem. O monitoramento e registro de informações de rios vêm de uma longa tradição. No passado isso era feito manualmente, atualmente a medição é realizada por sistemas de informação que medem tais informações automaticamente.

As RSSFs podem aumentar a precisão e a cobertura desses sistemas de informação, possibilitando o uso de dezenas, centenas ou milhares de sensores para formar uma rede a fim de cobrir uma ampla área. Os *motes*, também conhecidos como nós ou nodos, precisam ser capazes de medir as informações de interesse do sistema e de comunicarem-se uns com os outros. Dessa forma os dados coletados podem ser enviados através de outros nodos para o nodo *gateway*, que por sua vez são armazenados em um servidor.

A fim de evitar dualidades, neste trabalho o termo *mote* refere-se ao hardware específico de um nodo, ou seja, a sua relação em uma rede não importa, já um nodo refere-se a um ponto de interconexão em uma rede de computadores.

## 1.2 OBJETIVOS

O presente trabalho visa o projeto, implementação e avaliação de um sistema para monitoramento de rios baseado na plataforma

EPOSMote II (LISHA, 2010). Neste tempo, foi desenvolvido um protótipo do hardware que realiza a interface dos sensores com o nodo e a integração de meios de comunicação com o mundo externo. Assim como o software escrito especialmente para compreender o comportamento dos sensores. O protótipo permitiu testar o consumo de energia de um nodo, a sincronização de relógio entre os nodos na rede, e a fidelidade dos dados coletados pelos sensores. Os objetivos específicos a serem alcançados são:

- Medir variáveis climáticas locais relacionadas ao nível do rio;
- Avaliar a plataforma EPOSMote em aplicações;
- Permitir a possibilidade de conectar sensores adicionais ao nodo;
- Prover a visualização de dados por meio de uma página *web* pública;
- Prover serviços para uso das informações por outros sistemas.

## 1.3 ESTRUTURA DO TRABALHO

Este trabalho está organizado da seguinte forma: o capítulo 2 apresenta a fundamentação teórica utilizada; no capítulo 3 são apresentadas os trabalhos relacionados; o capítulo 4 apresenta tecnologias utilizadas, o fluxo de desenvolvimento e a arquitetura do sistema; no capítulo 5 são discutidos os resultados com a finalidade de avaliar o protótipo; por fim, no capítulo 6, são apresentadas as conclusões e trabalhos futuros.

# 2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são descritos os conceitos básicos e as tecnologias utilizadas no desenvolvimento do trabalho.

## 2.1 REDES DE SENSORES SEM FIO

Uma Rede de Sensores Sem Fio (RSSF) pode ser vista como uma ferramenta utilizada para a captura de informações do ambiente, tanto temporais quanto espaciais em grande escala. Uma rede de sensores sem fio é composta por um conjunto de nodos, onde cada nodo, normalmente, consiste de um microcontrolador, sensores/atuadores, uma fonte de energia e um dispositivo de comunicação sem fio (HILL; CULLER, 2002).

Os sensores e atuadores são usados como interfaces do nodo para o mundo físico. Um sensor é usado para observar o ambiente físico ao redor do sensor e os atuadores são usados para alterar o ambiente. Os tipos de sensores podem variar desde simples sensores de temperatura até sensores de imagem de alta resolução (YIFENG; XIAOFENG; MING, 2007).

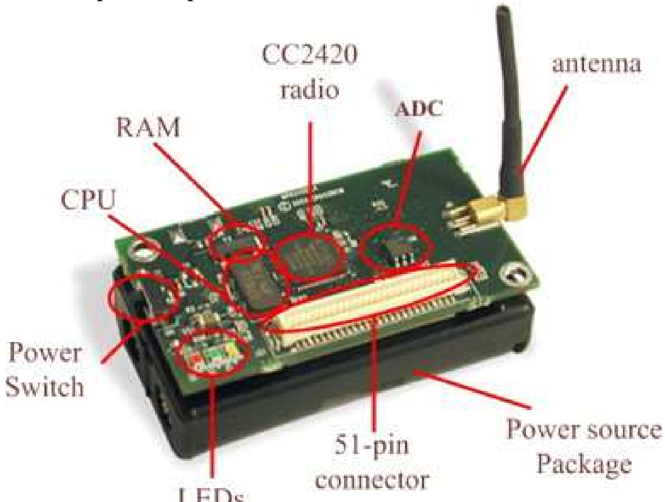
O microcontrolador é a unidade central no nodo, e é responsável por administrar os sensores e atuadores, decidir como e quando observar ou controlar o ambiente, e processar a informação obtida. O microcontrolador, normalmente, também possui memória interna ou externa possibilitando o armazenamento das informações coletadas e/ou processadas. Por fim, o microcontrolador acessa o dispositivo de comunicação sem fio e decide quando e para onde enviar as informações (WARNEKE et al., 2001). Também pode receber dados de outros nodos e decidir o que fazer com base no conteúdo dos dados recebidos.

Adicionalmente, um nodo sensor precisa de uma fonte de energia para se manter em modo de operação. Esta energia pode ser fornecida por baterias, ou pode ser coletada do meio ambiente. O modo mais comum para efetuar a coleta de energia é através da utilização de células solares. Outras fontes de captação de energia podem ser de vibração, mudanças de temperatura e fluxo de água ou ar (LI et al., 2008). A energia coletada, e não utilizada imediatamente, é armazenada em baterias recarregáveis. Para nodos sensores, energia é geralmente um recurso bastante limitado. Portanto o microcontrolador deve realizar atividades em ciclos, como uma estratégia para maximizar a operação do nodo sensor (SHARMA; BANERJEE; SIRCAR, 2008).

Por último, mas não menos importante, um nodo sensor sem fio também inclui um transceptor (transmissor/receptor). Estes transceptores podem ser bastante simples, com funções apenas de envio e recepção de bits individuais de informações para a rede, ou até transceptores mais avançados, que lidam com tarefas da camada de enlace. Isso pode incluir empacotar informação em pacotes, controlar o acesso ao meio através do Medium Access Control (MAC) e criptografar e descriptografar dados (FANG; LIN, 2006). O transceptor geralmente é o componente que mais consome energia em um nodo sensor sem fio. A maioria dos transceptores usados em redes de sensores sem fio operam nas faixas de rádio industriais, científicas e médicas chamadas bandas ISM (*Industrial, Scientific and Medical radio bands*). Algumas das frequências mais comuns são 433 MHz (Europa), 868 MHz (Europa), 915 MHz (EUA e Austrália), e 2,45 GHz (mundial) (SMOLNIKAR et al., 2010).

A Figura 2 ilustra os principais componentes de um nodo sensor sem fio. Este *mote* possui um microcontrolador, um transceptor sem fio, fonte de energia e periféricos de suporte para o microcontrolador.

Figura 2 – Principais componentes de um nodo sensor sem fio.




Adaptado de (HILL; CULLER, 2002)

### 2.1.1 Evolução da RSSF

No final da década de 1990 pesquisadores em MEMS (*Micro Electrical Mechanical Systems*) da Universidade de Berkeley cunharam

o termo *smart dust*, em tradução literal, "poeira inteligente" (WARNEKE et al., 2001). Esta ideia consiste na visão da miniaturização de dispositivos eletrônicos, possibilitando criar um nodo sensor sem fio do tamanho de um milímetro cúbico contendo sensores, microcontrolador, transceptor sem fio, célula solar, e bateria. Esses nodos do "tamanho de poeira" podem ser distribuídos no ambiente para coletar, processar e comunicar informações. O termo "poeira inteligente" ainda é usado quando se fala em miniaturização do tamanho de nodos sensores sem fio (HAJDAREVIC; KURTANOVIC, 2011; LEE et al., 2012).

No inicio do século 21 surgiu a primeira plataforma de redes de sensores sem fio realizada por um grupo de pesquisadores da Universidade da Califórnia em Berkeley. Esta plataforma, desenvolvida utilizando microcontroladores e transceptores de radio encontrados na indústria, ficou denominada como *Berkeley motes* (HILL et al., 2000). O mesmo grupo também iniciou o desenvolvimento de um sistema operacional para redes de sensores sem fio chamado TinyOS (LEVIS et al., 2005), que tem sido até hoje um dos sistemas operacionais mais utilizados para redes de sensores sem fio. Outro sistema operacional popular é o ContikiOS, desenvolvido por pesquisadores do Instituto Sueco de Ciencia da Computação (SICS) (DUNKELS; GRONVALL; VOIGT, 2004). Sendo este o primeiro sistema operacional para redes de sensores a prover comunicação utilizando o Protocolo de Internet (IP). O ContikiOS é *open source* e escrito em C, tendo sua primeira versão lançada em 2003.

Em 2003, o Instituto de Engenheiros Eletricistas e Eletrônicos (IEEE) lançou um padrão de transceptores chamado 802.15.4 (HOWITT; GUTIERREZ, 2003). Este padrão define a camada física (PHY) e o controle de acesso ao meio (MAC) de transceptores para redes de sensores sem fio. O lançamento deste padrão foi um grande marco para a aceitação das RSSFs na indústria. Até o momento não existia um padrão global para transceptores de rádio de baixo consumo. Este padrão tem sido muito bem sucedido e a maioria das plataformas de rede de sensores tem utilizado transceptores com o padrão IEEE 802.15.4 (JOHNSON et al., 2009).

Atualmente a tendência vai na direção do uso do Protocolo de Internet (IP) como o protocolo de rede para RSSFs (ZHANG et al., 2011). Os termos "Internet das coisas" (do Inglês, *Internet of things*) e "Internet sem fio embarcada" também têm sido largamente discutidos (HU, 2011; MA, 2011). Recentemente o grupo 6LoWPAN da comunidade *Internet Engineering Task Force* (IETF) definiu uma série

de padrões para habilitar o uso de IPv6 em RSSFs de baixo consumo e taxa de transmissão (BRUNO et al., 2011).

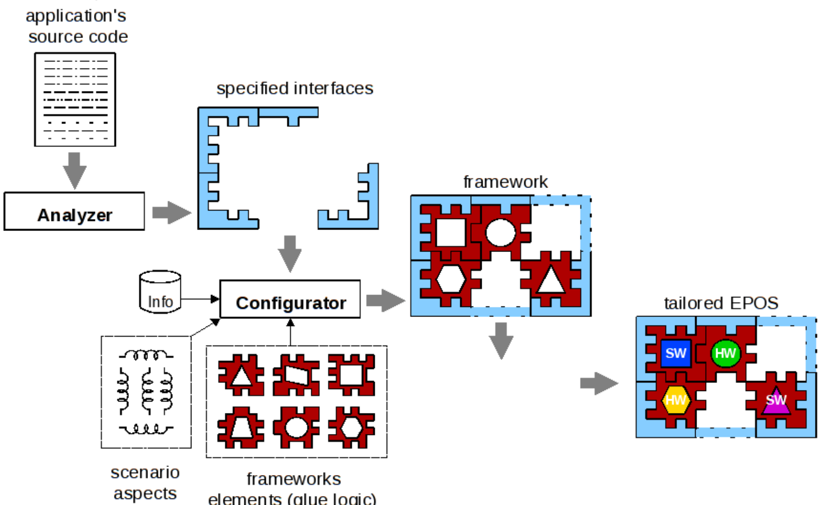
### 2.1.2 O Sistema Operacional EPOS

O sistema operacional EPOS (*Embedded Parallel Operating System*) (FRÖHLICH, 2001) é um sistema operacional voltado para aplicações embarcadas. O EPOS foi desenvolvido em C++ utilizando-se de técnicas como orientação a aspectos e metaprogramação estática para alcançar um alto grau de configurabilidade e ao mesmo tempo atender aos rigorosos requisitos associados à área de sistemas embarcados, tais como desempenho e tamanho de código. Várias aplicações interessantes e inovadoras demonstram a aplicabilidade do EPOS, por exemplo, a aplicação em redes de sensores sem fio envolvendo a plataforma EPOSMote (FRÖHLICH; WANNER, 2008).

O EPOS utiliza a metodologia de desenvolvimento de sistemas embarcados orientados à aplicação, guiando o desenvolvimento conjunto de componentes de software e hardware adaptáveis aos requisitos particulares de cada aplicação. De fato, pode-se dizer que o EPOS é um repositório de componentes de software e hardware, juntamente com ferramentas capazes de integrar tais componentes e produzir um ambiente de suporte a execução.

O sistema assim gerado possui somente os componentes julgados necessários para a aplicação. A Figura 3 detalha o processo de integração e configuração dos componentes e aspectos que podem fazer parte de uma imagem de sistema EPOS. Também se destaca o fato de que uma aplicação que utiliza o EPOS pode ser implementada como um SoC (*System on a Chip*), utilizando-se de componentes implementados em hardware e em software (FRÖHLICH, 2011).

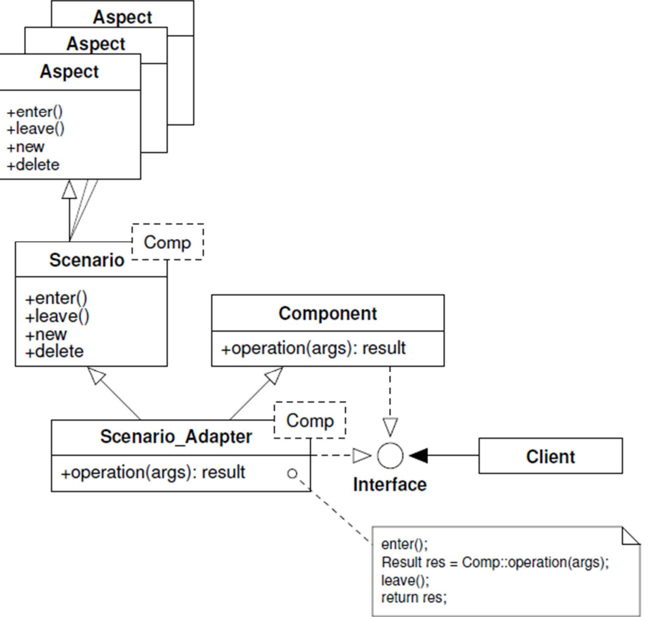
Figura 3 - Processo de integração e adaptação dos componentes para produzir uma imagem de sistema EPOS




Fonte: (FRÖHLICH, 2011).

Os conceitos fundamentais de sistemas operacionais (e.g., escalonador, *thread* e semáforo) são descritos como famílias de abstrações. As técnicas de programação orientada a aspectos são utilizadas para adaptar essas abstrações às peculiaridades do ambiente em que o sistema irá executar (convenção de chamadas, número de registradores, ordem de bytes, entre outras) (FRÖHLICH, 2011). A entidade responsável por realizar essa adaptação da abstração ao cenário de execução é convenientemente chamada de Adaptador de Cenário. A Figura 4 demonstra a relação entre o adaptador de cenário, o cenário, os aspectos, os componentes do sistema operacional e a aplicação em si.

Figura 4- Diagrama UML ilustrando as relações entre os principais conceitos do sistema EPOS




Fonte: (FRÖHLICH, 2011).

## 2.2 TECNOLOGIAS UTILIZADAS

Esta seção visa apresentar as tecnologias envolvidas no desenvolvimento do trabalho.

### 2.2.1 EPOSMote

Os nodos da RSSF são baseados na plataforma EPOSMoteII, um projeto de hardware aberto (LISHA, 2010). O seu principal objetivo é disponibilizar uma plataforma de hardware para permitir pesquisas em captação de energia, biointegração e sensores baseados em MEMS. A plataforma EPOSMoteII tem foco na modularização, sendo assim composta por módulos intercambiáveis para cada função. A Figura 5 ilustra o kit de desenvolvimento do EposMoteII.

Figura 5 – Kit de desenvolvimento do EPOSMoteII

Adaptado de (LISHA, 2010)

O hardware deste mote foi projetado como uma arquitetura de camadas composta por um módulo principal, um módulo de sensoriamento, e um módulo de energia. O modulo principal é responsável pelo processamento, armazenamento e comunicação. O modelo utilizado neste trabalho possui um processador ARM de 32 bits, 128 kB de memoria flash, 96 kB de memória RAM, e transceptor de acordo com a norma IEEE 802.15.4. O módulo de sensoriamento contém alguns sensores (i.e., temperatura e acelerômetro), LEDs, chaves e uma porta micro USB (que também pode ser utilizada como fonte de energia).

Vale destacar a funcionalidade de sincronização de relógios implementada pela plataforma, que foi decisiva para o funcionamento do sistema, uma vez que existe a ocorrência de uma instabilidade da frequência do sinal de relógio no processador utilizado pela plataforma. Esta instabilidade é prevista no manual de referência do microcontrolado.

## 2.2.2 Terminal Serial GPRS

O Terminal Serial GPRS (LISHA, 2010) é uma solução para telemetria de equipamentos remotos que utiliza a rede GPRS, também utilizada para comunicação de dados em celulares GSM. O Terminal Serial GPRS (TSG) é um equipamento composto por um modem

GSM/GPRS incialmente idealizado para conectar um dispositivo com uma porta serial RS-232C a um servidor central.

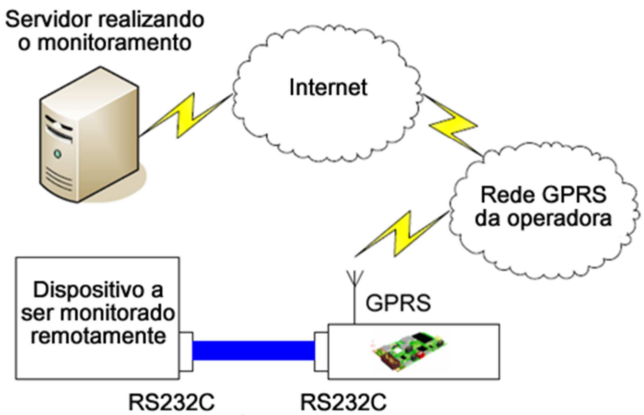
O GPRS é uma ótima forma de comunicação para telemetria e seu uso tem crescido muito nos últimos anos nesse nicho de mercado. A concorrência entre as operadoras de telecomunicações GSM propicia uma queda de preços para transmissão de dados utilizando a rede GPRS. Em redes GPRS a conexão de dados é feita sem necessidade de se estabelecer um circuito telefônico, o que permite a tarifação por utilização/tráfego e não por tempo de conexão, permitindo que o serviço esteja sempre disponível para o usuário.

Um dos grandes benefícios do GPRS é o seu baixo custo de assinatura e dado transmitido se comparado ao custo de uma linha discada (Assinatura + Chamadas + Interurbanos). Outro ponto muito importante é que esta tecnologia pode ser usada onde não há linhas telefônicas disponíveis, mas há cobertura de uma rede GSM/GPRS.

O TSG possui um hardware baseado em um processador ATMega128L, 128kB de memória flash, 5kB de memória RAM, e um modem Motorola G24. Baseado na especificação dos principais componentes, este dispositivo pode trabalhar corretamente a uma condição de temperatura ambiente ente -20°C a +60ºC, utilizando 3,3V e 2,6 mA em modo *sleep*, totalizando 8,6mW de potencia.

Depois de configurado, o TSG permite uma conexão serial transparente através da rede GPRS. Para isso, é necessário que o servidor disponibilize um canal de comunicação entre processos, ou seja, um *socket* conectado a uma porta para receber os dados e então possibilitar o acesso ao equipamento através de uma interface serial virtual. A Figura 6 ilustra os elementos do sistema.

Figura 6 – Visão geral de funcionamento do Terminal Serial GPRS




Adaptado de (LISHA, 2010)

### 2.2.3 Sensores

Nesta seção são apresentados os sensores utilizados para medir variáveis ambientais, destacando-se o princípio de funcionamento e comportamento de cada um. Para definir os tipos de sensores a serem utilizados neste trabalho foram consideradas algumas definições da comunidade científica baseadas em *deployments* realizados (BASHA; RAVELA, 2008; INGELREST et al., 2010).

#### 2.2.3.1 Nível do rio

Para se adquirir dados de nível de rios existem várias soluções, desde os métodos mais simples, tais como sensores de pressão ou sensores resistivos, até métodos que medem à distancia, como sonares e câmeras. Sensores resistivos de contato são utilizados em sequencia, como uma régua, com isso quando uma resistência entra em contato com a água a mesma fecha uma chave que envia um sinal ao sistema, correspondendo a um nível de água. Entretanto este tipo de sensor está sujeito a corrosão, não sendo confiável para *deployments* de longa duração (BASHA; RAVELA, 2008).

Sonares, normalmente utilizados em pontes, medem a distância relativa entre o sensor e o nível da água, desta forma calculando o nível da água. Porém, este tipo de solução pode não funcionar muito bem quando há ocorrência de fortes ventos, que por sua vez, podem fazer com que o sensor perca sua calibração. Existem pesquisas sobre o uso

de câmeras em RSSFs (YIFENG; XIAOFENG; MING, 2007), entretanto o uso de câmeras precisa ser avaliado devido as limitações do *mote*, como capacidade de processamento e custos de transmissão de imagem.

Os sensores de nível por coluna de fluido operam pelo princípio de Pascal ($P = \gamma.h$), onde $\gamma$ é o peso específico do líquido, através de um elemento piezoresistivo que converte a pressão aplicada pela coluna de fluido em sinal elétrico. Para utilizar esta solução normalmente cava-se um fosso ao lado do rio, onde é inserido um vaso comunicante, como pode ser visto na Figura 7, com a finalidade de evitar a interação de objetos que podem vir a estar no rio.

Figura 7 - Princípio de funcionamento do sensor de pressão por coluna da água

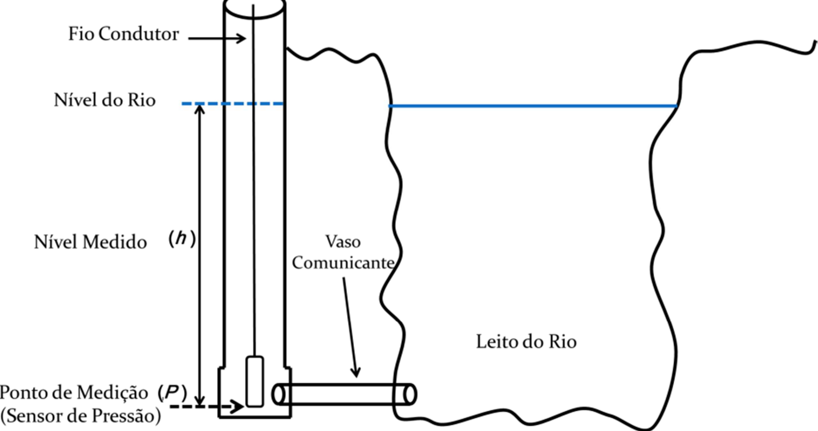
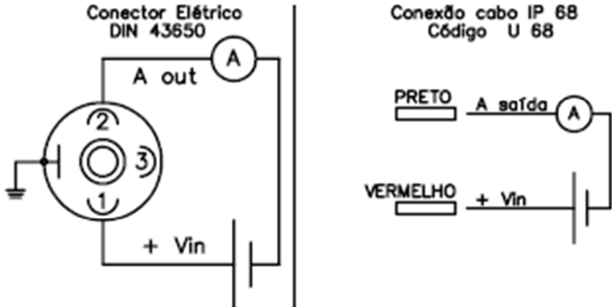



Figura do Autor

O sensor utilizado foi o ACROSS TP-ST18-SUB, este sensor utiliza um cabo especial com compensação de pressão atmosférica, podendo medir níveis entre 0 e 150 cm, que podem ser mensurados através de um sinal de saída de 4 a 20 mA (vide Figura 8), podendo operar a temperaturas líquidas entre 0 a 70ºC.

Figura 8 - Esquemático de conexão e interface de leitura do sensor de nível



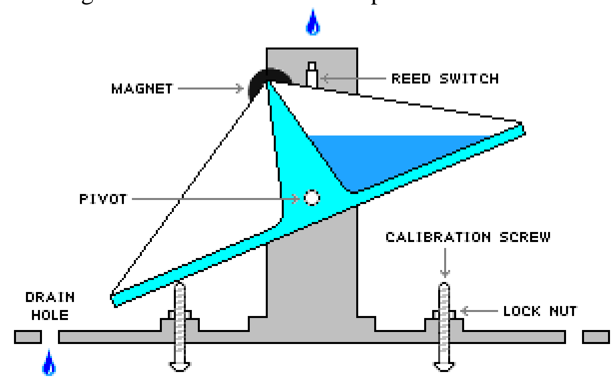

Adaptado de manual do sensor ACROSS TP-ST18-SUB

2.2.3.2 Pluviosidade

Para efetuar o sensoriamento da pluviosidade de determinado local utiliza-se o pluviômetro. O pluviômetro é um instrumento meteorológico utilizado para recolher e medir a quantidade de líquidos ou sólidos como chuva ou granizo (e também neve para regiões onde ocorre). Como o equipamento mensura a quantidade de chuva que precipita, é elementar para estudos meteorológicos e hidrológicos em conjunto com o sensor de temperatura. Entre os tipos de sensores os mais populares são: infravermelho, piezoelétrico e mecânico (báscula), sendo este último o mais utilizado, devido a facilidade de construção e baixo custo.

O princípio de funcionamento do pluviômetro de báscula baseia-se em uma "gangorra" com dois reservatórios idênticos, um de cada lado, correspondentes a um determinado volume de água. Quando um dos reservatórios enche, o peso da água, pelo principio da gravidade, faz com que a gangorra incline para o outro lado. Quando isso ocorre um ímã se move junto com a "gangorra", que passa próximo a um *reed switch*, chave que se fecha ao receber uma excitação magnética, gerando um sinal elétrico. A Figura 9 ilustra o funcionamento do sensor.

Figura 9 - Diagrama de funcionamento do pluviômetro de báscula




Adaptado de (WEATHERSHACK, 2012)

O coletor de chuva utilizado foi o pluviômetro de báscula da empresa DavisNet (DAVISNET, 2012) projetado para atender as diretrizes da Organização Mundial de Meteorologia (WMO). Este sensor foi utilizado para mensurar a precipitação diária e acumulada, bem como a taxa de precipitação, com precisão de leitura de 0,2 milímetros de chuva, ou seja, cada oscilação da báscula corresponde a 0,2 mm.

### 2.2.3.3 Temperatura

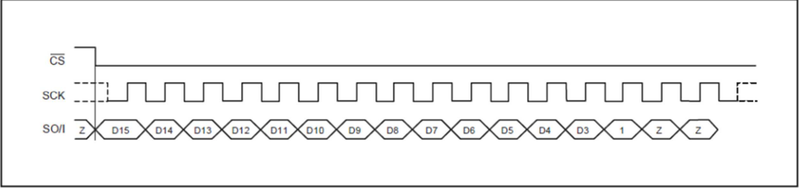
Para medir a temperatura ambiente, utilizou-se o circuito integrado TMP123 (TEXAS INTRUMENTS, 2012). Este sensor é capaz de medir temperaturas com uma precisão de 1,5 ºC e uma resolução de 0,0625 ºC dentro de uma variação de –25 ºC a +85 ºC. Com uma baixa drenagem de corrente de 50 µA e tensão de alimentação entre 2,7 V e 5,5 V, o TMP123 torna-se um excelente componente para aplicações de baixo consumo de energia. No TMP123, 16 bits são disponibilizados contendo o valor da temperatura adquirida, como pode ser visto na Tabela 1.

Tabela 1- Formato dos dados de leitura do sensor TMP123

| Temperatura (ºC) | Saída digital (binário) | Hexadecimal |
|---|---|---|
| 150 | 0100 1011 0000 0000 | 4B00 |
| 125 | 0011 1110 1000 0000 | 3E80 |
| 25 | 0000 1100 1000 0000 | 0C80 |
| 0,0625 | 0000 0000 0000 1000 | 0008 |
| 0 | 0000 0000 0000 0000 | 0000 |
| -0,0625 | 1111 1111 1111 1000 | FFF8 |
| -25 | 1111 0011 1000 0000 | F380 |

A Figura 10 apresenta o procedimento de leitura da temperatura através do protocolo de comunicação SPI (*Serial Peripheral Interface*). Nele, o microcontrolador gera o sinal de clock SCK, fazendo com que o dispositivo selecionado (TMP123) disponibilize as informações no canal SO/I, iniciado pelo bit D15 (o mais significativo) e finalizado com o bit D3 (menos significativo). Os três bits finais (1, Z e Z) servem apenas para completar a palavra de dados de 16 bits. O dado então é recebido pelo microcontrolador e montado no pacote de dados para ser transmitido.

Figura 10 - Procedimento de leitura através da SPI



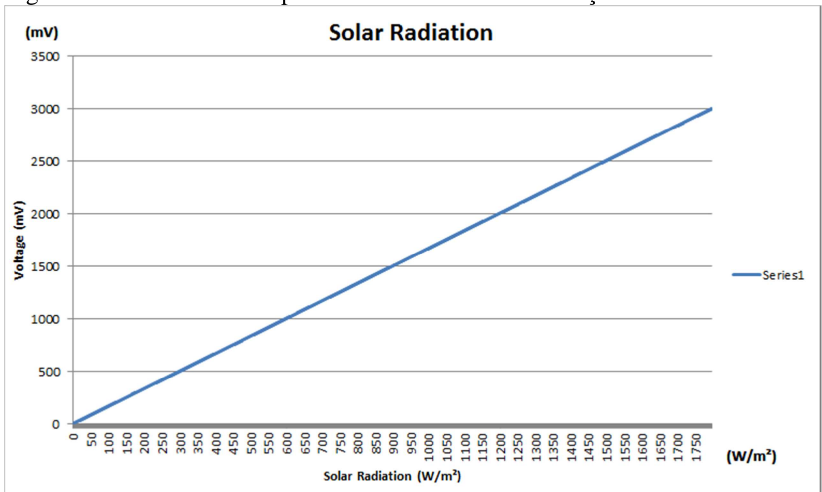

Adaptado de manual de referencia do TMP123

### 2.2.3.4 Radiação solar

Outra variável de interesse deste trabalho é o nível de radiação, tanto solar quanto ultravioleta, que é importante para medir o potencial solar energético de determinada região. Radiação solar é a designação dada à energia radiante emitida pelo Sol, em particular aquela que é transmitida sob a forma de radiação eletromagnética. Cerca de metade desta energia é emitida como luz visível na faixa de frequência mais alta do espectro eletromagnético e o restante na faixa do infravermelho e como radiação ultravioleta. A radiação solar fornece anualmente para a atmosfera terrestre $1{,}5 * 10^{18}$ kW/h de energia, sendo assim o principal responsável pela dinâmica da atmosfera terrestre e pelas características climáticas do planeta (DOUGLAS, 1997).

O sensor utilizado para medir a radiação solar é baseado em um fotodiodo que detecta comprimentos de onda entre 300 e 1100 nanômetros, produzido pela DavisNet. Este sensor requer uma fonte de alimentação de 3,3 V e drena aproximadamente 1 mA, oferecendo um sinal de saída analógico que varia de 0 a 3 V. Na Figura 11 podemos notar o comportamento linear do sensor.

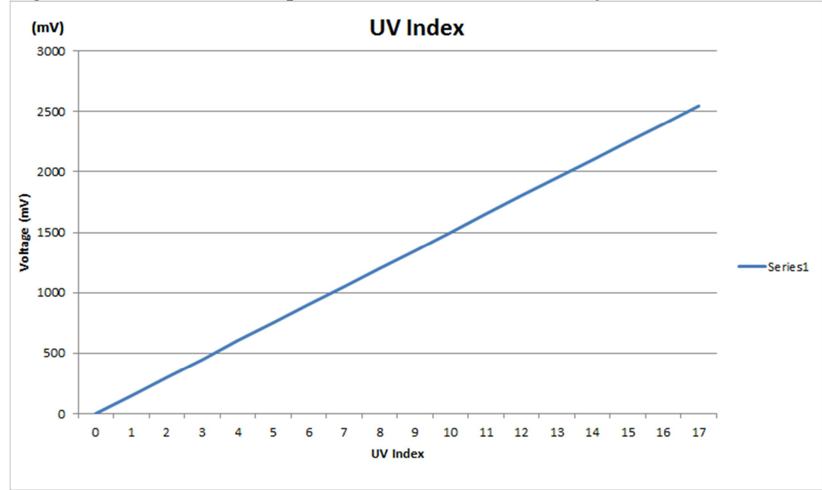
Figura 11 - Gráfico de comportamento do sensor de radiação solar




### 2.2.3.5 Radiação ultravioleta

O sensor de radiação ultravioleta, também produzido pela empresa DavisNet, mede comprimentos de onda entre 280 e 360 nanômetros que compreendem a porção do espectro ultravioleta recebida diariamente na superfície terrestre. Este sensor requer uma fonte de alimentação de 3,3 V e drena aproximadamente 2,4 mA, oferecendo um sinal de saída analógico que varia de 0 a 2,5 V. Na Figura 12 podemos observar o comportamento sensor.

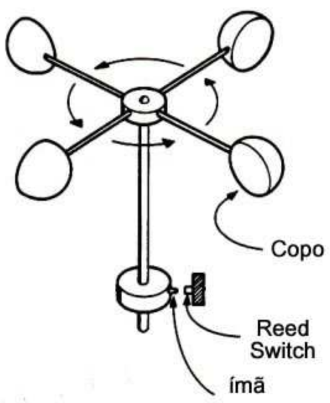
Figura 12 - Gráfico de comportamento do sensor de radiação ultravioleta




### 2.2.3.6 Velocidade e direção do vento

O sensor para medir a velocidade do vento, mais conhecido como anemômetro, utiliza pequenos copos dispostos na vertical que rotacionam na existência de um fluxo de vento, como pode ser visto na Figura 13. Este sensor possui um *reed switch* que a cada revolução realizada fecha um contato, gerando uma interrupção no microcontrolador.

Figura 13 - Diagrama de funcionamento do sensor de velocidade do vento



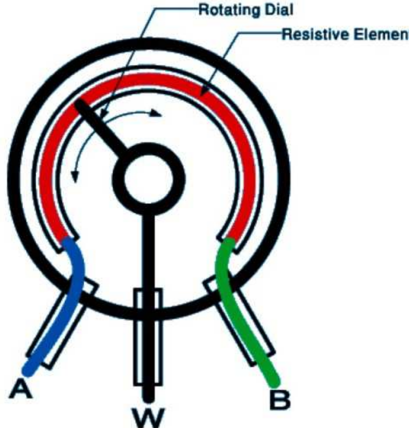

Adaptado de (MIAJAS, 2012)

O número de revoluções $n$ em um determinado período de tempo $t$ indica a velocidade atingida pelo vento, como mostra a equação:

$$V = n\frac{2\pi r}{t}$$

Para medir a direção do vento há uma aleta que ao interagir com o vento gira um potenciômetro que, por sua vez, altera a resistência relativa, logo alterando o valor de tensão entre A e W, permitindo definir o ângulo de direção, como ilustra a Figura 14. O ângulo correspondente é relativo ao Norte (0º), ou seja, 90º corresponde ao leste, 180º corresponde ao sul, e assim sucessivamente.

Figura 14 - Funcionamento do sensor de direção do vento




Adaptado de (WIND 101 BLOG, 2012)

Baseado nestes sensores, decidiu-se utilizar a estação meteorológica da empresa DavisNet e o sensor de coluna da água da empresa ACROSS, devido a disponibilidade de ambos além de serem largamente utilizados por hidrólogos e sistemas de medições climáticas atuais. A Tabela 2 apresenta os sensores disponíveis para o desenvolvimento do sistema.

Tabela 2 - Sensores disponíveis para o desenvolvimento do sistema

| Variável | Sensor | Variação | Precisão |
|---|---|---|---|
| Temperatura | TMP123 (Texas Instruments) | -25–85ºC | ±0,0625 ºC |
| Pluviosidade | Pluviômetro de Báscula (Davis) | 0–∞ mm | ±1 mm |
| Radiação Solar | Radiação Solar (Davis) | 0–1800 W/m² | ±90 W/m² |
| Radiação Ultravioleta | Radiação Ultravioleta (Davis) | 0-16 (Índice) | ±0.1(Índice) |
| Nível de Fluido | TPST18SUB (Across) | 0–1.5 mCA | - |
| Direção do Vento | Anemômetro (Davis) | 0–360º | ±7º |
| Velocidade do Vento | Anemômetro (Davis) | 1.5–79 m/s | ±1.5 m/s |

## 2.3 CONSIDERAÇÕES SOBRE O CAPÍTULO

Este capítulo apresentou brevemente a evolução das Redes de Sensores Sem Fio (RSSFs) e as tecnologias utilizadas como base para o desenvolvimento do trabalho, assim como os principais componentes de uma RSSF. Entre estes componentes destacou-se o sistema operacional e suas características, onde explicamos o EPOS, sistema operacional utilizado neste trabalho.

O próximo capítulo apresenta os trabalhos relacionados explicitando suas principais características.

# 3 TRABALHOS RELACIONADOS

Nos últimos anos, com o advento das RSSFs, o monitoramento ambiental tornou-se um campo de aplicação importante, onde atualmente tem-se observado um aumento nas pesquisas na área de sensoriamento remoto, como pode ser visto na Figura 15. Uma vez que "monitoramento de rios" é um tipo de monitoramento ambiental, nesta seção serão discutidos trabalhos que objetivam o monitoramento ambiental e seus aspectos.

Figura 15 - Resultado de pesquisa do termo "Remote Sensing" no ieeexplore

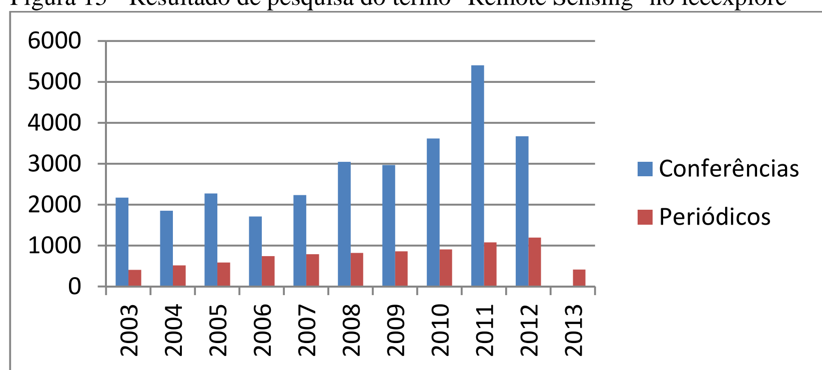
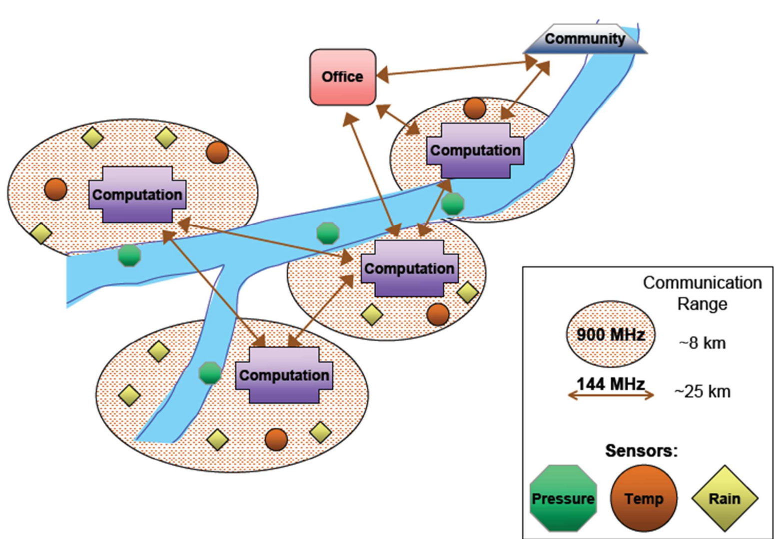



Figura do Autor

## 3.1 SISTEMA DE ALERTA TEMPRANA

Um dos primeiros *deployments* de RSSFs para monitoramento de rios que pode se encontrar na literatura foi realizado em Honduras e publicado pelo MIT (*Massachusetts Institute of Technology*) em 2007 (BASHA; RUS, 2007). O artigo propõe um sistema baseado em RSSF para países em desenvolvimento chamado *Sistema de Alerta Temprana* (SAT). A Figura 16 ilustra a arquitetura do sistema, que é composta por: nodos sensores, onde os dados são coletados e transmitidos; nodos de processamento, onde um algoritmo embarcado processa os dados coletados a fim de prever um possível evento de inundação; e nodos informativos instalados em comunidades ribeirinhas, com o objetivo de alertar sobre o perigo.

Figura 16 – Visão geral da arquitetura do *Sistema de Alerta Temprana*




Adaptado de (BASHA; RAVELA, 2008)

O sistema possui duas camadas de comunicação, onde a primeira camada possui os nodos sensores que se comunicam com um nodo mestre. Este nodo mestre faz parte da segunda camada e é responsável por capturar os dados da primeira camada, processá-los e enviar para o nodo informativo. A primeira camada é uma rede *single-hop*, ou seja, os nodos sensores se comunicam somente com o nodo mestre. Já a segunda camada possui *multi-hop*, ou seja, o nodo é capaz de se comunicar com todos os nodos vizinhos daquela camada. Este trabalho também discute tipos adequados de sensores para monitoramento de nível de rios, enfatizando os problemas encontrados com sensores resistivos, devido a corrosão, e sensores ultrassônicos, devido ao problema de fixação no ambiente na ocorrência de fortes rajadas de vento. Sendo assim, baseado em vários *deployments*, Basha et al.(2008) concluem que atualmente a maneira mais eficaz de coletar dados de nível de rios é através de sensores de pressão instalados no fundo do rio.

## 3.2 REDE

Hughes *et al.*(2011) apresentam uma plataforma de *middleware* para apoiar o monitoramento de rios chamado DisSeNT. O trabalho também apresenta uma plataforma de RSSF chamada REDE (Redes de

sensores sem fio para Detectar Enchentes) baseada em nodos *Sun Spot* (ORACLE, 2011) integrados com sensores de pressão, metano e condutividade. A principal contribuição do trabalho é uma arquitetura de software para facilitar o desenvolvimento e gerencia de aplicações baseadas em RSSF. O *middleware* proposto possui um modelo de componente reconfigurável em tempo de execução chamado LooCI (*Loosely-coupled Component Infrastructure*), uma arquitetura de gerenciamento baseada em políticas chamado PMA (*Policy-based Management Architecture*), e um mecanismo para composição e especificação da aplicação baseado nos requisitos de *deployment* chamado QARI (*Quality Aware Reconfiguration Infrastructure*). A Figura 17 ilustra como os componentes descritos acima podem ser conectados por desenvolvedores de software no desenvolvimento de aplicações baseadas em RSSF. O trabalho comenta os resultados baseado em um *deployment* realizado em um rio na cidade de São Carlos-SP.

Figura 17 – Visão geral dos componentes de software do middleware DisSeNT

Adaptado de (HUGHES et al., 2011)

## 3.3 SENSORSCOPE

Ingelrest *et al.*(2010) propõem o SensorScope, uma plataforma de RSSF baseada em nodos TinyNode (TINYNODE, 2012) com a

finalidade de coletar dados meteorológicos. O trabalho apresenta o desenvolvimento de uma pilha própria de comunicação com recursos de rede *multihop* e uma camada MAC sincronizada por ciclos do microcontrolador a fim de reduzir o consumo geral de energia. O trabalho também descreve o hardware e os sensores usados, e mostra os resultados experimentais com base em vários *deployments* em diferentes regiões, apontando lições aprendidas em relação a problemas com a resiliência do hardware ao ambiente, a obtenção de dados sobre a saúde da rede, o ciclo de vida da aplicação, e a calibração dos sensores. A Figura 18 mostra a visão geral do sistema, vale ressaltar o nodo sincronizador que possui um *gateway* GPRS e é responsável por enviar os dados coletados para o servidor, assim como possibilitar o gerenciamento remoto da RSSF.

Figura 18 - Visão geral da arquitetura do SensorScope



Adaptado de (INGELREST et al., 2010)

## 3.4 ECOLOGY-HIDROLOGY

Zhang *et al*. (2012) propõem um framework chamado *Ecology-Hydrology Wireless Sensor Network*, com o objetivo de monitorar uma bacia hidrográfica. O artigo descreve uma arquitetura de sistema para fornecer dados confiáveis e relatórios dos problemas enfrentados devido ao ambiente severo (i.e., alta umidade e baixas temperaturas). O trabalho também comenta que o código da aplicação embarcada na RSSF deve ser o mais simples possível, evitando interações entre camadas de software, a fim de reduzir o consumo de energia e uso de memória.

Figura 19 - Visão geral da arquitetura do framework *Ecology-Hidrology Wireless Sensor Network*

Adaptado de (ZHANG et al., 2012)

Com base nos trabalhos relacionados podemos destacar as principais características de sistemas para monitoramento ambiental, conforme pode ser observado na Tabela 3.

Tabela 3 – Características dos trabalhos relacionados

| **Trabalho** | **Gerência remota** | **Topologia** | **Frequencia de operação** | **Alcance** | **Camadas de rede** |
|---|---|---|---|---|---|
| SAT | Não | Single-hop/ Multihop | 144 Mhz/900 Mhz | 25 Km/8 Km | 2 |
| REDE | Não | Multihop | 2,4 Ghz | 50 m | 1 |
| Sensor Scope | Sim | Multihop | 868 Mhz/GPRS | 1,2 Km | 2 |
| Eco-Hydrology | Sim | Multihop | 2,4 Ghz/GPRS | 1,05 Km | 2 |
| RiverSense | Sim | Single-hop | 2,4 Ghz/GPRS | 80 m | 2 |

## 3.5 CONSIDERAÇÕES SOBRE O CAPÍTULO

Neste capítulo foram apresentados quatro projetos de sistemas para monitoramento ambiental e suas principais características. O capítulo de considerações finais desta dissertação retoma as

características identificadas nesta seção para que possam ser comparadas com o sistema proposto.

Baseado na pesquisa realizada chegou-se a conclusão sobre a viabilidade técnica para a proposta de um sistema que execute a coleta de dados climáticos relacionados ao nível de rios e, através de um sistema de monitoramento, disponibilize serviços de dados meteorológicos que possam ser utilizados por outros sistemas. O próximo capítulo apresenta a metodologia, a arquitetura e a implementação do sistema proposto.

# 4 SISTEMA PROPOSTO

Este capítulo descreve o sistema RiverSense. Inicialmente, é apresentada a metodologia utilizada para a concepção do sistema, após é descrito a organização e funcionamento da arquitetura e, por último, o projeto detalhado de cada um dos componentes implementados no protótipo.

## 4.1 METODOLOGIA

O desenvolvimento do protótipo do RiverSense foi dividido em duas etapas, sendo a primeira de Projeto/Análise, onde se gerou a documentação técnica, que serviu de suporte para a segunda etapa, a Codificação, onde o protótipo foi desenvolvido e avaliado.

Para a modelagem do protótipo (primeira etapa), foi utilizada a linguagem UML (*Unified Modeling Language*) (BOOCH; JACOBSON; RUMBAUGH, 1996) por se tratar de um padrão internacionalmente reconhecido e difundido.

Para codificar os diversos componentes do RiverSense Server foram utilizadas tecnologias que seguem os requisitos de portabilidade, utilizando para isso a linguagem Java (ORACLE, 2012a). Para o armazenamento de dados, foi utilizado o banco de dados MySQL (ORACLE, 2012b) por ser gratuito e de fácil uso. Já para a codificação da página utilizou-se das linguagens PHP (PHP, 2012) e JavaScript (FOUNDATION, 2012).

A fim de seguir os requisitos de adaptabilidade, onde os processamentos principais são organizados de forma a possibilitar alterações sem grande impacto para a aplicação, optou-se pelo uso de Web Services através do framework GlassFish (ORACLE, 2012c), atualmente um dos mais utilizados por ser de código aberto e devido a sua flexibilidade e facilidade na configuração e publicação de serviços.

Para a codificação da RSSF foi utilizada a linguagem C/C++ e o sistema operacional EPOS, específico para a plataforma EPOSMoteII. Já para o projeto de hardware para a interface dos sensores de cada nodo da RSSF utilizou-se a ferramenta Eagle, da empresa CadSoft (CADSOFT USA, 2012).

## 4.2 ARQUITETURA

O RiverSense é composto por duas partes: o RiverSense Server e o RiverSense RSSF, vide Figura 20. O RiverSense Server é formado

pelos módulos que deverão estar disponíveis no servidor para serem acessados pela RSSF. O RiverSense RSSF é a parte responsável por coletar dados do ambiente e enviá-los para a parte servidora.

Figura 20 – Visão geral da arquitetura do RiverSense

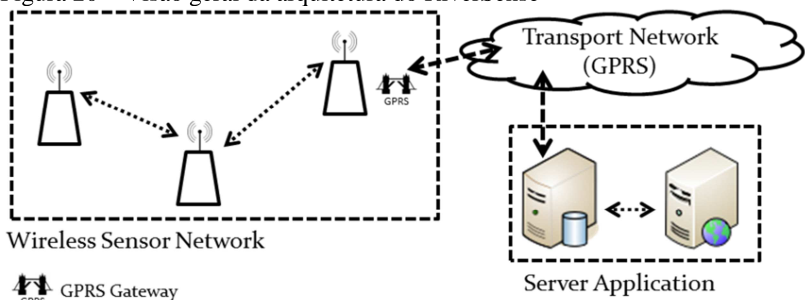



Figura do Autor

O projeto possui alguns requisitos, são eles:

- A precisão de medição do nível do rio deve ser igual ou menor que 10 cm;
- Alcançar um intervalo de entrega de dados igual ou menor que 5 minutos;
- Atualizar a cada 30 segundos os relógios dos nodos da RSSF, baseado em testes empíricos realizados pelo LISHA.

### 4.2.1 RiverSense Server

A arquitetura do RiverSense Server está organizada em três camadas e um banco de dados, conforme a Figura 21. A camada mais inferior é chamada de *RiverRepository*, que é responsável pelo acesso ao banco de dados. A camada intermediária é representada pelo *RiverManager*, que tem o propósito de atender as requisições realizadas pela RSSF. Já a camada superior é chamada de *RiverServices* e é responsável pela publicação dos dados armazenados e disponibilização de serviços para consulta de dados.

Figura 21 - Arquitetura do RiverSense Server

Figura do Autor

O *RiverManager* é responsável por intermediar a comunicação entre a RSSF e o RiverSense Server. Esta comunicação é realizada através de um protocolo baseado em mensagens de texto (ASCII), desenvolvido para suportar a troca de dados. A Figura 22 ilustra as mensagens esperadas pelo sistema.

Figura 22 - Tipos de mensagens trocadas entre a RSSF e o Servidor

| MessageSize | MessageErrors | GPRS ToA | Battery | Date/Hour |
|---|---|---|---|---|

(a) Mensagem do Nodo Sincronizador

| MessageSize | MessageErrors | Rain | Temperature | SolarRadiation |
|---|---|---|---|---|
| UVRadiation | WindSpeed | WindDirection | Battery | Date/Hour |

(b) Mensagem do Nodo Estação Meteorológica

| MessageSize | MessageErrors | RiverLevel | Battery | Date/Hour |
|---|---|---|---|---|

(c) Mensagem do Nodo Estação de Nível

Figura do Autor

Além de tratar da comunicação, o *RiverManager* também realiza o processo de armazenamento dos dados enviados pela RSSF no banco de dados através da camada *RiverRepository*. Esta camada tem a função de disponibilizar uma interface para o acesso ao banco de dados, previsto pela arquitetura do sistema. O banco de dados (Figura 23) está organizado, inicialmente, em três tabelas:

- Tabela Nodo Sincronizador: contém as informações do nodo responsável por sincronizar a RSSF, tais como tamanho da mensagem, número de tentativas para enviar uma mensagem, tempo de chegada de mensagens através do *gateway*, e nível de carga da bateria;

- Tabela Estação Meteorológica: contém as informações do nodo responsável por coletar dados meteorológicos, tais como temperatura, pluviosidade, radiação solar, radiação ultravioleta, direção e velocidade do vento, assim como informações relativas à RSSF, como tamanho da mensagem, número de tentativas para enviar uma mensagem, e nível de carga da bateria;
- Tabela Estação de Nível: contém as informações do nodo responsável por coletar dados sobre o nível do rio em determinado ponto, além do nível do rio são coletadas informações relativas à RSSF, como tamanho da mensagem, número de tentativas para enviar uma mensagem, e nível de carga da bateria.

Figura 23- Banco de dados do RiverSense Server

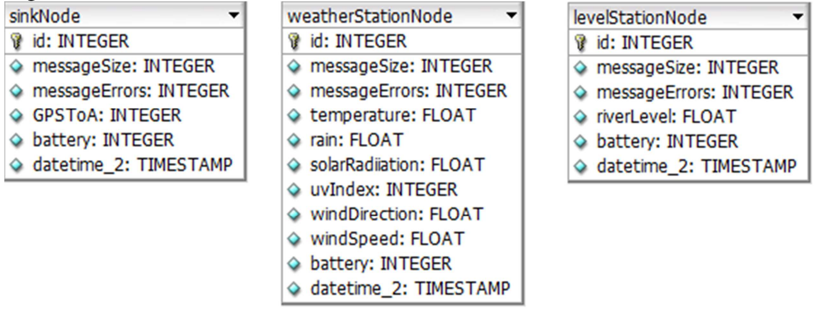



Figura do Autor

A camada denominada *RiverServices* é responsável pelo gerenciamento dos serviços disponibilizados pelo RiverSense Server. A arquitetura propõe que todas as operações de acesso e consulta sejam realizadas através de Web Services[1] (W3C, 2012).

Sendo assim, a finalidade da camada *RiverServices* é possibilitar que sistemas de terceiros tenham acesso às informações e utilizá-las conforme as suas necessidades. As principais funcionalidades da camada *RiverServices* são: (i) prover interfaces de consulta para

[1] Web Service é uma solução utilizada na integração de sistemas e na comunicação entre aplicações diferentes. Com esta tecnologia é possível que novas aplicações possam interagir com aquelas que já existem e que sistemas desenvolvidos em plataformas diferentes sejam compatíveis. Os Web Services são componentes que permitem às aplicações enviar e receber dados em formato XML. Cada aplicação pode ter a sua própria linguagem, que é traduzida para uma linguagem universal, o formato XML.

sistemas de terceiros; (ii) publicar os serviços disponíveis; (iii) permitir que novos serviços possam ser implementados a partir das necessidades de cada aplicação.

O RiverSense Server ainda disponibiliza uma página *web* que, através da camada *RiverServices*, consulta periodicamente os dados mais recentes de cada nodo da RSSF, e os disponibiliza para visualização através de gráficos interativos. Vale ressaltar que neste trabalho foram criados alguns serviços específicos que foram utilizados pela página *web* para possibilitar o acesso dos dados. Estes mesmos serviços podem ser utilizados por outros sistemas que desejam utilizar o RiverSense.

### 4.2.2 RiverSense RSSF

Esta parte do sistema é responsável por coletar dados ambientais através de sensores, processá-los e enviá-los através de mensagens, apresentadas na sessão anterior, para o RiverSense Server. A Figura 24 ilustra a comunicação e os dados coletados na RSSF.

Figura 24 - Elementos que compõem o RiverSense RSSF

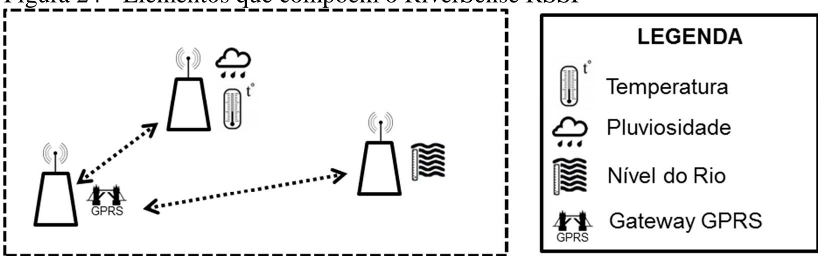



Figura do Autor

O Nodo Sincronizador é responsável por realizar a sincronia de relógio dos nodos da RSSF. Regularmente o nodo envia uma mensagem de requisição do tempo atual para o servidor através de um *gateway* GPRS. Já o Nodo Estação Meteorológica é responsável por realizar a coleta de dados de temperatura, pluviosidade, radiação solar e ultravioleta, além da direção e velocidade do vento. Por fim, o Nodo Estação de Nível é responsável por realizar a sincronia de relógio dos nodos da RSSF. Regularmente o nodo envia uma mensagem de requisição do tempo atual para o servidor através de um gateway GPRS.

Vale ressaltar que os nodos da RSSF trabalham em ciclos longos (i.e., cinco minutos) no intuito de economizar energia, uma vez que o RiverSense RSSF não possui requisitos de observação intensa.

## 4.3 IMPLEMENTAÇÃO

Esta seção apresenta uma visão geral da estrutura do protótipo do RiverSense, com todos os componentes de software e hardware desenvolvidos para dar suporte à execução dos experimentos. Inicialmente são descritos os principais aspectos da aplicação servidora e a seguir é descrito o funcionamento das principais partes da aplicação cliente. A Figura 25 mostra uma visão geral do relacionamento entre os diversos componentes implementados no protótipo.

Figura 25 – Diagrama de implantação do protótipo RiverSense

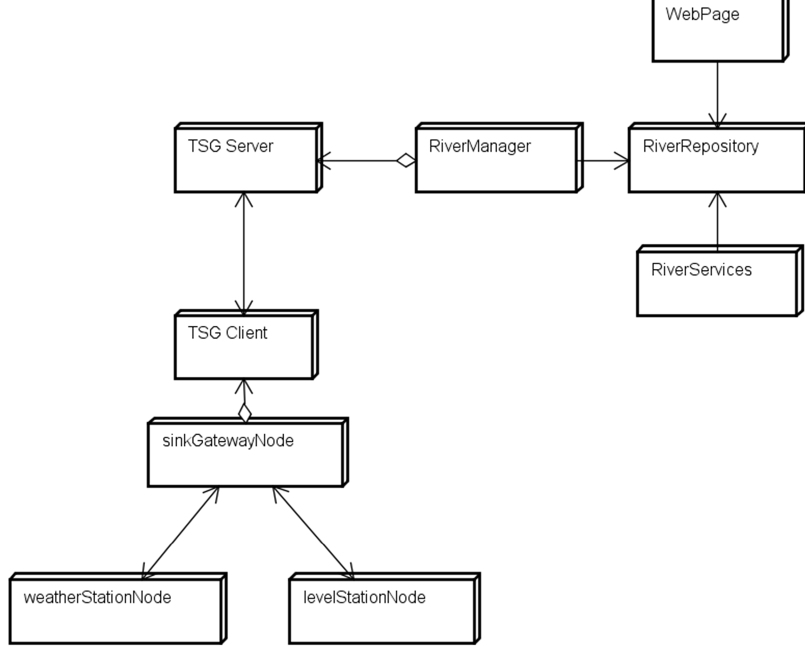



Figura do Autor

A seguir são listados os componentes e uma breve descrição sobre cada um:

- WebPage: Este componente é responsável por exibir os dados coletados em forma de gráficos.
- RiverServices: Este componente é responsável por disponibilizar serviços (Web Services) de consulta aos dados coletados para sistemas de terceiros.

- RiverRepository: Este componente é responsável por armazenar os dados coletados em um banco de dados MySQL.
- RiverManager: Este componente é responsável por receber e enviar mensagens da Rede de Sensores Sem Fio.
- TSGServer: Este componente é responsável por abstrair e disponibilizar o meio de comunicação (GPRS) com a RSSF para o componente RiverManager.
- TSGClient: Este componente é responsável por abstrair e disponibilizar o meio de comunicação (GPRS) com o servidor para o componente sinkGatewayNode.
- sinkGatewayNode: Este componente é responsável por realizar a comunicação entre a RSSF e o servidor, realizando envios de dados e requisições de relógio para sincronização da RSSF.
- weatherStationNode: Este componente é responsável por realizar a coleta de dados meteorológicos de determinado local e enviá-los ao componente sinkGatewayNode.
- levelStationNode: Este componente é responsável por realizar a coleta de dados do nível do rio em determinado local e enviá-los ao componente sinkGatewayNode.

### 4.3.1 RiverSense Server

Conforme colocado anteriormente, a aplicação servidora do Terminal Serial GPRS disponibiliza uma porta virtual serial para que outras aplicações acessem os dados de forma transparente. A Figura 26 ilustra a porta serial como interface de acesso ao TSG Server, assim como os demais componentes envolvidos no sistema, como o tratamento da mensagem recebida pelo *RiverManager* realizado pela classe *MessageParser*.

Figura 26 - Diagrama de componentes do RiverSense Server

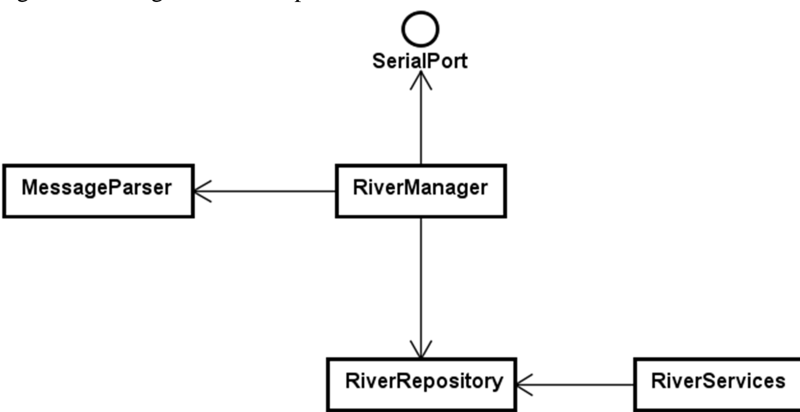



Figura do Autor

#### 4.3.1.1 Fluxo de mensagens

A Figura 27 traz o diagrama de sequência que demonstra como foi implementado o fluxo de mensagens por parte da aplicação servidora. Neste diagrama, a camada *RiverManager* fica aguardando por um evento na porta serial virtual, conectada ao TSG Server, através do método *SerialPortEventHandler*, quando ocorre um evento, é realizada uma verificação do tipo de mensagem recebida, se é uma mensagem de dados ambientais vindo de um nodo da RSSF ou se é uma mensagem de requisição de atualização de relógio.

Caso seja uma requisição de atualização de relógio, o método sendDateRequest, informando a hora e data atual no formato de *timestamp*, é chamado enviando os dados para a porta serial virtual do TSG Server, e assim finalizando um ciclo de fluxo de mensagem. Quando a mensagem recebida for uma mensagem de dados de um nodo, por haver nodos com diferentes sensores, cada nodo possui uma mensagem diferente, dessa forma o método parseMessage é chamado, afim de identificar de qual nodo vem a mensagem através do campo *MessageType* da mensagem. Uma vez identificado os dados são conferidos e então armazenados no banco de dados através da camada RiverRepository.

Figura 27 - Diagrama de sequência - Recebimento de mensagens

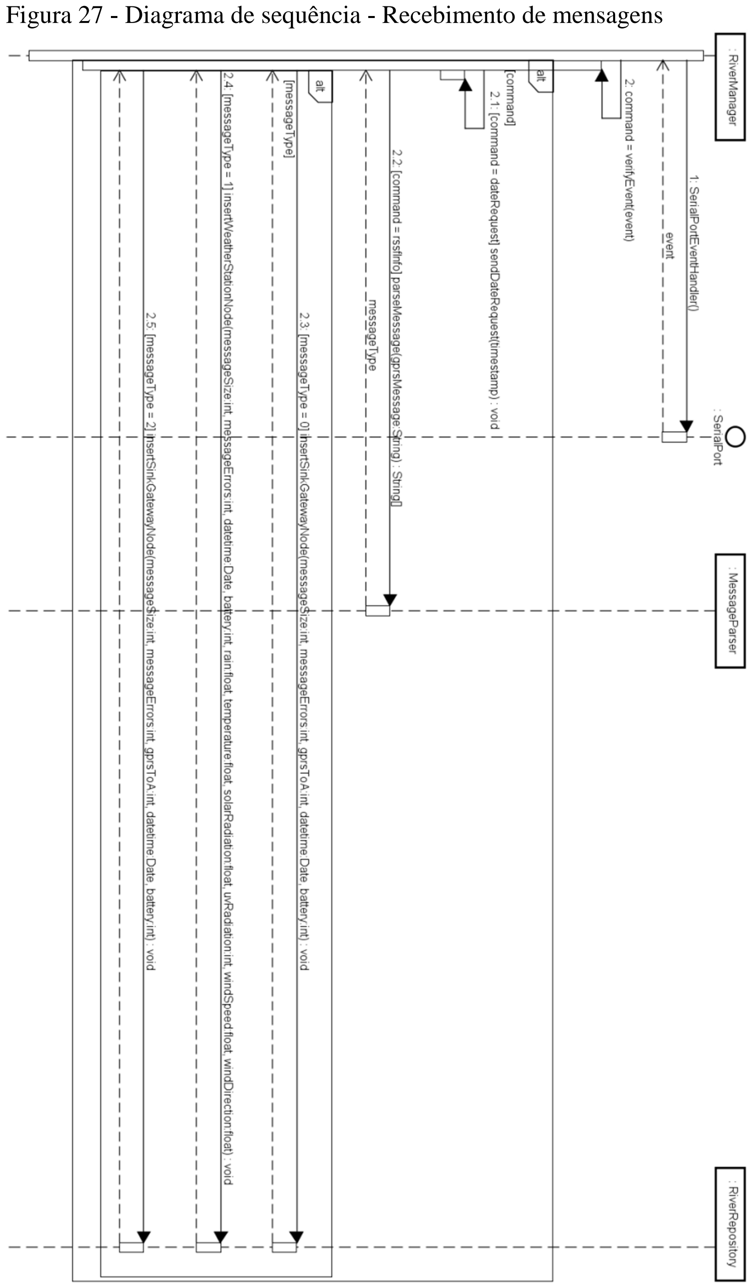



Figura do Autor

#### 4.3.1.2 Serviços

Nesta subseção são apresentados os serviços disponíveis pela camada *RiverServices*. Esta camada é responsável pelo gerenciamento dos Web Services que são disponibilizados pelo RiverSense Server.

Na arquitetura do RiverSense, seguindo os padrões de segurança, os serviços são protegidos por *sessionID*, ou seja, para utilizar um serviço o requisitante deve ter em sua posse uma chave que é concedida após o usuário efetuar *login* através do serviço *authentication*. Portanto se o requisitante não estiver cadastrado, ele não poderá acessar os serviços do *RiverSense*, dessa forma possibilitando um controle de requisições diárias por usuário.

Com a finalidade de exemplificar as funcionalidades dos serviços que podem ser oferecidos pela camada *RiverServices*, foram criados alguns exemplos iniciais. Estes serviços podem oferecer desde uma simples consulta de nodos ou estações de medição existentes, até consultas pontuais, como quais estações possuem determinado tipo de sensor, ou quais as temperaturas foram registradas em um determinado período.

A Tabela 4 apresenta uma lista resumida de serviços inicialmente disponibilizados pelo *RiverSense Server*, descrevendo a função de cada serviço.

Tabela 4 – Web Services criados para consulta

| **Serviço** | **Descrição** |
|---|---|
| Authentication | Verifica usuário e senha, retornando uma chave de acesso. |
| getStationsList | Retorna uma lista com todas as estações em operação. |
| getTemperatureStations | Retorna uma lista com estações que possuem sensor de temperatura. |
| getCurrentTemperature | Retorna a última temperatura de determinado nodo registrada no banco de dados. |
| getTemperature | Retorna uma lista de valores de temperatura em um intervalo de tempo de um nodo. |
| getHighestTemperature | Retorna a maior temperatura em um intervalo de tempo de um nodo. |

#### 4.3.1.3 Página web

A página web permite visualizar os dados a partir de qualquer navegador *web*. Utilizando-se da linguagem PHP para consulta de dados no banco de dados, e a linguagem JavaScript para a ilustração dos dados através de gráficos interativos. A Figura 28 mostra a interação que a página *web* disponibiliza para o usuário, através de gráficos com dados

dos últimos cinco dias. Como observado, a primeira parte apresenta um gráfico de temperatura com intervalo definido pelo usuário através das barras limitadoras, e outro com o intervalo original com dados de até cinco dias atrás.

Figura 28 - Página web para visualização gráfica

Figura do Autor

### 4.3.2 RiverSense RSSF

A rede de sensores sem fio (RSSF) é responsável por coletar dados do ambiente e enviá-los através do Terminal Serial GPRS conectado ao nodo sincronizador da RSSF. A Figura 29 apresenta uma visão geral da comunicação entre os nodos.

Figura 29 - Visão geral dos componentes do RiverSense RSSF

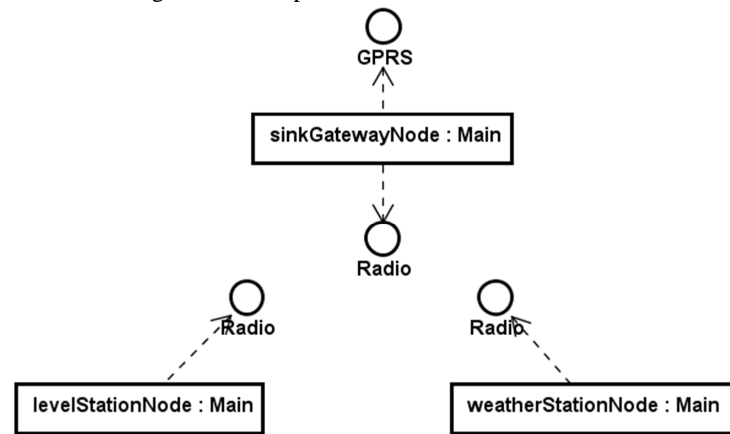



Figura do Autor

As próximas seções descrevem detalhadamente as classes utilizadas e implementadas, assim como o fluxo de programa de cada nodo da rede.

#### 4.3.2.1 Nodo Sincronizador

O nodo sincronizador possui a função de permitir a interação da RSSF com o servidor, conectado ao Terminal Serial GPRS (TSG), através da rede de telefonia móvel. A Figura 30 ilustra os módulos utilizados pelo nodo sincronizador.

Figura 30 - Visão geral das classes do nodo sincronizador e suas dependências

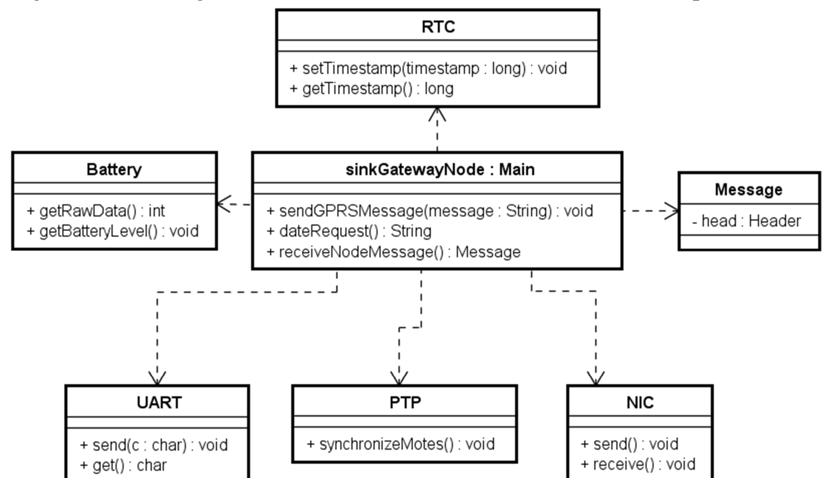



Figura do Autor

Cada módulo de hardware (i.e., UART, Battery, NIC) é encapsulado em uma única classe. Essas classes representam a funcionalidade do hardware. Elas também abstraem a complexidade do hardware e do acesso de registro específico. Portanto, os módulos podem ser usados facilmente sem a necessidade de conhecimento específico do hardware.

É importante destacar a estratégia de manter o microcontrolador no modo *sleep* o máximo possível, sendo ativado somente se ocorrer um evento. O microcontrolador suporta diferentes modos de baixo consumo de energia, que desativam diferentes componentes (i.e., *off*, *sleep*, *light* e *full*). Neste caso foi utilizado o modo *sleep*, que mantém o temporizador e interrupções externas ativos durante este modo. Todos os outros componentes são desligados visando um baixo consumo de energia. Desta forma, quando uma interrupção ocorre, o microcontrolador é ativado e começa a executar novamente. A seguir são listadas as classes utilizadas pelo nodo sincronizador:

- Battery: Esta classe é responsável por abstrair o acesso a dados relacionados ao estado da fonte de alimentação do *mote*.
- Message: Esta classe contém os tipos de mensagens que podem ser trafegadas na RSSF.

- NIC: Uma abreviação de *Network Interface Card*, esta classe é responsável por abstrair o módulo de comunicação do *mote.*
- PTP: Uma abreviação de *Precision Time Protocol*, esta classe é responsável por gerar mensagens de sincronização na RSSF.
- RTC: Uma abreviação de *Real-Time Clock*, esta classe é responsável pela contagem de relógio do *mote*.
- UART: Uma abreviação de *Universal Asynchronous Receiver/Transmitter*, esta classe é responsável por se comunicar com o Terminal Serial GPRS.

A seguir é descrito a operação básica do software. A Figura 31 ilustra o fluxo simplificado do programa da função *Main*. Esta função é chamada após o microcontrolador ser inicizalizado ou reinicializado. A função principal ativa o modo *sleep* do microcontrolador após inicializar todos os módulos a serem utilizados. Posteriormente, a função fica aguardando pela ocorrência de eventos, para assim então executar instruções específicas relacionadas à interrupção.

Figura 31 - Fluxo de programa simplificado do nodo sincronizador

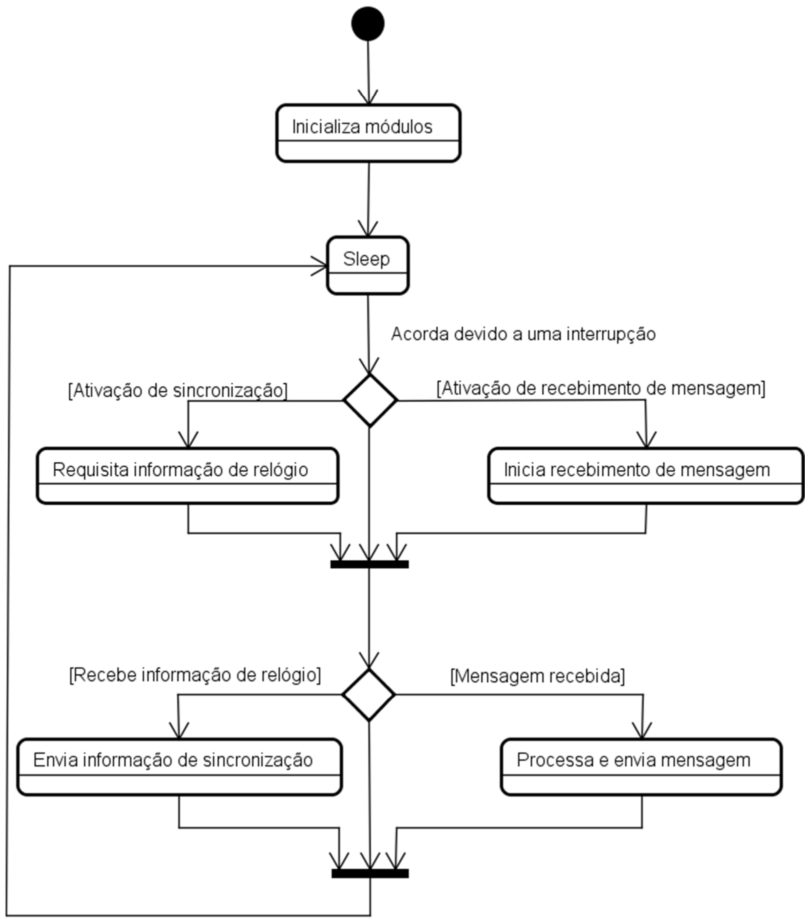



Figura do Autor

O programa pode receber três tipos de interrupção, uma interrupção gerada pela NIC, uma interrupção gerada pela UART ou uma interrupção gerada pelo temporizador do RTC.

No caso de uma interrupção gerada pelo temporizador do RTC, a cada 30 segundos, o programa inicia enviando uma mensagem de requisição de informação de data/hora atual (*Requisita informação de relógio*) para o Terminal Serial GPRS (TSG), através da UART. Após isso, caso não haja nenhum dado na UART ou na NIC, o modo *sleep* é ativado novamente.

Quando um dado é recebido pelo TSG, o mesmo é disponibilizado no *buffer* da UART gerando uma interrupção que acorda o microcontrolador. Como nenhuma das condições é satisfeita o programa segue para o próximo bloco de condições, satisfazendo a condição *Informação de relógio recebida*, desta forma a informação disponibilizada pela UART é lida e inicia-se a sincronização de relógio na RSSF. Após, o modo *sleep* é ativado novamente.

No caso do recebimento de uma mensagem vinda de um nodo da RSSF, a NIC é responsável por gerar uma interrupção (*Ativação de recebimento de mensagem*) e acordar o microcontrolador. O recebimento de pacotes e montagem da mensagem é iniciado, identificando de qual nodo e que tipo de mensagem está sendo recebida. Ao fim do recebimento da mensagem (*Mensagem recebida*), inicia-se o processo de envio da mensagem através da UART, conectada ao TSG, para a aplicação servidora. Após o envio, o modo *sleep* é ativado novamente.

O tempo levado pelo microcontrolador para escorregar (*drift*) 1 segundo na contagem de tempo do RTC é de cerca de 6 minutos. Assim, a fim de assegurar a sincronia do relógio, foi definido um tempo de 30 segundos. A integração do nodo sincronizador com o Terminal Serial GPRS (TSG) foi realizada através do conector de expansão da placa de circuito impresso do TSG, como mostra a Figura 32.

Figura 32 - Interface de expansão disponibilizada pelo Terminal Serial GPRS

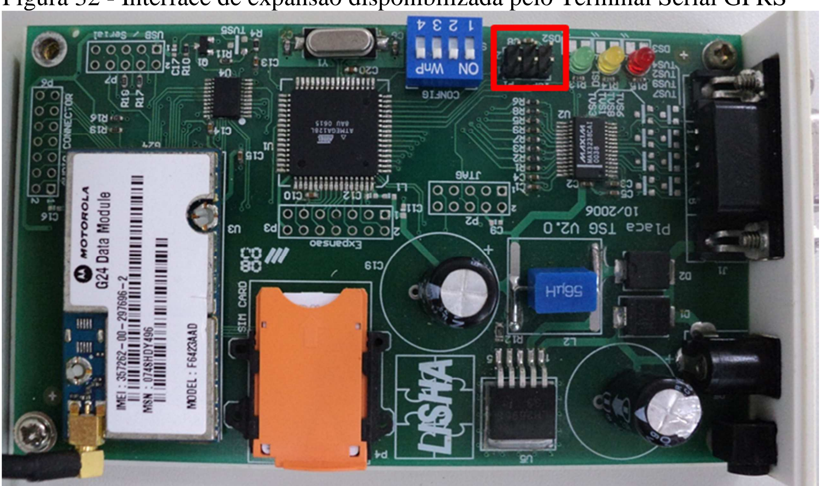



Figura do Autor

4.3.2.2 Nodo Estação Meteorológica

O nodo estação meteorológica possui a função de coletar dados de temperatura, pluviosidade, radiação solar, radiação ultravioleta, e velocidade e direção do vento. A Figura 33 ilustra os módulos utilizados pelo nodo estação meteorológica, destacando em amarelo os módulos desenvolvidos para o RiverSense.

Figura 33- Visão geral das classes do nodo estação meteorológica e suas dependências

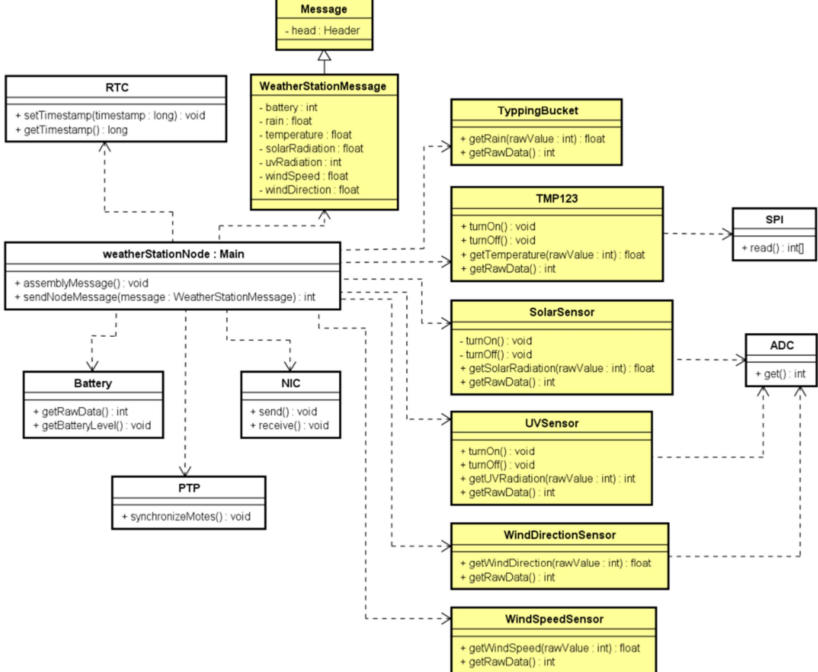



Figura do Autor

Assim como o nodo sincronizador, o nodo estação meteorológica possui módulos de hardware encapsulados em classes, que por sua vez, representam a funcionalidade de cada hardware. No nodo estação meteorológica foram desenvolvidas classes de interface para os sensores (i.e., TMP123, SolarSensor, WindSpeedSensor). A seguir são listadas as classes utilizadas pelo nodo sincronizador, exceto as já descritas no nodo sincronizador, e uma breve descrição sobre cada uma:

- ADC: Esta classe é responsável por converter sinais analógicos em valores digitais.

- SolarSensor: Esta classe é responsável por abstrair o acesso ao sensor de radiação solar. Utiliza a classe ADC para se comunicar com o hardware.
- SPI: Esta classe realiza a interface de comunicação SPI do microcontrolador.
- TMP123: Esta classe é responsável por abstrair o acesso ao sensor de temperatura. Utiliza a classe SPI para se comunicar com o hardware.
- TyppingBucket: Esta classe é responsável por abstrair o acesso ao sensor de pluviosidade.
- UVSensor: Esta classe é responsável por abstrair o acesso ao sensor de radiação ultravioleta. Utiliza a classe ADC para se comunicar com o hardware.
- WeatherStationMessage: Esta classe contém os parâmetros da mensagem a ser enviada para o nodo sincronizador.
- WindDirectionSensor: Esta classe é responsável por abstrair o acesso ao sensor de direção do vento. Utiliza a classe ADC para se comunicar com o hardware.
- WindSpeedSensor: Esta classe é responsável por abstrair o acesso ao sensor de velocidade do vento.

A seguir é descrito a operação básica do software. A Figura 34 ilustra o fluxo simplificado do programa da função *Main* do nodo estação meteorológica. O programa pode receber dois tipos de interrupção, uma interrupção gerada pela NIC ou uma interrupção gerada pelo temporizador do RTC.

No caso de uma interrupção gerada pelo temporizador do RTC, a cada 60 segundos (*Ativação de evento cíclico*), uma vez determinado pelos requisitos do projeto, o programa inicia a coleta de dados requisitando para cada módulo sensorial a leitura atual. Após a coleta de dados (*Coleta de dados completada*) os dados são processados, a mensagem é montada e finalmente enviada ao nodo sincronizador através da NIC. Após o modo *sleep* é ativado novamente.

No caso do recebimento de uma mensagem de sincronização vinda do nodo sincronizador, a NIC é responsável por gerar uma interrupção (*Ativação de sincronização*) e acordar o microcontrolador. O recebimento de pacotes e montagem da mensagem é iniciado, e três mensagens são trocadas entre os nodos para diminuir o erro de sincronização. Ao fim do recebimento da mensagem (*Sincronização*

*completada*), o relógio é atualizado. Após a atualização, o modo *sleep* é ativado novamente.

Figura 34 - Fluxo de programa simplificado do nodo estação meteorológica

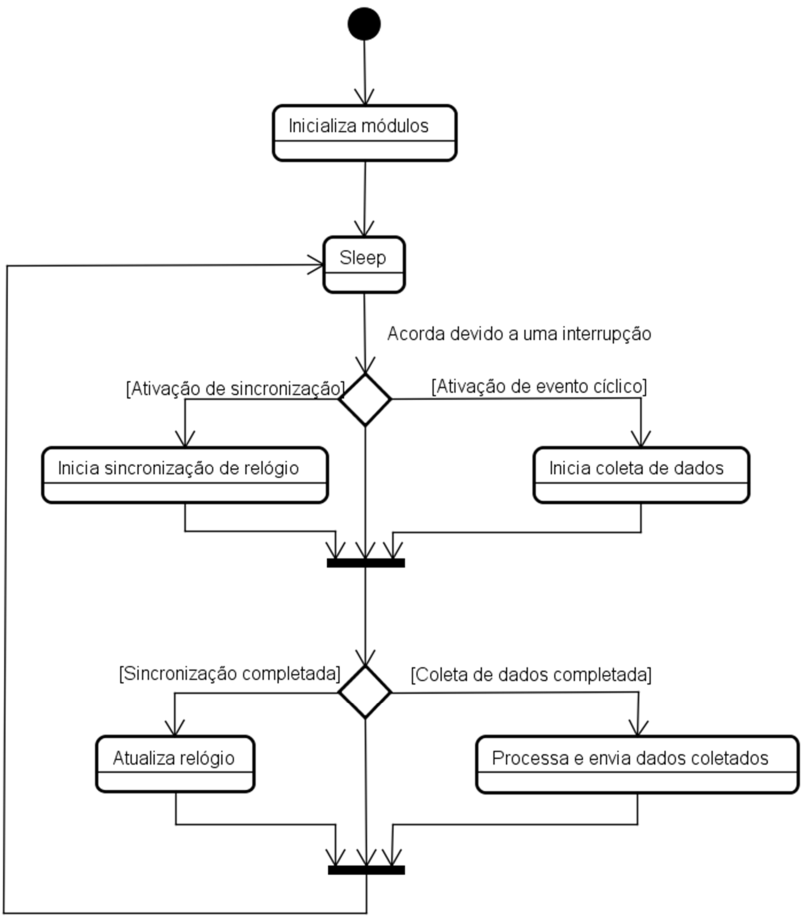



Figura do Autor

A integração do nodo estação meteorológica com os sensores foi realizada através de uma placa de circuito impresso projetada para conduzir os sinais dos sensores até o conector de expansão do *mote*, como mostra a Figura 35. A placa também conta com um suporte para baterias do tipo CR123A.

Figura 35 - Placa de interface para sensores da estação meteorológica

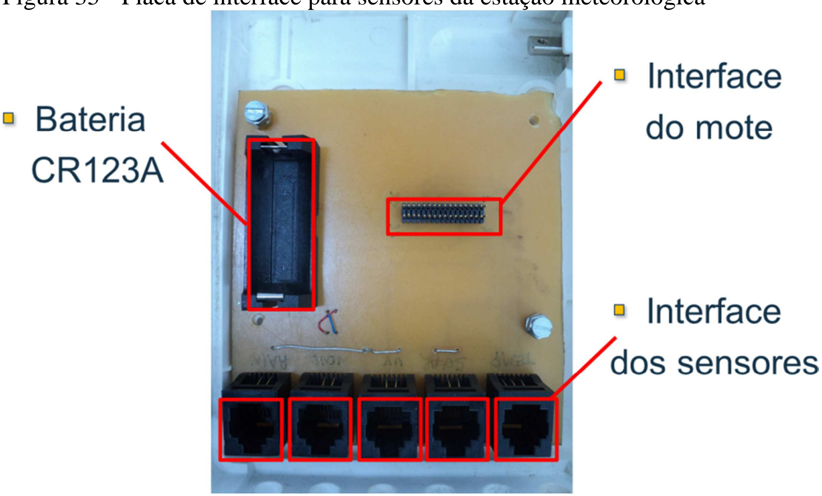



Figura do Autor

4.3.2.3 Nodo Estação de Nível

Assim como o nodo estação meteorológica, o nodo estação de nível possui a função de coletar dados do ambiente, neste caso, dados de nível o rio. A Figura 36 ilustra os módulos utilizados pelo nodo estação de nível que possui módulos de hardware encapsulados em classes que representam a funcionalidade de cada hardware.

No nodo estação de nível foi desenvolvida a classe de interface para o sensor de nível TPST18SUB. A seguir são listadas as classes utilizadas pelo nodo sincronizador, exceto as já descritas no nodo sincronizador e no nodo estação meteorológica, e uma breve descrição sobre cada uma:

- LevelStationMessage: Esta classe contém os parâmetros da mensagem a ser enviada para o nodo sincronizador.
- TPST18SUB: Esta classe é responsável por abstrair o acesso ao sensor de nível. Utiliza a classe ADC para se comunicar com o hardware.

Figura 36 - Visão geral das classes do nodo estação de nível e suas dependências

Message
- head : Header
- attribute0 : int

LevelStationMessage
+ battery : int
+ riverLevel : float
- timestamp : long
- messageType : int

RTC
+ setTimestamp(timestamp : long) : void
+ getTimestamp() : long

ADC
+ get() : int

levelStationNode : Main
+ sendNodeMessage(message : LevelStationMessage) : int
+ assemblyMessage() : void

TPST18SUB
- turnOn() : void
- turnOff() : void
- getLevel(rawValue : int) : float
- getRawData() : int

Battery
+ getRawData() : int
+ getBatteryLevel() : void

PTP
+ synchronizeMotes() : void

NIC
+ send() : void
+ receive() : void

Figura do Autor

O fluxo de programa deste nodo possui a mesma ideia do nodo estação meteorológica, ou seja, coletar dados e sincronizar o relógio. Para entender o comportamento do sensor de nível foi necessário realizar um ensaio, por não haver informações em seu manual de referência. Assim, para simular a coluna de água foi montado um ensaio de bancada utilizando um tubo de PVC com 170 cm de altura por 10 cm de diâmetro. Após realizar medições de níveis com uma divisão de escala de 1 cm, chegou-se ao gráfico de comportamento do sensor, como pode ser visto na Figura 37. A linha pontilhada vermelha representa as medições realizadas, já a linha preta representa a curva de comportamento estimada, neste caso linear.

Figura 37 - Gráfico de comportamento do sensor de nível

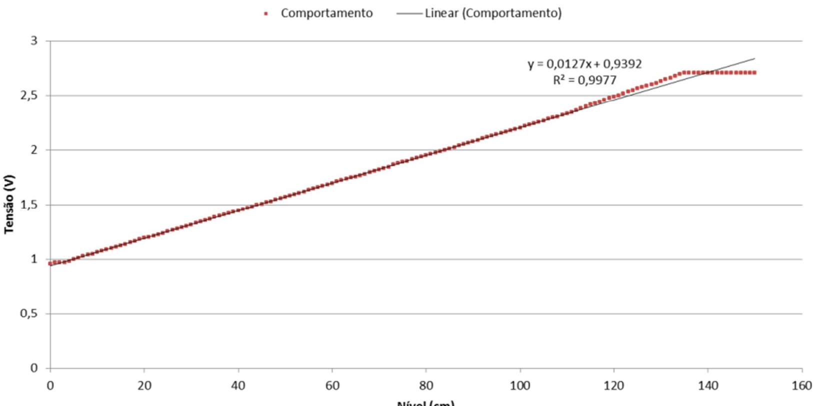



Figura do Autor

Após o ensaio notou-se que o sensor, inicialmente construído para medir níveis até 150 cm, parou de variar a sua corrente após o nível de 135 cm. Uma vez que este sensor já havia sido usado, pressupomos que o mesmo poderia estar avariado, permitindo medições somente até 135 cm de altura. Baseado na curva de dados do gráfico foi possível obter a equação que mais se adequava ao comportamento, com um coeficiente de determinação igual a 0,9994, ou seja, muito próximo do comportamento real.

$$f(x) = 0{,}0624x - 71{,}79 \qquad R^2 = 0{,}9994$$

A integração do nodo estação de nível com o sensor foi realizada através de uma placa de circuito impresso projetada para conduzir os sinais de ativação e dados do sensor, através de um resistor de *shunt*, até o conector de expansão do *mote*, como mostra a Figura 35. O resistor de shunt é uma solução para medir corrente em circuitos, e nada mais é que um resistor dimensionado para medir um intervalo de corrente conhecido. Como o sensor necessita de uma alimentação de 12V, a placa possui um suporte para uma pequena bateria do tipo 23A, além de possuir um conector para alimentação externa.

Figura 38 - Placa de interface para sensor da estação de nivel

Figura do Autor

## 4.4 CONSIDERAÇÕES SOBRE O CAPÍTULO

Neste capítulo foram abordados os aspectos da arquitetura proposta, assim como as partes que a compõem. Também foram abordadas as ferramentas e estratégias utilizadas para a implementação do sistema. O próximo capítulo apresenta os resultados obtidos, assim como uma breve avaliação sobre o sistema proposto.

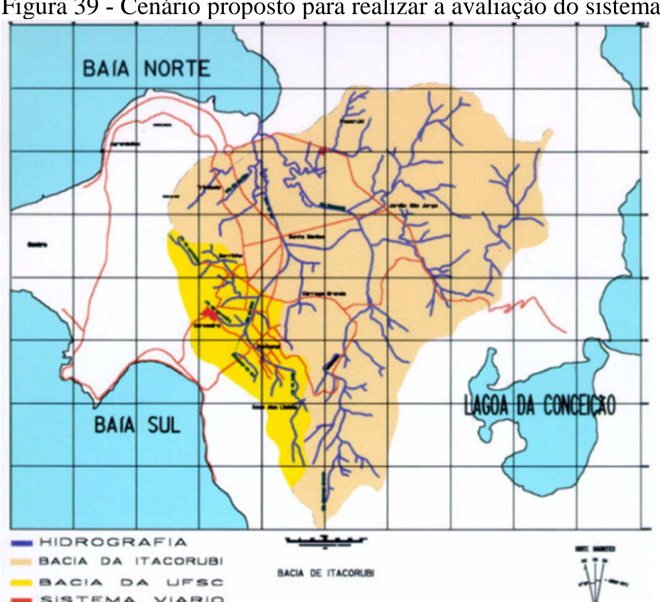
# 5 RESULTADOS OBTIDOS

Este capítulo descreve os resultados obtidos, assim como o estudo de caso proposto para a validação do sistema. Na seção 5.2 são realizadas algumas considerações na intenção de avaliar o sistema como um todo.

## 5.1 IMPLANTAÇÃO

A comunidade científica tem utilizado *deployments* para avaliar os sistemas de RSSFs. A Figura 39 ilustra a sub-bacia do campus da UFSC na bacia do Itacorubi em Florianópolis. Foi proposto realizar um *deployment* com três nodos medindo os dados ambientais na região próxima a exutória do rio da UFSC, ou seja, o local por onde passa toda a água precipitada na bacia hidrográfica. O objetivo é avaliar a resiliência do hardware ao ambiente, a comunicação em um ambiente não controlado e o ciclo de vida da aplicação no ambiente. Porém, devido a imprevistos no processo de aquisição do material para a implementação do sistema e realização do *deployment*, não houve tempo hábil para realizá-lo, entretanto, foi realizada uma validação em laboratório.

Figura 39 - Cenário proposto para realizar a avaliação do sistema




Adaptado de (TASCA et al., 2009)

Em laboratório, foram realizados testes em cada nodo separadamente, avaliando a coleta de dados e comunicação, a fim de excluir possíveis problemas relacionados à rede. Após foi realizado um teste com os três nodos em funcionamento, durante dois dias, com a finalidade de avaliar os dados obtidos pelos sensores, assim como o consumo de energia do nodo estação meteorológica.

Para realizar os testes com o sensor de nível, foi necessário desenvolver um ambiente de teste, que consistiu em um tubo de PVC com 170 cm de altura por 10 cm de diâmetro para simular a coluna de água. O sensor apresentou uma precisão maior do que o esperado, como pode ser visto na Figura 40. Ao medir um nível estático de 17 cm, percebeu-se uma variação de leitura de cerca de ±3 cm, oscilando entre 19,6 cm e 14,7 cm.

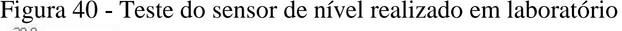


Figura 40 - Teste do sensor de nível realizado em laboratório

Figura do Autor

Na Figura 41 podemos ver o sensor de temperatura em funcionamento, uma vez que o ambiente do laboratório possui ar condicionado, foi possível identificar o funcionamento do condicionador de ar, configurado para manter uma temperatura de 23ºC, variando a temperatura entre 22ºC e 24ºC.

Figura 41 - Teste do sensor de temperatura realizado em laboratório

Durante os testes a bateria estava sendo monitorada, possibilitando avaliar o limite de funcionamento do sensor de temperatura. O sensor TMP123 funcionou como esperado até a tensão

de 1953 mV, abaixo deste valor de tensão o sensor realizou leituras de temperatura inesperadas (i.e., 6ºC e 400ºC), como pode ser observado na Figura 42. Neste caso, com uma faixa de segurança, podemos assegurar o funcionamento do sensor até um limite 2000mV, apesar do manual de referencia sugerir uma alimentação entre 2,7V e 5,5V.

Figura 42 - Leituras inesperadas realizadas pelo sensor de temperatura

Figura do Autor

Na Figura 43 podemos observar o sensor de radiação solar em funcionamento, apesar de não estar em um ambiente propício, foi possível realizar medições de ondas entre 300 e 1100 nanômetros emitidas pelas lâmpadas fluorescentes.

Figura 43 - Teste do sensor de radiação solar realizado em laboratório

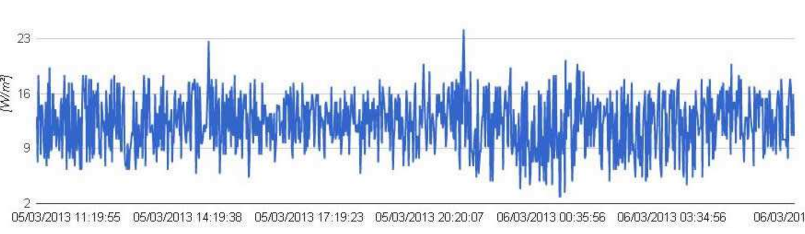



Figura do Autor

Durante os testes, foi possível avaliar o limite de funcionamento do sensor de temperatura. O sensor de radiação solar funcionou como esperado até a tensão de 1893mV, abaixo deste valor de tensão o sensor começou a realizar leituras de radiação acima do coletado dos dias anteriores, como pode ser observado na Figura 44. Neste caso, com um limite de segurança, podemos assegurar o funcionamento do sensor até um limite 1900mV, apesar do manual de referencia sugerir uma alimentação entre 2,7V e 3,3V.

Figura 44 - Leituras inesperadas realizadas pelo sensor de radiação

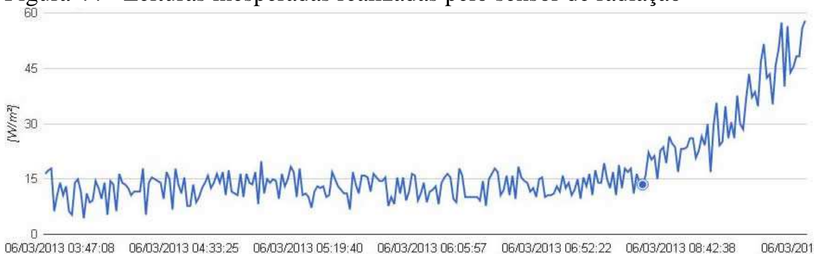



Figura do Autor

Para testar o sensor de pluviosidade, realizamos testes utilizando uma pipeta para ter um controle maior sobre a quantidade de água despejada no coletor. Para realizar a leitura de 1mm de chuva, é necessário primeiramente realizar um cálculo simples, onde a leitura de 1mm de chuva corresponde à área da secção transversal do coletor multiplicado por 1mm de altura. No nosso caso, a área da secção transversal medida era de 200 cm². Logo para realizar uma leitura de 1mm de chuva foi necessário despejar 20ml de água no coletor resultando em uma oscilação da báscula, logo, uma leitura correspondente a 1mm de chuva no nodo.

Para testar os sensores de velocidade e direção do vento foram realizados testes simples apenas verificando o funcionamento. Para o sensor de direção realizou-se várias vezes a mudança de direção na aleta que é responsável por interagir com o vento, evidenciando a funcionalidade da mesma. Já para o sensor de velocidade, da mesma forma que o sensor de direção, foram realizadas várias revoluções em curtos períodos, evidenciando a variação na velocidade coletada.

O sensor de radiação ultravioleta não pôde ser testado propriamente, uma vez que a radiação ultravioleta é medida em uma escala de índices que vai de 0 a 15, sendo 15 um nível extremo de radiação ultravioleta. Após uma análise dos valores coletados diretamente do conversor analógico digital observou-se que as medições da radiação ultravioleta emitida pelas lâmpadas fluorescentes não chegavam a um décimo do primeiro índice da escala de radiação ultravioleta. Confirmando o estudo realizado por (LYTLE et al., 1993) ao avaliarem a emissão de radiação ultravioleta em 58 tipos de lâmpadas fluorescentes, chegou-se a conclusão de que a exposição às lâmpadas fluorescentes numa jornada de trabalho de 8h/dia equivaleria a um minuto de exposição ao sol ao meio-dia. Esta comparação demonstra claramente a relativa insignificância da quantidade de raios ultravioletas que algumas lâmpadas fluorescentes podem emitir.

Com a finalidade de avaliar a conexão GPRS, foram coletados dados relacionados ao tempo de envio e resposta de requisições (também chamado de *Round-Trip Time*) do nodo sincronizador através da rede GPRS. O *Round-Trip Time* (RTT) foi calculado a nível de aplicação. Baseado nos dados coletados (Figura 45) o RTT variou entre 1161µs (1,1ms) e 581457µs (581ms), obtendo-se uma média em torno de 181698µs (181ms) para enviar um dado e receber uma resposta.

Figura 45 - Round-Trip Time medido da conexão GPRS no nodo sincronizador

Figura do Autor

Apesar dos valores obtidos no período de desenvolvimento deste trabalho, na região do campus da UFSC, existia um sinal adequado da operadora TIM. Sugere-se que o usuário final faça um estudo sobre a operadora mais adequada para a sua região e necessidade.

Através dos dados coletados pelo sistema é possível realizar uma estimativa de ocorrência de uma inundação, no intuito de alertar a população. Estas estimativas são extremamente dependentes do rio a ser monitorado, sendo o sistema configurável a fim de cobrir o comportamento de cada rio. Por exemplo, no rio da UFSC, com base na Figura 46, podemos definir a vazão na exutória da bacia do campus da UFSC após a ocorrência de uma chuva ininterrupta de 30 minutos com média de 0,5mm de precipitação (barras azuis) causando um aumento de cerca de 4$m^3$/s de vazão de água após 10 minutos do término da chuva.

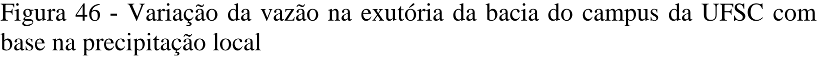
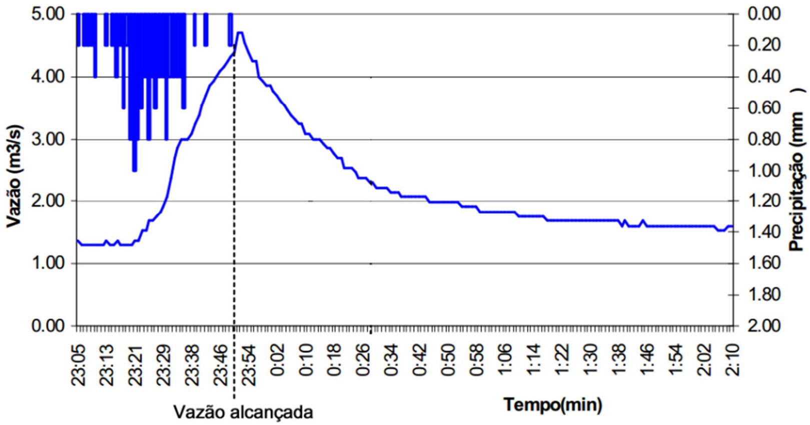


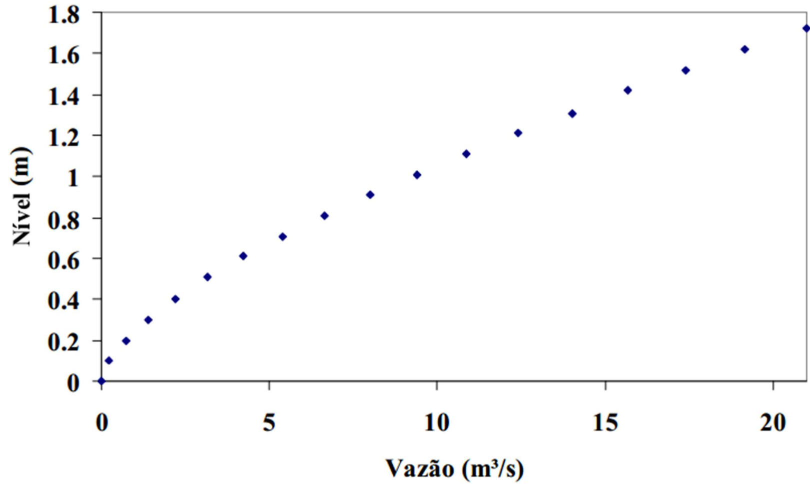
Figura 46 - Variação da vazão na exutória da bacia do campus da UFSC com base na precipitação local




Adaptado de (KOBIYAMA et al., 2006)

Para definir o nível através da vazão utilizou-se da equação definida por (KOBIYAMA et al., 2006) ilustrada na Figura 47, que define o comportamento da exutória da bacia do campus da UFSC. Com base nisso, salvo em condições atípicas, pode-se afirmar que a ocorrência de uma chuva ininterrupta de 30 minutos com média de 0,5mm de precipitação causa um aumento de cerca de 60 cm no nível do rio após 10 minutos do término da chuva.

Figura 47 - Comportamento da exutória da bacia do campus da UFSC




Adaptado de (KOBIYAMA et al., 2006)

Segundo (KOBIYAMA; MANFROI, 1999), o sucesso de modelagens, simulações e previsões depende da qualidade do monitoramento, e que não há um bom modelo sem o uso de dados obtidos pelo fenômeno monitorado. Assim, a modelagem e o monitoramento são métodos científicos utilizados paralelamente na hidrologia.

## 5.2 AVALIAÇÃO

Com o intuito de avaliar a plataforma EPOSMoteII, podemos destacar o seu tamanho, pequeno o bastante para adaptar-se a aplicações onde há requisitos de tamanho. Por exemplo, se não fosse o capacitor no meio da placa de circuito impresso do Terminal Serial GPRS, seria possível integrar o mote ao módulo com um cabo maleável e fechar a caixa, sem necessidade de mudanças.

A plataforma, originalmente projetada para ser modular, possui uma interface de expansão de 34 pinos, oferecendo acesso a várias funções do microcontrolador, tais como comunicação UART e SPI, conversores analógico-digital, *timers* e pinos de interrupção externa.

Em testes de campo, foi obtido um alcance de comunicação de 80m, com uma antena omnidirecional, que representa um valor adequado, ao se considerar a distância proposta originalmente para redes deste tipo, que é de 100m. O mesmo *mote* pode chegar a alcances de até 1,5km através de modificações na antena do módulo de rádio, porém consumindo mais energia.

No nodo estação meteorológica, único nodo testado com bateria, foi utilizado um *duty cycle* de 1 minuto para leitura dos sensores e envio da informação, e 30 segundos para a sincronização do relógio entre os nodos. Além disso, vale ressaltar que neste nodo não há uma chave para ligar e/ou desligar os sensores, ou seja, os sensores estavam consumindo energia durante todo o teste. Avaliando o consumo de bateria apresentado na Figura 48, apesar de obtermos um resultado abaixo do esperado, foi possível obter, em um primeiro momento, dados de comportamento de consumo do nodo.

Figura 48 - Nível de bateria durante o teste em laboratorio

Figura do Autor

Na Figura 49 é ilustrado o momento anterior à queda de tensão, onde os sensores deixaram de funcionar como esperado. Porém vale ressaltar que o *mote*, designado para operar entre 2,7V e 3,6V, continuou funcionando até uma tensão de 1536mV. Sendo a tensão de funcionamento do sistema limitada apenas pelos sensores que o compõem.

Figura 49 - Nível de bateria no momento em que os sensores pararam de funcionar como esperado

Figura do Autor

## 5.3 CONSIDERAÇÕES SOBRE O CAPÍTULO

Embora não tenha sido possível realizar um *deployment* da RSSF e os dados não tenham sido obtidos em um cenário real, foi possível obter resultados satisfatórios para avaliar o protótipo do RiverSense como um todo.

# 6 CONCLUSÃO

A evolução das tecnologias de comunicação sem fio e o maior poder de processamento em circuitos integrados cada vez menores tem gerado oportunidades em diversas áreas, incluindo as áreas de sensoriamento remoto e monitoramento ambiental (PUCCINELLI; HAENGGI, 2005). A partir deste cenário, possibilita-se coletar dados do ambiente em diversos locais e a qualquer momento com sensores conectados a nodos de Rede de Sensores Sem Fio. Com base neste contexto, informações que poderiam ter uma maior granularidade não são exploradas. O principal fator de motivação deste trabalho foi a definição de um sistema para obtenção, tratamento e disponibilização de informações do meio-ambiente, neste caso dados relacionados ao nível de rios, a fim de alertar a sociedade sobre ocorrências de inudações em locais pontuais.

Neste sentido, esse trabalho apresentou o RiverSense, um sistema que tem como objetivo o monitoramento de rios e divulgação dos dados coletados que podem ser utilizados por outras aplicações que desejam utilizar informações ambientais. Desenvolveu-se um protótipo do hardware que realiza a interface dos sensores com o *mote* da plataforma EPOSMoteII e a integração de meios de comunicação, como o Terminal Serial GPRS. Assim como o software desenvolvido especialmente para compreender o comportamento dos sensores. O protótipo permitiu testar o consumo de energia de um nodo, a sincronização de relógio a nível de software entre os nodos na rede, e a fidelidade dos dados coletados pelos sensores. O trabalho alcançou os seguintes objetivos definidos inicialmente:

- A medição de variáveis climáticas locais relacionadas ao ambiente do rio;
- Realizar uma avaliação da plataforma EPOSMote em aplicações reais;
- Avaliar a possibilidade de conexão de sensores adicionais ao nodo.
- A visualização de dados por meio de uma página *web* pública.
- A disponibilidade de serviços para uso das informações por outros sistemas.

Com base na Tabela 3, onde são apresentadas as principais características dos trabalhos relacionados, pode-se definir as principais características do RiverSense:

- Gerenciamento Remoto: o sistema permite a sincronização do relógio da RSSF, através da rede GPRS. Assim como a visualização de dados coletados através de gráficos;
- Topologia: a topologia utilizada foi a *single-hop*;
- Frequência de Operação: o sistema utilizou a frequência de 2.4GHz para a comunicação dos nodos e a rede GPRS para comunicação com o servidor;
- Alcance: a RSSF possui um alcance de 80m na comunicação ente nodos;
- Camadas de rede: o sistema utiliza a abordagem de duas camadas de rede, permitindo a independência da RSSF em relação à aplicação servidora.

Este trabalho também obteve contribuições técnicas do ponto de vista das tecnologias utilizadas, uma vez que ao utilizá-las estavam sendo avaliadas suas características. Um exemplo de problema identificado nas tecnologias envolvidas está relacionado ao Terminal Serial GPRS. A comunicação entre o *mote* e o módulo GPRS não se estabelecia. Ao se realizar a medição com equipamentos de teste da interface do módulo, chegou-se a conclusão que os pinos da interface estavam invertidos, sendo necessário realizar uma atualização no manual do módulo.

Outro exemplo de contribuição técnica está relacionado à capacidade de gravar na memória Flash do mote, um código em python desenvolvido pela comunidade do sistema operacional TinyOS, responsável por abrir a porta serial e enviar os comandos de gravação. O firmware existente possuía uma linha de código responsável por enviar para o primeiro endereço de memória da flash, um comando para limpar a memória, seguido de um comando para gravação. Porém, este comando estava sendo enviado para o endereço seguinte e não para o endereço inicial. Gerando um *timeout* ao tentar gravar, pois o código aguardava uma resposta do microcontrolador, que nada fazia. Portanto o código foi corrigido e disponibilizado no repositório SVN do EPOSMoteII.

Embora não tenha sido possível realizar o *deployment* para obter dados ambientais do rio, os resultados obtidos na validação atenderam as expectativas com relação à coleta de dados, por parte da RSSF; recepção de dados, persistência e visualização por parte da aplicação servidora em busca da disponibilização de dados ambientais para a sociedade. O uso de RSSFs para coleta de dados ambientais permite aos

sistemas de previsão tornarem-se mais efetivos, devido ao uso de informações mais precisas sobre comportamentos de microclimas. O objetivo deste trabalho foi demonstrar que é viável o uso de RSSFs na coleta de dados, gerando dados geograficamente mais precisos.

As próximas atividades permitirão a continuidade do estudo: (1) a execução de um *deployment* com a finalidade de testar a resiliência do hardware ao ambiente; (2) avaliar o projeto de um circuito para coleta de energia solar, aumentando o ciclo de vida da aplicação; (3) considerar o uso de sensores digitais ao invés de sensores analógicos, que possuem um alto tempo de resposta e alto consumo de energia.

Ao longo do desenvolvimento deste trabalho, foram identificados trabalhos futuros relacionados ao RiverSense. A seguir são listados alguns tópicos:

- Comparar os dados obtidos em laboratório com dados de um caso real (i.e., Rio da UFSC);
- Implementar a parametrização de sensores remotamente;
- Avaliar técnicas de Amostragem Adaptativa (*Adaptive Sampling*) com a finalidade de diminuir o número de leituras de sensores que consomem muita energia;
- Avaliar técnicas de Sensoriamento Compressivo (*Compressive Sensing*) para diminuir o tempo de transmissão de pacotes na rede, reduzindo o tempo do rádio ligado, com a finalidade de economizar energia;
- Avaliar um sistema de alarme baseado em níveis atingidos pelos dados coletados. Esses alarmes podem enviar avisos através de e-mails, SMS e/ou redes sociais, a fim de alertar a população;
- Projetar um *gateway* dedicado para Redes de Sensores Sem Fio baseado nos requisitos de baixo consumo de energia e tamanho.

# REFERÊNCIAS


BASHA, E.; RAVELA, S. Model-based monitoring for early warning flood detection. **Proceedings fo the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys)**, p. 295-308, 2008.

BASHA, E.; RUS, D. Design of early warning flood detection systems for developing countries. **International Conference on Information and Communication Technologies and Development**, p. 1-10, dez. 2007.

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. **The Unified Modeling Language for Object-Oriented Development**. [S.l: s.n.].

BRUNO, L. et al. 6LoWDTN: IPv6-enabled Delay-Tolerant WSNs for Contiki. **International Conference on Distributed Computing in Sensor Systems and Workshops DCOSS**, p. 1-6, 2011.

CADSOFT USA. **Eagle PCB Designer**. Disponível em: <http://www.cadsoftusa.com/>. Acesso em: 10 dez. 2011.

DAVISNET. **DavisNet**. Disponível em: <http://www.davisnet.com/>. Acesso em: 15 jun. 2012.

DOUGLAS, B. C. Global sea rise: a redetermination. **Surveys in Geophysics**, v. 18, n. 2, p. 279-292, 1997.

DUNKELS, A.; GRONVALL, B.; VOIGT, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. **29th Annual IEEE International Conference on Local Computer Networks**, v. 2004, p. 455-462, 2004.

FANG, W.-C. F. W.-C.; LIN, T.-H. L. T.-H. Low-Power Radio Design for Wireless Smart Sensor Networks. **2006 International Conference on Intelligent Information Hiding and Multimedia**, p. 583–586, 2006.

FOUNDATION, M. **JavaScript Official Webpage**. Disponível em: <htt://developer.mozilla.org/en/JavaScript>. Acesso em: 13 jul. 2012.

FRÖHLICH, A. A. Application-Oriented Operating Systems. **Sankt Augustin GMD Forschungszentrum Informationstechnik**, v. 1, n. 17, p. 220, 2001.

FRÖHLICH, A. A. A Comprehensive Approach to Power Management in Embedded Systems. **International Journal of Distributed Sensor Networks**, v. 2011, p. 1-19, 2011.

FRÖHLICH, A. A.; WANNER, L. F. Operating System Support for Wireless Sensor Networks. **Journal of Computer Science**, v. 4, n. 4, p. 272-281, 2008.

FROHLICH, A.; STEINER, R.; RUFINO, M. A Trustful Infrastructure for the Internet of Things based on EPOSMote. **Autonomic and Secure**, 2011.

HAJDAREVIC, K.; KURTANOVIC, E. Simulation application for Sleep-Awake Probabilistic Forwarding Protocol for Smart Dust environments. . 2011, p. 468-471.

HILL, J. et al. System architecture directions for networked sensors. **ACM SIGARCH Computer Architecture News**, v. 35, n. 5, p. 93-104, 2000.

HILL, J. L.; CULLER, D. E. Mica: a wireless platform for deeply embedded networks. **IEEE Micro**, v. 22, n. 6, p. 12-24, nov. 2002.

HOWITT, I.; GUTIERREZ, J. A. IEEE 802.15.4 low rate - wireless personal area network coexistence issues. . 2003, p. 1481-1486.

HU, Z. The Research of Several Key Question of Internet of Things. . 2011, p. 362-365.

HUGHES, D. et al. A middleware platform to support river monitoring using wireless sensor networks. **Journal of the Brazilian Computer Society**, v. 17, n. 2, p. 85-102, 15 fev. 2011.

INGELREST, F. et al. SensorScope: Applications-Specific Sensor Network for Environmental Monitoring. **ACM Transactions on Sensor Networks**, v. 6, n. 2, p. 1-32, 1 fev. 2010.

JOHNSON, M. et al. A comparative review of wireless sensor network mote technologies. **2009 IEEE Sensors**, p. 1439-1442, 2009.

KERKEZ, B.; GLASER, S. **Wireless Sensor Networks for Hydrologic Measurements: Performance Evaluation and Design Methods**. [S.l: s.n.].

KOBIYAMA, M. et al. Estimativa Morfométrica e Hidrológica do Tempo de Concentração na Bacia do Campus da UFSC, Florianópolis-SC. **I Simpósio de Recursos Hídricos do Sul-Sudeste**, v. 1, p. 11, 2006.

KOBIYAMA, M.; MANFROI, O. J. Importância da modelagem e monitoramento em bacias hidrográficas. **Curso de Extensão: O Manejo de bacias hidrográficas sob a perspectiva florestal - UFPR**, p. 111-118, 1999.

LANGENDOEN, K.; BAGGIO, A.; VISSER, O. Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. **Proceedings 20th IEEE International Parallel & Distributed Processing Symposium**, p. 8 pp., 2006.

LEE, Y. et al. A modular 1mm 3 die-stacked sensing platform with optical communication and multi-modal energy harvesting. **SolidState Circuits**, p. 402-404, 2012.

LEVIS, P. et al. **Ambient Intelligence - TinyOS: An Operating System for Sensor Networks**. [S.l: s.n.]. p. 115-148

LI, Y. L. Y. et al. Hybrid Micropower Source for Wireless Sensor Network. . 2008, p. 678-681.

LISHA. **EPOS Project**. Disponível em: <http://epos.lisha.ufsc.br>.

LUNDQUIST, J. D.; CAYAN, D. R.; DETTINGER, M. D. Meteorology and Hydrology in Yosemite National Park : A Sensor Network Application. p. 518-528, 2003.

LYTLE, C. et al. An estimation of squamous cell carcinoma risk from ultraviolet radiation emitted by fluorescent lamps. **Photodermatology, Photoimmunology & Photomedicine**, v. 9, p. 268-274, 1993.

MA, H.-D. Internet of Things: Objectives and Scientific Challenges. **Journal of Computer Science and Technology**, v. 26, n. 6, p. 919-924, 28 nov. 2011.

MACIEL, C. B. et al. Desastres Naturais e Antropogênicos: Estudo de caso do projeto resposta ao desastre em Santa Catarina no ano de 2008. **XVIII Simpósio Brasileiro de Recursos Hídricos**, 2009.

MIAJAS. **Anemômetro**. Disponível em: <http://miajas.com/intercom/Anemometro/videoVeleta.htm>. Acesso em: 2 dez. 2012.

OKAZAKI, A. Ant-based Dynamic Hop Optimization Protocol: A routing algorithm for Mobile Wireless Sensor Networks. **GLOBECOM Workshops**, p. 1179-1183, 2011.

ORACLE. **SunSpot**. Disponível em: <http://www.sunspotworld.com/>. Acesso em: 17 nov. 2011.

ORACLE. **Java Official Website**. Disponível em: <http://www.java.com>. Acesso em: 16 maio. 2012a.

ORACLE. **MySQL Official Website**. Disponível em: <http://www.mysql.com/>. Acesso em: 18 out. 2012b.

ORACLE. **GlassFish Official Website**. Disponível em: <http://glassfish.java.net/>. Acesso em: 10 set. 2012c.

PHP. **PHP Official Website**. Disponível em: <http://php.net/>. Acesso em: 11 set. 2012.

PUCCINELLI, D.; HAENGGI, M. Wireless sensor networks: applications and challenges of ubiquitous sensing. **IEEE Circuits and Systems Magazine**, v. 5, n. 3, p. 19-31, 2005.

RNP. **CIA$^2$ - Construindo Cidades Inteligentes da Instrumentação dos Ambientes ao Desenvolvimento de Aplicações**. Disponível em: <http://www.nr2.ufpr.br/~cia2/?page_id=185>. Acesso em: 16 jul. 2012.

ROCHA, H.; KOBIYAMA, M. Análise Estatística de Chuvas Intensas Ocorridas nos Municípos de Blumenau e Rio dos Cedros no Período de Agosto de 2008 a Janeiro de 2009. **XVIII Simpósio Brasileiro de Recursos Hídricos**, p. 1-14, 2009.

ROMER, K. The design space of wireless sensor networks. **Wireless Communications, IEEE**, 2004.

SHARMA, A.; BANERJEE, A.; SIRCAR, P. Performance analysis of energy-efficient modulation techniques for wireless sensor networks. **2008 Annual IEEE India Conference**, v. 2, p. 1-6, 2008.

SMOLNIKAR, M. et al. ISM bands spectrum sensing based on Versatile Sensor Node platform. . 2010, p. 1-5.

TASCA, F. et al. Prevenção de desastres naturais através da popularização da hidrologia. **Seminário Internacional de Defesa Civil**, 2009.

TEXAS INTRUMENTS. **TMP123**. Disponível em: <http://www.ti.com/product/tmp123>. Acesso em: 9 ago. 2012.

TINYNODE. **TinyNode**. Disponível em: <http://www.tinynode.com>. Acesso em: 8 jan. 2012.

TRUBILOWICZ, J.; CAI, K.; WEILER, M. Viability of motes for hydrological measurement. **Water Resources Research**, v. 45, p. 1-6, 23 jan. 2009.

VARAPRASAD, G. Wireless sensor network for volcano environments. **ACM SIGBED Review**, v. 6, n. 3, 1 out. 2009.

W3C. **WebServices**. Disponível em: <http://www.w3.org/2002/ws/>. Acesso em: 12 ago. 2012.

WANG, H. et al. Acoustic sensor networks for woodpecker localization. **Proceeding of the SPIE conference on Adavanced Signal Processign Algorithms, Architectures, and Implementations**, 2005.

WARNEKE, B. et al. Smart Dust: communicating with a cubic-millimeter computer. **Computer**, v. 34, n. 1, p. 44-51, 2001.

WEATHERSHACK. **WeatherShack**. Disponível em: <http://www.weathershack.com/>. Acesso em: 23 fev. 2012.

WIND 101 BLOG. **Wind Sensor**. Disponível em: <http://www.wind101.net/blog/electronic-wind-vane-sensor-with-frequency-potentiometer-output/>. Acesso em: 12 dez. 2012.

YIFENG, W. Y. W.; XIAOFENG, Z. X. Z.; MING, Z. M. Z. A Solution of Video Sensor Node Implementation. . 2007.

ZHANG, Y. et al. Agent-based smart objects management system for real-time ubiquitous manufacturing. **Robotics and ComputerIntegrated Manufacturing**, v. 27, n. 3, p. 538-549, 2011.

ZHANG, Y. et al. An Eco-Hydrology Wireless Sensor Demonstration Network in High-Altitude and Alpine Environment in the Heihe River Basin of China. **Wireless Sensor Network**, v. 04, n. 05, p. 138-146, 2012.

## APÊNDICE A – (DIAGRAMA DE CIRCUITOS)

Os diagramas de circuitos das placas de expansão são ilustrados nas páginas 96 e 97. A primeira figura na página 96 mostra a interface de expansão para conectar o mote, as interfaces para os sensores da estação meteorológica, assim como a fonte de energia para o sistema. Já a segunda imagem na página 97 mostra o circuito de interface para o sensor de pressão, que através de transistores é ativado por um pino de saída do *mote*, a fonte de energia para o sistema e a interface de expansão para conectar o *mote*.

Figura 50 – Diagrama de circuito da interface para a estação meteorológica

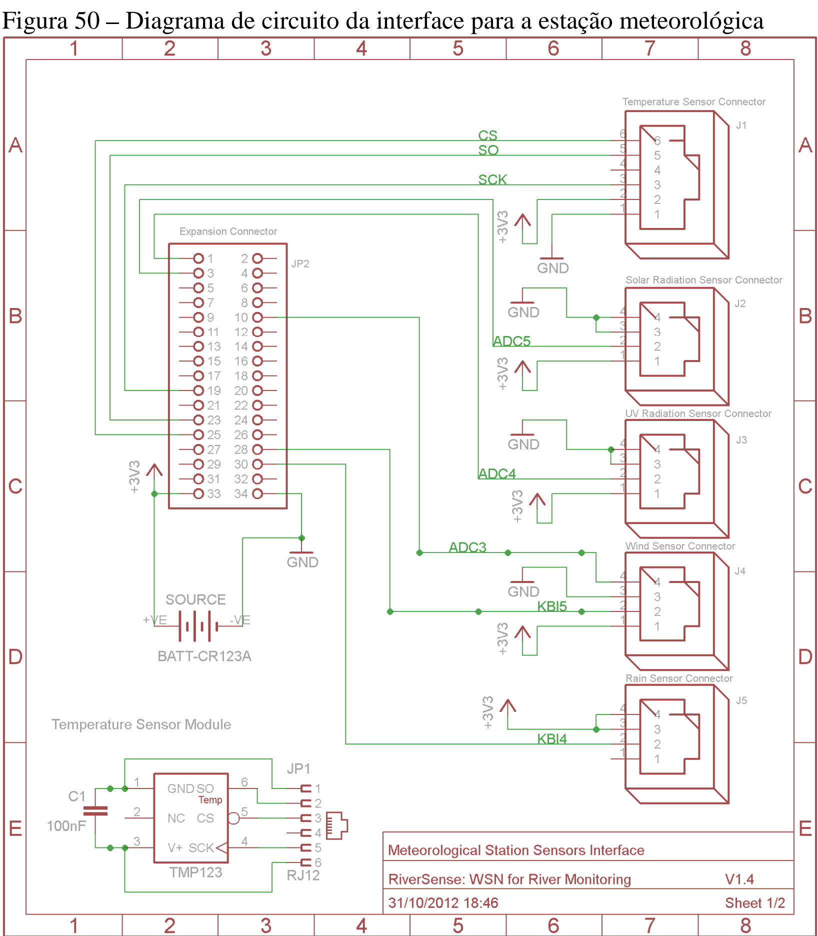
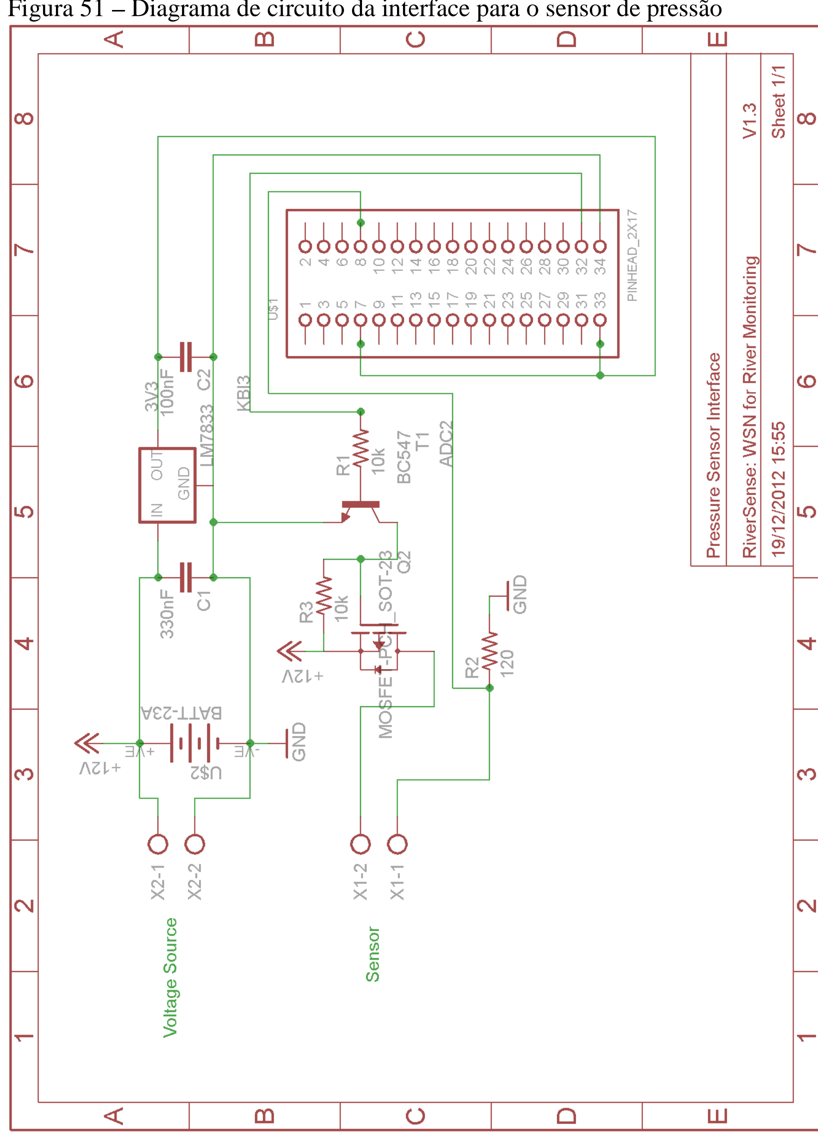


Figura 51 – Diagrama de circuito da interface para o sensor de pressão



## APÊNDICE B – (CÓDIGO FONTE)

O código fonte em anexo se divide em RiverSense Server e RiverSense RSSF. A primeira parte utiliza JavaScript e PHP para implementar a página *Web* (páginas 100 à 104), e Java para a comunicação entre RSSF e Servidor (105 à 108), ambos com conexão MySQL. Já a segunda parte utiliza C/C++ para implementar as diversas interfaces necessárias entre o *mote* e os sensores. O código fonte está dividido em: nodo sincronizador (páginas 109 à 113) , nodo estação de nível (páginas 114 à 117) e nodo estação meteorológica (páginas 118 à 124).

```
<html>
  <head>
    <!--Load the AJAX API-->
    <script type="text/javascript"
src="https://www.google.com/jsapi"></script>
    <script type="text/javascript"
src="http://code.jquery.com/jquery-1.7.2.min.js"></script>
    <script type="text/javascript">

    google.load('visualization', '1.1', {packages:
['corechart', 'controls']});

function drawVisualization() {
        var dashboard = new google.visualization.Dashboard(
             document.getElementById('dashboard'));

         var control = new
google.visualization.ControlWrapper({
           'controlType': 'ChartRangeFilter',
           'containerId': 'control',
           'options': {
             // Filter by the date axis.
             'filterColumnIndex': 0,
             'ui': {
               'chartType': 'LineChart',
               'chartOptions': {
                 'chartArea': {'width': '90%'},
                 'hAxis': {'baselineColor': 'none'}
               },
               // Display a single series that shows the
closing value of the stock.
               // Thus, this view has two columns: the date
(axis) and the stock value (line series).
               'chartView': {
                 'columns': [0, 1]
               },
               // 1 day in milliseconds = 24 * 60 * 60 * 1000 =
86,400,000
               'minRangeSize': 3600000
             }
           },
           // Initial range: 2012-02-09 to 2012-03-20.
           //'state': {'range': {'start': new Date(2012, 6,
21), 'end': new Date(2012, 6, 22)}}
         });

         var chart = new google.visualization.ChartWrapper({
           'chartType': 'LineChart',
           'curveType': 'function',
           'containerId': 'chart',
           'options': {
             // Use the same chart area width as the control
for axis alignment.
             'chartArea': {'height': '80%', 'width': '90%'},
```

```
             'hAxis': {'slantedText': false, 'direction':
+1},//dire磯 do grafico
             'vAxis': {title: "[C]", viewWindowMode: "pretty",
'viewWindow': {'min': 0, 'max': 35}},
             'legend': {'position': 'none'}
           },
           // Convert the first column from 'date' to 'string'.
           'view': {
             'columns': [
               {
                 'calc': function(dataTable, rowIndex) {
                   return dataTable.getFormattedValue(rowIndex,
0);
                 },
                 'type': 'string'
               }, 1]
           }
         });


         var jsonData = $.ajax({
          url: "getData.php",
          dataType:"json",
          async: false
          }).responseText;

        // Create our data table out of JSON data loaded from
server.
        var data = new
google.visualization.DataTable(jsonData);

         dashboard.bind(control, chart);
         dashboard.draw(data);
      }


      google.setOnLoadCallback(drawVisualization);

    // Set a callback to run when the Google Visualization API
is loaded.
    google.setOnLoadCallback(drawChart);

    function drawChart() {
      var jsonData = $.ajax({
          url: "getData.php",
          dataType:"json",
          async: false
          }).responseText;

      // Create our data table out of JSON data loaded from
server.
      var data = new google.visualization.DataTable(jsonData);
```

```
        // Instantiate and draw our chart, passing in some
options.
        var chart = new
google.visualization.LineChart(document.getElementById('chart_d
iv'));
        chart.draw(data, {curveType: "function",
                          lineWidth: 1.5,
                          title: "Temperatura",
                          width: 915, height: 400,
                          vAxis: {title: "[C]", viewWindowMode:
"pretty", viewWindow:{max:40, min:0}},
                          hAxis: {format:'MMM d',
gridlines:{color:"#FFF"}},
                          pointSize: 0,
                          legend: {position:"none"}}
                          );
    }

    function drawChartNivel() {
        var jsonDataNivel = $.ajax({
            url: "getDataNivel.php",
            dataType:"json",
            async: false
            }).responseText;

        // Create our data table out of JSON data loaded from
server.
        var data = new
google.visualization.DataTable(jsonDataNivel);

        // Instantiate and draw our chart, passing in some
options.
        var chartNivel = new
google.visualization.AreaChart(document.getElementById('chart_d
iv_nivel'));
        chartNivel.draw(data, {curveType: "function",
                          lineWidth: 1.5,
                          title: "Nl",
                          width: 915, height: 400,
                          vAxis: {title: "[m]", viewWindowMode:
"pretty", viewWindow:{max:1, min:0}},
                          hAxis: {format:'MMM d',
gridlines:{color:"#FFF"}},
                          pointSize: 0,
                          legend: {position:"none"}}
                          );
    }

    google.setOnLoadCallback(drawChartNivel);

    google.setOnLoadCallback(drawChartPluviometro);

    function drawChartPluviometro() {
        var jsonDataPluviometro = $.ajax({
```

```
            url: "getDataPluviometro.php",
            dataType:"json",
            async: false
            }).responseText;

        // Create our data table out of JSON data loaded from
server.
        var data = new
google.visualization.DataTable(jsonDataPluviometro);

        // Instantiate and draw our chart, passing in some
options.
        var chartPluviometro = new
google.visualization.LineChart(document.getElementById('chart_d
iv_pluviometro'));
        chartPluviometro.draw(data, {curveType: "function",
                              lineWidth: 1.5,
                              title: "Pluviosidade",
                              width: 915, height: 400,
                              vAxis: {title: "[mm]", viewWindowMode:
"pretty", viewWindow:{max:40, min:0}},
                              hAxis: {format:'MMM d',
gridlines:{color:"#FFF"}},
                              pointSize: 0,
                              legend: {position:"none"}}
                              );
     }

     google.setOnLoadCallback(drawChartRadiacaoSolar);

     function drawChartRadiacaoSolar() {
        var jsonDataRadiacaoSolar = $.ajax({
            url: "getDataRadiacaoSolar.php",
            dataType:"json",
            async: false
            }).responseText;

        // Create our data table out of JSON data loaded from
server.
        var data = new
google.visualization.DataTable(jsonDataRadiacaoSolar);

        // Instantiate and draw our chart, passing in some
options.
        var chartRadiacaoSolar = new
google.visualization.LineChart(document.getElementById('chart_d
iv_radiacao_solar'));
        chartRadiacaoSolar.draw(data, {curveType: "function",
                              lineWidth: 1.5,
                              title: "RadiacaoSolar",
                              width: 915, height: 400,
                              vAxis: {title: "[W/m]", viewWindowMode:
"pretty", viewWindow:{max:40, min:0}},
```

```
                                hAxis: {format:'MMM d',
gridlines:{color:"#FFF"}},
                                pointSize: 0,
                                legend: {position:"none"}}
                                );
     }

     </script>
   </head>

   <body>
     <!--Div that will hold the pie chart-->
     <div id="dashboard">
         <div id="chart" style='width: 915px; height:
300px;'></div>
         <div id="control" style='width: 915px; height:
50px;'></div>
     </div>
     <div id="chart_div">
     </div>
     <div id="chart_div_nivel">
     </div>
     <div id="chart_div_pluviometro">
     </div>
     <div id="chart_div_radiacao_solar">
     </div>
   </body>
</html>
```

```
package riverSense;

import ...;

public class SerialConnection implements
SerialPortEventListener,
                     CommPortOwnershipListener {
    private SerialDemo parent;

    private TextArea messageAreaOut;
    private TextArea messageAreaIn;
    private SerialParameters parameters;
    private OutputStream os;
    private InputStream is;
    private KeyHandler keyHandler;

    private CommPortIdentifier portId;
    private SerialPort sPort;

    private boolean open;

    OutputStream osd;

public SerialConnection(SerialDemo parent,
                SerialParameters parameters,
                TextArea messageAreaOut,
                TextArea messageAreaIn){}

public void openConnection() throws SerialConnectionException{}

public void closeConnection(){}

public void serialEvent(SerialPortEvent e) {
    // Create a StringBuffer and int to receive input data.
    StringBuffer inputBuffer = new StringBuffer();
    int newData = 0;
    Calendar cal = Calendar.getInstance();

    //BufferedWriter bw;

    // Determine type of event.
    switch (e.getEventType()) {

        // Read data until -1 is returned. If \r is received
substitute
        // \n for correct newline handling.
        case SerialPortEvent.DATA_AVAILABLE:
            while (newData != -1) {
                try {
                    newData = is.read();
                if (newData == -1) {
                break;
                }
                if ('\r' == (char)newData) {
```

```
                inputBuffer.append('\n');
                }else if('b' == (char)newData){
                    //manda escrever uma string na saida.
                    String s = "D";
                    //DateFormat dateFormat = new
SimpleDateFormat("ddMMyyHHmmss");
                    s += (cal.getTimeInMillis() / 1000);
                    System.out.println(s);

                    for(int i=0; i<s.length(); i++){
                        os.write(s.charAt(i));
                    }
                }else if('w' == (char)newData){
                    //inicia armazenamento da mensagem de
dados.
                    //reseta o buffer da mensagem
                    inputBuffer.delete(0,
inputBuffer.length());

                }else if('x' == (char)newData){
                    //finaliza armazenamento da mensagem de
dados
                    //coloca mensagem em uma string
                    String message = new String(inputBuffer);
                    //Faz o parser da mensagem e devolve um
vetor dos dados
                    String[] parsedMessage =
ProtocolParser.messageParser(message);
                    String messageType = parsedMessage[0];

                    //De acordo com o tipo de mensagem
direciona para o m鷂do de inserir
                    switch (messageType){
                        //Insere dados no banco de dados
                        case "0":
                            //w0
                            int messageSizeSink =
Integer.parseInt(parsedMessage[1]);
                            int messageErrorsSink =
Integer.parseInt(parsedMessage[2]);
                            double gprsToA =
Double.parseDouble(parsedMessage[3]);
                            //-DATETIME- in Seconds from Node
                            long dateSeconds =
Long.parseLong(parsedMessage[4]);
                            java.sql.Timestamp currentTimestamp
= new java.sql.Timestamp((dateSeconds)*1000);
                            int batterySink =
Integer.parseInt(parsedMessage[5]);


//[messageType][messageSize][messageErrors][DateTime][Battery]
```

```
ConnectionParser.insertSink(messageSizeSink, messageErrorsSink,
gprsToA, currentTimestamp, batterySink);

                            messageType = "";
                            break;

                        case "1":

                            int messageSizeStation =
Integer.parseInt(parsedMessage[1]);
                            int messageErrorsStation =
Integer.parseInt(parsedMessage[2]);
                            float temperature =
Float.parseFloat(parsedMessage[3]);
                            float rain =
Float.parseFloat(parsedMessage[4]);
                            rain = (float) (rain * 0.2);
                            float solarRad =
Float.parseFloat(parsedMessage[5]);
                            float uvRad =
Float.parseFloat(parsedMessage[6]);
                            uvRad = (float) (uvRad * 0.0054);
                            float windDir =
Float.parseFloat(parsedMessage[7]); /*
Integer.parseInt(parsedMessage[7]); */
                            float windSpeed =
Float.parseFloat(parsedMessage[8]); /*
Integer.parseInt(parsedMessage[8]);*/
                            int batteryStation =
Integer.parseInt(parsedMessage[9]);
                            java.sql.Timestamp
timestampStation;
                            long dateSecondsStation =
Long.parseLong(parsedMessage[10]);
                            if (dateSecondsStation <
1362266517){
                                timestampStation = new
java.sql.Timestamp(cal.getTimeInMillis());
                            }else{
                                timestampStation = new
java.sql.Timestamp((dateSecondsStation)*1000);
                            }//----------

//[messageType][messageSize][messageErrors][temperature][rain][
solarRadiation][uvIndex][windDirection][windSpeed][Battery][Dat
eTime]
ConnectionParser.insertStation(messageSizeStation,
messageErrorsStation, timestampStation, batteryStation, rain,
temperature, solarRad, uvRad, windSpeed, windDir);

                            messageType = "";
                            break;
```

```
                        case "2":
                            //w2;
//[messageType][messageSize][messageErrors][RiverLevel][Battery
][DateTime]
                            int messageSizeRiverLevel =
Integer.parseInt(parsedMessage[1]);
                            int messageErrorsRiverLevel =
Integer.parseInt(parsedMessage[2]);
                            float riverLevel =
Float.parseFloat(parsedMessage[3]);
                            //float riverLevel = 1;
                            int batteryRiverLevel =
Integer.parseInt(parsedMessage[4]);

                            //-DATETIME-=
                            java.sql.Timestamp
timestampRiverLevel;
                            long dateSecondsRiverLevel =
Long.parseLong(parsedMessage[5]);
                            if (dateSecondsRiverLevel <
1362266517){
                                timestampRiverLevel = new
java.sql.Timestamp(cal.getTimeInMillis());
                            }else{
                                timestampRiverLevel = new
java.sql.Timestamp((dateSecondsRiverLevel)*1000);
                            }

ConnectionParser.insertRiverLevel(messageSizeRiverLevel,
messageErrorsRiverLevel, riverLevel, batteryRiverLevel,
timestampRiverLevel);

                            messageType = "";
                            break;
                    }

                }else {
                    inputBuffer.append((char)newData);
                }
                } catch (IOException ex) {
                    System.err.println(ex);
                    return;
                }
            }

        //Append received data to messageAreaIn.
        messageAreaIn.append(new String(inputBuffer));
        break;
        // If break event append BREAK RECEIVED message.
        case SerialPortEvent.BI:
        messageAreaIn.append("\n--- BREAK RECEIVED ---\n");
    }
}
```

```
#include <machine.h>
#include <alarm.h>
#include <sensor.h>
#include <mach/mc13224v/buck_regulator.h>
#include <battery.h>
#include <utility/string.h>
#include "ieee1588.h"
#include "ieee1588_ptp.h"
#include <rtc.h>
#include <string.h>
#include <chronometer.h>

__USING_SYS

OStream cout;
NIC * nic;
UART uart(9600, 8 , 0 , 1);
RTC rtc;
Chronometer chron;

//PTP
IEEE1588_PTP * _ptp;
IEEE_1588 * _protocol;
bool initialized;
Thread * gprs;
Thread * sync;
bool interrupt_gprs;
NIC::Protocol prot;
NIC::Address src;

const unsigned char SINK_ID = 0x01;
int _messageType;
char * date;
char msg[100];
bool dateRequestFlag = false;
bool received_davis, received_level;

class Header {

public:
    int nodeId;
    unsigned int messageSize;
    int messageType;
    int messageErrors;
    unsigned long timeStamp;
} __attribute__((packed));

class Message_Station {

public:
    Header head;
    int battery;
    int rain;
    float temperature;
```

```
    float solarRadiation;
    int uvRadiation;
    int windSpeed;
    float windDirection;
} __attribute__((packed));

class Message_RiverLevel {

public:
    Header head;
    int battery;
    float riverLevel;
} __attribute__((packed));

Header * _header;
Message_Station * message;
Message_RiverLevel * message_river;

IEEE1588_PTP * startPTP()
{
    _ptp = new IEEE1588_PTP();
    _ptp->sync_interval = 11000000; //microsecond
    _ptp->clock_stratum = 4; //Master
    _ptp->_state = IEEE_1588::PTP_MASTER;
    _ptp->initialized = 0;
    return _ptp;

}

//GPRS - UART send function
void send(const char * s){
    while(*s != '\0')
        uart.put(*s++);
}

void dateRequest(){
    dateRequestFlag = true;
}

int sink() {

    if(dateRequestFlag == true){
        dateRequestFlag = false;
        uart.put('b');
        chron.start();
        //Alarm::delay(20000000); //2s
        int i;

        if(uart.get() == 'D'){
            i=0;
            while(i < 10){
                msg[i] = uart.get();
                i++;
            }
```

```
                chron.stop();
                int timestamp = atoi(msg);
                //cout << "Server TimeStamp: "  << timestamp <<
endl;
                rtc.setTimeStamp(timestamp);
            }
            int battery = Battery::sys_batt().sample();
            long timeStamp = rtc.getTimeStamp();
            cout << "w0;" << "0;" << "0;" << chron.read() << ";" <<
timeStamp << ";" << battery << "x"<< endl;
            chron.reset();
        }

        char data[nic->mtu()];
        nic->receive(&src, &prot, data, sizeof(data));

        _header = (Header*) data;

        _messageType = _header->messageType;

        switch (_messageType) {

//[messageType][messageSize][messageErrors][DateTime][Battery]
        //w0;10;15;592683315;3200x

        case 1: {
            if(!received_davis){
                message = (Message_Station *)data;

//[messageType][messageSize][messageErrors][temperature][rain][
solarRadiation][uvIndex][windDirection][windSpeed][battery][dat
eTime]
                //w1;3;5;25;3;150;2;55;15;3000;1359305358x
                cout << "w1;" << message->head.messageSize << ";"
<< message->head.messageErrors << ";" << message->temperature
<< ";" << message->rain << ";" << message->solarRadiation <<
";" << message->uvRadiation << ";" << message->windDirection <<
";" << message->windSpeed << ";" << message->battery << ";" <<
message->head.timeStamp << "x" << endl;
                received_davis = true;
            }
        }break;

        case 2: {
            if(!received_level){
                message_river = (Message_RiverLevel*)data;

//[messageType][messageSize][messageErrors][RiverLevel][Battery
][DateTime]
                //w2;3;5;18.500;2800;1359305358x
                cout << "w2;" << message_river->head.messageSize <<
";" << message_river->head.messageErrors << ";" <<
message_river->riverLevel << ";" << message_river->battery <<
";" << message_river->head.timeStamp << "x" << endl;
```

```
                received_level = true;
            }
        }break;

        }
        free(data);
        _messageType = 0;
        free(_header);
        return 0;
}


int synchronizeMotes(){
        if(!initialized){
            initialized = true;
            _protocol->startProtocol();
        }else{
            _protocol->synchronizing();
        }
        return 0;
}

int syncMotes(){
        _protocol->setInterrupt(false);
        if(interrupt_gprs){
            cout << "Synchronizing\n";
            synchronizeMotes();
        }
        else{
            gprs->join();
        }
        return 0;
}

void interrupt(){
        if(interrupt_gprs){
            interrupt_gprs = false;
            received_davis = false;
            received_level = false;
        }
        else{
            if(received_davis == true && received_level == true)
                interrupt_gprs = true;
        }
}

int triGPRS(){
        while(true){
            if(!interrupt_gprs){
                sink();
            }else{
                gprs->yield();
            }
        }
```

```
    return 0;
}

int main() {

    kout << "SINK GPRS\n";
    rtc.setTimeStamp(1362266517);

    nic = new NIC();
    //INICIALIZADO PTP
    _ptp = startPTP();
    _protocol = new IEEE_1588(_ptp);
    _protocol->setNIC(nic);
    _protocol->initializeServo();

    dateRequestFlag = true;

    Function_Handler handler_Date(&dateRequest);
    Alarm alarm_Date(180000000, &handler_Date, -1); // 3
minutos

    Function_Handler handler_ptp(&interrupt);
    Alarm alarm_ptp(33000000, &handler_ptp, -1); //33 segundos
/ alterar para 3 minutos

    gprs = new Thread(&triGPRS);
    synchronizeMotes();

    while(true);
    //  while(true){
    //      sink();
    //  }
    return 0;
}
```

```
#include <ic.h>
#include <cpu.h>
#include <machine.h>
#include <alarm.h>
#include <adc.h>
#include <arch/arm7/tsc.h>
#include <utility/handler.h>
#include <mach/mc13224v/buck_regulator.h>
#include <battery.h>
#include <rtc.h>
#include "ieee1588.h"
#include "ieee1588_ptp.h"

#include <string.h>

__USING_SYS

typedef Machine::IO IO;

OStream cout;
ADC adc(ADC::SINGLE_ENDED_ADC2);
bool flag = false;
bool sensorOn = false;
static int iterations = Alarm::INFINITE; //Data Handler
static Alarm::Microsecond time = 10000000; //10 seconds Data
Handler - //changeable
volatile unsigned int data0_mask = 0x0;
bool sent;
long timestamp_last_received;
RTC rtc;
//Threads
Thread * davis;
Thread * sync;
bool interrupt_hecops;

//PTP
IEEE1588_PTP * _ptp;
IEEE_1588 * _protocol;
bool initialized;


//melhorar com enum //tipo de mensagem
int mensagem_RiverLivel = 2;

void turn_sensor_on() {
    data0_mask |= (1 << 25);
    CPU::out32(IO::GPIO_DATA0, data0_mask);
}

void turn_sensor_off() {
    data0_mask &= ~(1 << 25);
    CPU::out32(IO::GPIO_DATA0, data0_mask);
}
```

```
//Periodic Amount Data Handler Function
void periodic_read() {
    //if(rtc.getTimeStamp() - timestamp_last_received > 120){
        cout << "Enable Sensor...\n";
        //turn_sensor_on();
        //sensorOn = true;
    //}
    flag = true;

}

float convert_level(int value) {
    float level = (float) ((0.0624f * (value)) - 71.79f);
    //turn_sensor_off();
    return level;
}

class Header {

public:
    int nodeId;
    unsigned int messageSize;
    int messageType;
    int messageErrors;
    unsigned long timeStamp;
} __attribute__((packed));

class Message_RiverLevel {

public:
    Header head;
    int battery;
    float riverLevel;
} __attribute__((packed));

Message_RiverLevel _message;

IEEE1588_PTP * startPTP()
{
    int i;
    // Iniciando e Configurando o PTP Clock
    _ptp = new IEEE1588_PTP();
    _ptp->clock_stratum = 100; //listener 100
    kout << "Iniciado com sucesso\n";
    return _ptp;
}

int synchronizeMotes(){
    if(!initialized){
        initialized = true;
        _protocol->startProtocol();
    }else{
        _protocol->synchronizing();
    }
```

```
    return 0;
}

//WSN Network
NIC * nic;

void send_message(){
    if(flag){
        flag = false;
        Alarm::delay(1000000); //100ms
        int adcValue = adc.get();
        _message.riverLevel = convert_level(adcValue);
        //_message.riverLevel = 15;

        _message.head.timeStamp = rtc.getTimeStamp();
        _message.battery = Battery::sys_batt().sample();
        _message.head.messageErrors = 0;
        _message.head.messageSize = sizeof(_message);

        char * msg;
        msg = (char *)&(_message);

        int ret;
        for(int i=0;i < 10; i++){
            ret = nic->send(NIC::BROADCAST, (NIC::Protocol) 1,
msg, sizeof(Message_RiverLevel));
        }
        if(ret == 11)
            sent = true;
        cout << "OK: " << ret << endl;
        free(msg);
    }

}

int triDAVIS(){
    while(true){
        if(!interrupt_hecops){
            send_message();
        }else{
            davis->yield();
        }
    }
    return 0;
}

int syncMotes(){
    _protocol->setInterrupt(false);
    while(true){
        if(interrupt_hecops){
            cout << "Synchronizing\n";
            synchronizeMotes();
        }
        else{
```

```
                sync->yield();
            }
        }
        return 0;
}

void interrupt(){
        if(interrupt_hecops){
            interrupt_hecops = false;
            _protocol->setInterrupt(true);
        }
        else{
            if(sent){
                interrupt_hecops = true;
                _protocol->setInterrupt(false);
            }
        }
}

int main() {
        //Periodic Handler Config
        Function_Handler handler(&periodic_read);
        Alarm alarm(time, &handler, iterations);

        //configurar KBI3 para output
        CPU::out32(IO::GPIO_PAD_DIR0, 0u // all as input, but:
        | (1 << 25) // GPIO_25: Saída
        );

        CPU::int_enable();

        turn_sensor_on();

        //MSG
        _message.head.nodeId = 2;
        _message.head.messageType = mensagem_RiverLivel;

        //WSN Network
        nic = new NIC();
        _ptp = startPTP();
        _protocol = new IEEE_1588(_ptp);
        _protocol->setNIC(nic);
        _protocol->initializeServo();

        Function_Handler handler_ptp(&interrupt);
        Alarm alarm_ptp(33000000, &handler_ptp, -1);

        davis = new Thread(&triDAVIS);
        sync = new Thread(&syncMotes);
        sync->join();

        while(true);
        return 0;
}
```

```
#include <ic.h>
#include <cpu.h>
#include <machine.h>
#include <alarm.h>
#include <adc.h>
#include <mach/mc13224v/spi.h>
#include <arch/arm7/tsc.h>
#include <utility/handler.h>
#include <mach/mc13224v/buck_regulator.h>
#include <battery.h>
#include <rtc.h>
#include "ieee1588.h"
#include "ieee1588_ptp.h"
#include <math.h>

#include <string.h>

__USING_SYS

typedef Machine::IO IO; //External Interruptions KBI4/KBI5

int mensagem_estacao = 1;

OStream cout; //General

//Threads
Thread * davis;
Thread * sync;
bool interrupt_hecops;
RTC rtc;
bool sent;
long timestamp_last_received;

//PTP
IEEE1588_PTP * _ptp;
IEEE_1588 * _protocol;
bool initialized;

class Header {

public:
    int nodeId;
    unsigned int messageSize;
    int messageType;
    int messageErrors;
    unsigned long timeStamp;
} __attribute__((packed));

class Message_Station {

public:
    Header head;
    int battery;
    int rain;
```

```
    float temperature;
    float solarRadiation;
    int uvRadiation;
    int windSpeed;
    float windDirection;
} __attribute__((packed));

Message_Station _message;

int i, j; //Rain Sensor and Wind Speed Sensor
static int iterations = Alarm::INFINITE; //Data Handler
static Alarm::Microsecond time = 13000000; //12 seconds Data
Handler - //changeable



IEEE1588_PTP * startPTP()
{
    int i;
    // Starting PTP Clock
    _ptp = new IEEE1588_PTP();
    _ptp->clock_stratum = 255; //slave
    kout << "Iniciado com sucesso\n";
    return _ptp;
}

int synchronizeMotes(){
    if(!initialized){
        initialized = true;
        _protocol->startProtocol();
    }else{
        _protocol->synchronizing();
    }
    return 0;
}


MC13224V_SPI * _spi; //Temperature Sensor - SPI
ADC adc3(ADC::SINGLE_ENDED_ADC3); //Wind Direction Sensor ADC3
float direction; //Wind Direction Sensor
ADC adc4(ADC::SINGLE_ENDED_ADC4); //UV Sensor ADC4
int uvIndex; //UV Sensor
ADC adc5(ADC::SINGLE_ENDED_ADC5); //Solar Sensor ADC5
float solarRadiation; //Solar Sensor
bool data = false;

int t;

//WSN Network
NIC * nic;

// Rain and Wind Sensor - Interruption Handler Function
KBI4/KBI5
```

```
void kbi_handler() {
    //Rain Sensor
    if (CPU::in32(IO::CRM_STATUS) & 0x00000010) { //KBI4
        i = i + 1;
        cout << "KBI4 ";
        cout << i;
        cout << " handled\n";

        CPU::out32(IO::CRM_STATUS, CPU::in32(IO::CRM_STATUS)
//Needs to clear flag
        | (1 << 4) //EXT_WU_EVT
        );

        //Wind Speed Sensor
    } else if (CPU::in32(IO::CRM_STATUS) & 0x00000020) { //KBI5
        j = j + 1;
        cout << "KBI5 ";
        cout << j;
        cout << " handled\n";

        CPU::out32(IO::CRM_STATUS, CPU::in32(IO::CRM_STATUS)
//Needs to clear flag
        | (2 << 4) //EXT_WU_EVT
        );
    }
}

float getRain(){
    if(i>0){
        return i*0.2f;
    }else
        return 0.000;
}

//Wind Direction Sensor - ADC3 Reader Function
float getWindDirection() {
    int value = adc3.get();
    direction = (float) ((0.0879 * (value)) - 0.000000000001f);
    return direction;
}

//Temperature Sensor - SPI Reader Function
int read() {
    MC13224V_SPI::spiErr_t e;
    unsigned int data[] = { 0 };
    e = _spi->SPI_ReadSync(data);
    return data[0];
}

float getTemperature(){
    t = (0x0000FFFF & (read())) >> 3;
    float temperature = (0.0625 * (t));
    return temperature;
}
```

```
//Solar Sensor - ADC5 Reader Function
float getSolarRadiation() {
    int value = adc5.get();
    solarRadiation = (float) ((0.4826f * (value)) +
0.000000000002f); // W/mÂ²
    return solarRadiation;
}

//UV Sensor - ADC4 Reader Function
float getUV() {
    int value = adc4.get();
    uvIndex = (float)(0.0054f * (value));
    return value;
}
//General - Periodic Amount Data Handler Function -
void periodic_amount() {

    //TimeStamp - UNIX
    _message.head.timeStamp = rtc.getTimeStamp();

    //Status Battery
    _message.battery = Battery::sys_batt().sample();

    //Rain Sensor - millimeters
    _message.rain = getRain;

    //Temperature Sensor - Celsius
    _message.temperature = getTemperature();

    //Solar Sensor
    _message.solarRadiation = getSolarRadiation();

    //UV Sensor
    _message.uvRadiation = getUV();

    //Wind Speed Sensor
    _message.windSpeed = j;

    //Wind Direction Sensor  - converter isso em N/S/L/O
    _message.windDirection = getWindDirection();

    i = j = 0;

    data = true;
}

void send_message(){
    if(data){

        _message.head.messageErrors = 0;
        _message.head.nodeId = 3;
        _message.head.messageType = mensagem_estacao;
        _message.head.messageSize = sizeof(_message);
```

```
        char * msg;
        msg = (char *)&(_message);

        int ret;
        for(int i=0;i < 5; i++){
            ret = nic->send(NIC::BROADCAST, (NIC::Protocol) 1,
msg, sizeof(Message_Station));
        }
        if(ret == 11)
            sent = true;
        cout << "OK: " << ret << endl;
        free(msg);
        data = false;
    }

}

int triDAVIS(){
    while(true){
        if(!interrupt_hecops){
            send_message();
        }else{
            davis->yield();
        }
    }
    return 0;
}

int syncMotes(){
    _protocol->setInterrupt(false);
    while(true){
        if(interrupt_hecops){
            cout << "Synchronizing\n";
            synchronizeMotes();
        }
        else{
            sync->yield();
        }
    }
    return 0;
}

void interrupt(){
    if(interrupt_hecops){
        interrupt_hecops = false;
        _protocol->setInterrupt(true);
    }
    else{
        if(sent){
            interrupt_hecops = true;
            _protocol->setInterrupt(false);
        }
    }
}
```

```
int main() {
    //General - Periodic Handler Config

    //Rain & Wind Sensor - KBI Interruption Handler Config -
KBI4 & KBI5
    IC::int_vector(IC::IRQ_CRM, kbi_handler);
    IC::enable(IC::IRQ_CRM);

    //KBI Interruption Config - KBI4/KBI5
    CPU::out32(IO::GPIO_PAD_DIR0, CPU::in32(IO::GPIO_PAD_DIR0)
& ~(3 << 26) // GPIO_26 and GPIO_27 as input
    );
    CPU::out32(IO::GPIO_FUNC_SEL1,
CPU::in32(IO::GPIO_FUNC_SEL1) & ~(15 << 20) // GPIO_26 and
GPIO_27 are at default function
    );
    CPU::out32(IO::GPIO_PAD_PU_SEL0,
            CPU::in32(IO::GPIO_PAD_PU_SEL0) & ~(1 << 27) //
GPIO_27 as weak pull-up
    );
    CPU::out32(IO::GPIO_DATA_SEL0,
CPU::in32(IO::GPIO_DATA_SEL0) & ~(3 << 26) // GPIO_26 and
GPIO_27 data comes from the pad
    );
    CPU::out32(IO::CRM_WU_CNTL, CPU::in32(IO::CRM_WU_CNTL) |
(0x3 << 4) // EXT_WU_EN [3:0]0x4 << 4
            | (0x3 << 8)  // EXT_WU_EDGE 0x4 << 8
            | (0x3 << 12) // EXT_WU_POL 0x4 << 12
            | (0x3 << 20) // EXT_WU_IEN 0x4 << 20
    );

    CPU::int_enable();

    //Temperature Sensor - SPI Configuration
    MC13224V_SPI::spiConfig_t config;
    MC13224V_SPI::spiDelay delay = MC13224V_SPI::delayZero;
    MC13224V_SPI::spiSdoInactive doInactive =
MC13224V_SPI::Mosi_out_00;
    MC13224V_SPI::spiClockPol clkPol = MC13224V_SPI::polOne;
    MC13224V_SPI::spiClockPhase clkPha = MC13224V_SPI::phaOne;
    MC13224V_SPI::spiMisoPhase misoPhase = MC13224V_SPI::Same;
    MC13224V_SPI::spiClockFreq clkFreq = MC13224V_SPI::_1_5MHz;
    MC13224V_SPI::spi3Wire wire = MC13224V_SPI::Wire4;
    MC13224V_SPI::spiMode mode = MC13224V_SPI::Master;
    MC13224V_SPI::spiSlaveSelectSetup ssSetup =
MC13224V_SPI::Auto_Active_Low;

    config.Setup.Bits.SsSetup = ssSetup; //table 15-10
automatic, low
    config.Setup.Bits.SsDelay = delay;
    config.Setup.Bits.SdoInactive = doInactive;
    config.Setup.Bits.ClockPol = clkPol; //ok CPOL
    config.Setup.Bits.ClockPhase = clkPha; //ok CPHA
```

```
    config.Setup.Bits.MisoPhase = misoPhase; //UPDATE ON
RISE???
    config.Setup.Bits.ClockFreq = clkFreq; //24/n
    config.Setup.Bits.Mode = mode;  //low = masterMode
    config.Setup.Bits.S3Wire = wire; //4Wire
    config.ClkCtrl.Bits.DataCount = 16; //0b011000;//32-8;
    config.ClkCtrl.Bits.ClockCount = 16; //0b0011000;

    _spi = new MC13224V_SPI(config);

    //WSN Network
    nic = new NIC();
    _ptp = startPTP();
    _protocol = new IEEE_1588(_ptp);
    _protocol->setNIC(nic);
    _protocol->initializeServo();

    data = true;
    Function_Handler handler(&periodic_amount);
    Alarm alarm(time, &handler, iterations);

    Function_Handler handler_ptp(&interrupt);
    Alarm alarm_ptp(33000000, &handler_ptp, -1);

    davis = new Thread(&triDAVIS);
    sync = new Thread(&syncMotes);
    sync->join();

    while(true);
    return 0;
}
```

# APÊNDICE C – (TUTORIAL DE SETUP)

1. Configuração do servidor
   a) Para suportar a página web é necessário um serviço Apache que suporte PHP e MySQL. Caso não o possua, faça o download para Windows na página http://www.wampserver.com/en/ ou para Linux na página http://sourceforge.net/projects/xampp/.
   b) Para executar o RiverSense Server é preciso possuir a IDE Eclipse com um JDK (Java Development Kit) instalado.
   c) Instalar o TSG Server. Caso não o possua, faça o download através do servidor *gse.ufsc.br* no diretório */home/gse_work_release/MS/Cladio/v1.0*

2. Configuração e execução da rede de sensores sem fio
   a) Fazer download do código fonte em https://bitbucket.org/poliveirax/riversense
   b) Conectar o EPOSMote na porta USB.
   c) Acessar o diretório raiz do código fonte através do terminal.
   d) Executar o script *doit_flash.sh* para iniciar o processo de compilação e gravação do firmware, explicitando o arquivo fonte encontrado na pasta *app*. (e.g., *./doit.sh level_sensor, ./doit.sh davis_station, ./doit.sh sink_gprs*).
   e) Desconectar o módulo principal do EPOSMote e conectar na placa de interface (e.g., Estação de Nível, Estação Meteorológica, Nó Sincronizador).
   f) Para regravar a flash do EPOSMote é necessario colocar dois jumpers nos pinos 2 e 3 do conectores J4 E J5.

3. Inicialização do sistema
   a. Conecte os sensores ou módulo GPRS às respectivas placas de interface com o EPOSMote.
   b. Inicialize o TSGServer. Ele possibilita realizar a conexão do software Java no servidor com a RSSF.
   c. Inicialize o RiverSense Server para iniciar a conexão com o TSGServer e banco de dados, assim como receber dados da RSSF.
   d. Com o firmware gravado em todos os nós de interesse da RSSF, conecte uma fonte de energia e aperte o botão de *reset* do EPOSMote de cada um dos nós para iniciar a operação do sistema.