UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

THIAGO HENRIQUE DEICKE

SISTEMA DE MONITORAMENTO EM TEMPO REAL DOS NÍVEIS ACÚSTICOS URBANOS BASEADO EM REDES DE SENSORES SEM FIO

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2016

THIAGO HENRIQUE DEICKE

SISTEMA DE MONITORAMENTO EM TEMPO REAL DOS NÍVEIS ACÚSTICOS URBANOS BASEADO EM REDES DE SENSORES SEM FIO

Trabalho de Conclusão de Curso como requisito parcial à obtenção do título de Bacharel em Engenharia de Computação, do Departamento Acadêmico de Informática da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Gustavo Weber Denardin

PATO BRANCO

2016

MINISTÉRIO DA EDUCAÇÃO
Universidade Tecnológica Federal do Paraná
Câmpus Pato Branco
Departamento Acadêmico de Informática
Curso de Engenharia de Computação

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

TERMO DE APROVAÇÃO

Às 8 horas e 30 minutos do dia 04 de julho de 2016, na sala V008, da Universidade Tecnológica Federal dão Paraná, Câmpus Pato Branco, reuniu-se a banca examinadora composta pelos professores Gustavo Weber Denardin (orientador), Fabio Luiz Bertotti e Bruno Cesar Ribas para avaliar o trabalho de conclusão de curso com o título **Sistema de monitoramento em tempo real dos níveis acústicos urbanos baseado em redes de sensores sem fio**, do aluno **Thiago Henrique Deicke**, matrícula 01159178, do curso de Engenharia de Computação. Após a apresentação o candidato foi arguido pela banca examinadora. Em seguida foi realizada a deliberação pela banca examinadora que considerou o trabalho aprovado.

_____
Gustavo Weber Denardin
Orientador (UTFPR)

_____
Fabio Luiz Bertotti
(UTFPR)

_____
Bruno Cesar Ribas
(UTFPR)

_____
Beatriz Terezinha Borsoi
Coordenador de TCC

_____
Pablo Gauterio Cavalcanti
Coordenador do Curso de
Engenharia de Computação

A Folha de Aprovação assinada encontra-se na Coordenação do Curso.

AGRADECIMENTOS

Agradeço aos meus pais Ademir José Deicke e Judite Lorenzon Deicke, e minha irmã Maiara Vitória Deicke, por sempre estarem ao meu lado, me apoiando e incentivando durante esta jornada, e por nunca medirem esforços para permitir que este sonho se realizasse.

Aos meus amigos e colegas, por todos os momentos e aprendizados compartilhados.

Aos profissionais de excelência que lecionam no curso de Engenharia de Computação no campus Pato Branco, em especial às Professoras Dra. Beatriz Terezinha Borsoi e Dra. Kathya Silvia Collazos Linares, por todo o apoio e ajuda que ofereceram durante estes anos.

Ao meu orientador Professor Dr. Gustavo Weber Denardin pelo apoio, disposição e conhecimentos repassados durante este trabalho e outras disciplinas.

He who would learn to fly one day must first learn to stand and walk and run and climb and dance; one cannot fly into flying.

Friedrich W. Nietzsche

RESUMO

DEICKE, Thiago Henrique. Sistema de monitoramento em tempo real dos níveis acústicos urbanos baseado em redes de sensores sem fio. 2016. 78f. Trabalho de Conclusão de Curso de bacharelado em Engenharia de Computação - Universidade Tecnológica Federal do Paraná. Pato Branco, 2016.

Este trabalho de conclusão de curso consiste no desenvolvimento de um sistema de monitoramento em tempo real dos níveis acústicos em ambientes urbanos, auxiliando primariamente na detecção de fontes de poluição sonora. O projeto é composto por uma rede sem fio de objetos inteligentes, capazes de se comunicar entre si através de protocolos abertos de comunicação; um servidor de dados que armazena os dados provenientes da rede de sensores sem fio; e uma página *web*, que mostra as informações em formato de mapa sonoro. A rede sem fio tem sua camada de enlace baseada no padrão IEEE 802.15.4 de comunicação, e utiliza o protocolo IPv6 na camada de rede, juntamente com a camada de adaptação para dispositivos com recursos restritos 6LoWPAN e o protocolo RPL para a topologia de rede e roteamento de pacotes. A rede é composta por nós sensores espalhados em diferentes pontos, que têm sua posição determinada através de GPS. Através de um microfone e um circuito de pré-amplificação, estes nós sensores realizam a medição da intensidade sonora do ambiente em que estão inseridos. Cada um dos nós sensores envia os dados obtidos através de pacotes UDP pela rede, até que cheguem ao coordenador da rede. O coordenador da rede se responsabiliza por enviar os dados, através da rede celular, para um servidor de dados na Internet. Neste servidor, todos os dados são armazenados num banco de dados, o qual mantém as informações de todo o período de vida da rede. O banco de dados é continuamente consultado por um *script* da página *web*, que apresenta visualmente as informações atuais no formato de mapa sonoro. Os resultados obtidos até o fim deste trabalho mostraram que a solução utilizada neste sistema se mostra funcional, porém, exige grande parte da memória RAM dos microcontroladores. Os dados trafegam dentro da rede de sensores sem fio de forma efetiva através dos protocolos de Internet.

Palavras-chave: Rede de sensores sem fio. Poluição sonora. Protocolos de Internet. 6LoWPAN. RPL.

ABSTRACT

DEICKE, Thiago Henrique. Real-time monitoring system of urban sound levels based on wireless sensor networks. 2016. 78f. Trabalho de Conclusão de Curso de bacharelado em Engenharia de Computação - Universidade Tecnológica Federal do Paraná. Pato Branco, 2016.

This course conclusion work consists of the development of a real-time monitoring system of sound levels in urban environments, helping primarily in the detection of sound pollution sources. The project is composed of a wireless network of smart objects, capable of communicating between each other through open communication protocols; a data server which stores the data coming from the wireless sensor network; and a webpage, that shows the information in sound map format. The wireless network has its link layer based on IEEE 802.15.4 and uses IPv6 protocol in the network layer, along with the adaptation layer for constrained devices 6LoWPAN and RPL protocol for the network topology and packet routing. The network is composed of sensor nodes spread across different points, which have its position determined by GPS. Through a microphone and a preamplifier circuit, these nodes measure the sound intensity of the environment in which they're in. Each of these sensor nodes sends the data obtained in UDP packets through the network, until they arrive at the network coordinator. The network coordinator is responsible of sending the data to a data server in the Internet via cellular network. On this server, all data is stored in a database, which is continuously consulted by a script of the webpage, that shows visually the current information in sound map format. The results obtained until the end of this work showed that the solution used on this system is functional, however, it uses most part of the RAM of the microcontrollers. The data traffics in the wireless sensor network effectively through the Internet protocols.

Keywords: Wireless sensor network. Sound pollution. Internet protocols. 6LoWPAN. RPL.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTAGEM DE CÓDIGOS

LISTA DE SIGLAS E ABREVIATURAS

6LoWPAN	IPv6 *over Low Power Wireless Personal Area Network*
ABS	*Anti-lock Braking System*
ADC	*Analog to Digital Converter*
AJAX	*Asynchronous JavaScript and XML*
API	*Application Program Interface*
APN	*Access Point Name*
BLE	Bluetooth *Low-Energy*
CoAP	*Constrained Application Protocol*
CPU	*Central Processing Unit*
DAG	*Directed Acyclic Graph*
DODAG	*Destination-Oriented Directed Acyclic Graph*
ETX	*Estimated number of Transmissions*
GPIO	*General Purpose Input/Output*
GPRS	*General Packet Radio Service*
GPS	*Global Positioning System*
HAL	*Hardware Abstraction Layer*
HTTP	*HyperText Transfer Protocol*
HTML	*HyperText Markup Language*
IEEE	*Institute of Electrical and Electronics Engineers*
IETF	*Internet Engineering Task Force*
IoT	*Internet of Things*
IP	*Internet Protocol*
IPv4	*Internet Protocol version 4*
IPv6	*Internet Protocol version 6*
JSON	*JavaScript Object Notation*
LLC	*Logical Link Control*
LLN	*Low-power and Lossy Network*
LP-WPAN	*Low Power Wireless Personal Area Network*
LR-WPAN	*Low Rate Wireless Personal Area Network*
M2M	*Machine to Machine*
MAC	*Medium Access Control*
MANET	*Mobile Ad hoc Network*

MCU	*Microcontroller Unit*
MEMS	*Micro Electro-Mechanical Systems*
MQTT	*Message Queuing Telemetry Transport*
MPU	*Microprocessor Unit*
QoS	*Quality of Service*
RAM	*Random-Access Memory*
ReST	*Representational State Transfer*
RISC	*Reduced Instruction Set Computer*
RF	*Radio Frequency*
RFID	*Radio-Frequency Identification*
ROLL	*Routing Over Low-power and Lossy networks*
RSSF	Rede de Sensores Sem Fio
RTOS	*Real-Time Operating System*
SLIP	*Serial Line Internet Protocol*
SQL	*Structured Query Language*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
UWB	*UltraWideBand*
TIC	Tecnologias da Informação e Comunicação
WSN	*Wireless Sensor Network*
WPAN	*Wireless Personal Area Network*

SUMÁRIO

1 INTRODUÇÃO

Desde os primórdios da humanidade, sons e ruídos fazem parte da vida dos seres humanos. Com o avanço tecnológico, principalmente após a Revolução Industrial, os sons emitidos, particularmente em áreas urbanas, vêm se tornando mais incômodos. Embora seja um problema cada vez mais comum e generalizado na sociedade moderna, os incômodos gerados pela poluição sonora são debatidos desde a antiguidade. Na antiga Roma, por volta de 45 a.C, já existiam regras para evitar o barulho das carruagens trafegando pelas ruas de pedra da metrópole em certos períodos do dia (BRAMBILLA, 2004).

Hoje, convive-se com diversas fontes de sons e ruídos, seja o tráfego de aeronaves, metrôs, automóveis, frotas de ônibus ou caminhões, seja a construção civil, o barulho das fábricas e indústrias, bares e casas noturnas, equipamentos de som automotivo, acidentes de trânsito, fogos de artifício e entre outras. Tais fontes de sons e ruídos causam o que vem sendo definido como poluição sonora, que é uma sobreposição de sons indesejáveis que provocam a perturbação individual e coletiva, devido à excessiva intensidade dos mesmos.

De acordo com a World Health Organization (2011), existem evidências suficientes dos impactos negativos causados pela poluição sonora à saúde humana. Estudos observacionais e experimentais têm demonstrado que a exposição ao ruído leva à irritação, perturba o sono e causa sonolência diurna, afeta a melhora de pacientes e o desempenho de profissionais em hospitais, aumenta a ocorrência de hipertensão e doenças cardiovasculares, e prejudica o desempenho cognitivo das crianças e adolescentes nas escolas (BASENER et al., 2014).

Quando se trata da qualidade de vida das pessoas e meio ambiente, é muito importante que sejam investidos recursos, tempo e pesquisa em soluções para melhoria dos fatores envolvidos nesses indicadores. O avanço tecnológico pode ter agravado a poluição sonora com o passar do tempo, mas também trouxe as tecnologias necessárias para possibilitar o desenvolvimento de forma sustentável. Com o estudo e devido emprego dessas tecnologias é possível eliminar ou reduzir, além da poluição sonora, muitos outros problemas decorrentes, entre outros, do crescimento e adensamento das cidades e do aumento populacional, de equipamentos e máquinas.

O surgimento de novas tecnologias de Internet que promovem serviços baseados em nuvem, como as máquinas virtuais em nuvem, a Internet das Coisas ou *Internet of Things* (IoT), Redes de Sensores Sem Fio (RSSFs) ou *Wireless Sensor Networks* (WSNs) e *Radio-Frequency Identification* (RFID), o uso de *smartphones* e *smartmeters*, interfaces de usuário com o mundo físico, comunicações com base na *web* semântica, abre novos caminhos para tomadas de ação

coletiva e resolução de problemas de forma colaborativa. Estas tecnologias são importantes para o desenvolvimento de cidade mais inteligentes, visando proporcionar melhores condições de vida e sustentabilidade.

Existem diversas definições para o que é de fato uma cidade inteligente ou *smart city*, mas de maneira genérica, uma cidade inteligente é aquela que explora as tecnologias digitais ou Tecnologias de Informação e Comunicação (TIC) para melhorar os serviços públicos (CENEDESE et al., 2014). Segundo a Sator (2015), a empresa brasileira responsável pelo evento *Connected Smart Cities* no Brasil, cidade inteligente é aquela que cresce de forma planejada, tendo como base a análise do desenvolvimento de indicadores, como economia, mobilidade, governo, meio ambiente, urbanismo e qualidade de vida. Para Cenedese et al. (2014), *smart city* é um paradigma que aplica as tecnologias de comunicação mais avançadas em ambiente urbano com o objetivo de melhorar a qualidade de vida e fornecer um amplo conjunto de serviços de valor para os cidadãos e a administração das cidades.

Este trabalho se insere no contexto de *smart city* com o desenvolvimento de um sistema em uma abordagem IoT, utilizando as tecnologias para o monitoramento em tempo real dos níveis acústicos urbanos, em serviço do bem-estar público. Como resultado, este trabalho traz inovação e a busca por melhor qualidade de vida nos centros urbanos. A solução envolve o uso das tecnologias envolvidas em IoT, como RSSFs e sistemas embarcados, utilizando *General Packet Radio Service* (GPRS) e IPv6 *over Low Power Wireless Personal Area Networks* (6LoWPAN) para as comunicações sem fio e *Global Positioning System* (GPS) para posicionamento, bem como sistemas em nuvem, tendo como recursos *HyperText Transfer Protocol* (HTTP), *HyperText Markup Language* (HTML), JavaScript, *Hypertext Preprocessor* (PHP), *Structured Query Language* (SQL) e Google Maps *Application Program Interface* (API).

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Desenvolver um sistema para monitoramento e mapeamento em tempo real dos níveis acústicos em ambientes urbanos, com acesso às informações obtidas por dispositivos eletrônicos inteligentes a partir da Internet em forma de mapas sonoros urbanos.

1.1.2 Objetivos Específicos

- Analisar a aquisição do sinal do sensor;
- Programar a comunicação com os módulos de RF, de GPRS e GPS;
- Modelar e programar o banco de dados do Servidor *Web*;
- Desenvolver a interface *web* para usuário que apresenta as informações;
- Integrar o serviço em nuvem e os sistemas embarcados.

1.2 JUSTIFICATIVA

O ruído constitui, atualmente, um dos principais problemas ambientais dos grandes centros urbanos. Nas últimas décadas, o ruído urbano deixou de ser um problema de vizinhança, para se transformar em um problema generalizado. Segundo Zajarkiewicch (2010), esse problema tem sido considerado como o maior fator de distúrbio ambiental, percepção que se evidencia pela grande quantidade de estudos que têm sido realizados para avaliar os efeitos causados devido à exposição ao ruído. Os efeitos podem ser classificados em específicos (auditivos) e não específicos (extra-auditivos), que por sua vez podem ser subdivididos em subjetivos (incômodo) e objetivos (interferência na comunicação, distúrbios do sono, etc.).

Os ruídos excessivos provocam perturbação da saúde mental, degradando a qualidade do sono durante períodos de repouso e as capacidades cognitivas durante períodos de trabalho, estudo, ou de outras atividades, reduzindo, por exemplo, a atenção de motoristas enquanto dirigem. Segundo Stansfeld et al. (2003), alguns estudos indicam conexão direta entre a exposição a ruídos e o declínio de saúde e qualidade de vida, citando o aumento de pressão arterial, perda de audição, degradação de funções cognitivas, incômodo e até mesmo sintomas psicológicos como possíveis efeitos.

Segundo Machado (2003), a poluição sonora ofende o meio ambiente, e consequentemente afeta o interesse difuso e coletivo. Os níveis excessivos de sons e ruídos causam deterioração na qualidade de vida, na relação entre as pessoas, especialmente quando acima dos limites suportáveis pelo ouvido humano ou prejudiciais ao repouso noturno e ao sossego público. Em função dos frequentes estudos acerca das consequências maléficas da poluição sonora sobre o organismo humano e da enorme quantidade de fontes causadoras de

poluição sonora, esta vem sendo interpretada como crime de acordo com os artigos 54 e 59 da Lei 9.605/98, que trata dos Crimes Ambientais (MACHADO, 2003; CAVALCANTE, 2012).

Além dos fatores antropológicos, sociais e ambientais relacionados à poluição sonora, a realização deste trabalho se justifica pelo uso de tecnologias de comunicação em sistemas com recursos restritos, como memória, capacidade de processamento, energia, etc. Uma destas tecnologias é a arquitetura *Internet Protocol* (IP) e seus protocolos para redes de objetos inteligentes, ou *smart objects*. De acordo com Dunkels e Vasseur (2010, p. 3), objeto inteligente pode ser tecnicamente definido como sendo um item equipado com uma forma de sensor ou atuador, um pequeno microprocessador, um dispositivo de comunicação e uma fonte de energia. Dunkels e Vasseur (2010, p. 38) argumentam que a arquitetura IP é a que permite a interoperabilidade entre as diversas tecnologias de comunicação existentes, sendo escalável a ponto de alcançar os desafios impostos pelas redes de objetos inteligentes de larga escala, e leve o suficiente para as restrições de recursos existentes a nível de nó.

Levando em conta a relevância desses fatos, este trabalho vem a colaborar com uma aplicação das tecnologias existentes em IoT, visando o bem-estar público em meio urbano e o desenvolvimento de cidades mais inteligentes, que proporcionem melhor qualidade de vida e com mais respeito ao meio ambiente. O sistema obtido como resultado da realização deste trabalho, quando implantado, acrescentará parâmetros de informação acústica contextualizados no espaço geográfico urbano em uma base de dados, permitindo um estudo mais profundo no planejamento civil e público. O sistema também descreve uma plataforma aberta, de modo que qualquer cidadão tenha acesso aos dados acústicos do ambiente urbano ao seu redor, podendo abranger futuramente outros dados sensoriais. Além disso, um sistema assim poderia vir a auxiliar as autoridades responsáveis pelo bem-estar público em questões de monitoramento, confirmação e redução de possíveis fontes de poluição sonora em curto prazo ou em tempo real.

2 REFERENCIAL TEÓRICO

Este capítulo apresenta alguns trabalhos relacionados, aspectos teóricos de sistemas embarcados e redes de sensores sem fio, bem como uma introdução à Internet das Coisas e à teoria do som.

2.1 TRABALHOS RELACIONADOS

2.1.1 Padova Smart City

Um dos trabalhos correlacionados à Internet das Coisas e ao paradigma de *smart cities*, e muito parecido com a proposta envolvida neste trabalho, foi experimentado na cidade de Pádua (Padova, em italiano), no norte da Itália. Padova Smart City é o nome do sistema que foi arquitetado para obter dados sensoriais do ambiente urbano de Pádua. O sistema consiste em alguns nós sensores colocados em postes de luz, e conectados à rede municipal e à Internet por meio de *gateways*. Cada um dos nós sensores foi geograficamente localizado, assim os dados eram enriquecidos com a informação contextual de localidade (CENEDESE et al., 2014).

Os nós sensores foram equipados com sensores de luminosidade, que diretamente medem a intensidade da luz emitida pelas lâmpadas dos postes e quaisquer outras fontes de luz que alcancem o sensor. As medições são realizadas em períodos regulares de tempo ou por meio de uma requisição. Também foram utilizados sensores de temperatura e umidade, para monitorar as condições climáticas. Um dos nós sensores foi equipado com um sensor de benzeno (C_6H_6), para monitorar a qualidade do ar. Os dispositivos são alimentados por pequenas baterias, para facilitar a implantação em qualquer localidade. Apenas o nodo com sensor de benzeno foi alimentado com uma fonte externa de energia, pelo fato de o componente sensorial necessitar de muita energia, podendo drenar rapidamente a energia das baterias (CENEDESE et al., 2014).

O projeto adotou padrões da *Internet Engineering Task Force* (IETF), que são padrões abertos e livres de *royalties*, satisfazendo o requisito de uso de *software Open Source* em sistemas do governo italiano e seus órgãos públicos. A abordagem adotada pelo projeto tornou possível a criação de dispositivos IoT flexíveis, que podem facilmente interagir com serviços *web* através da adoção do modelo *Representational State Transfer* (ReST). O modelo garante a compatibilidade na estrutura dos serviços de IoT e de serviços tradicionais de *web*,

promovendo a adoção da Internet das Coisas por ambos usuários finais, e desenvolvedores. Para garantir a compatibilidade com os serviços *web*, a abordagem exigiu o uso de camadas de protocolos adequadas nos diferentes elementos da rede. Em particular, foram utilizadas tecnologias como *Constrained Application Protocol* (CoAP), IPv6 e 6LoWPAN (CENEDESE et al., 2014).

2.1.2 SmartSantander

O projeto SmartSantander é uma colaboração entre empresas, como a Telefonica I+D, e instituições (incluindo Universidades) que trouxe a implantação de laboratórios de IoT de larga escala nas cidades de Santander (Espanha), Belgrado (Sérvia), Guildford (Reino Unido) e Lübeck (Alemanha). O projeto comporta dois tipos de experimentos: experimentos com redes de sensores sem fio, concebendo a IoT, e experimentos com prestações de serviços que só seriam possíveis através da IoT (aplicações utilizando dados do mundo real gerados por sensores, em tempo real, para o benefício público) (SMARTSANTANDER, [2013]; BIELSA, 2013). A Figura 1 mostra um mapa com diversos sensores em funcionamento.

Os serviços providos pelo projeto contam com gerenciamento integral de tráfego (informações sobre obras nas estradas, intensidade e fluxo de tráfego, etc.), monitoramento ambiental (poluição do ar, pressão sonora, temperatura, luminosidade e umidade), irrigação inteligente de parques e jardins, coleta de lixo otimizada, sistema de câmeras e vigilância, iluminação pública inteligente, mapas de estacionamento (vagas disponíveis e ocupadas), sensoriamento participativo (através de *smartphones*), realidade aumentada e etc.

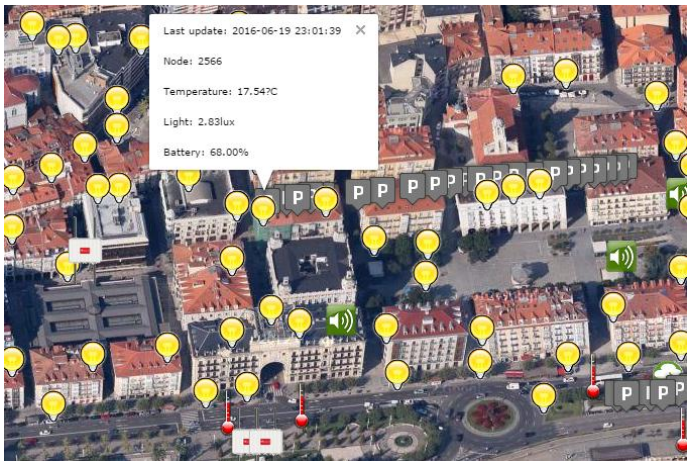

Figura 1 – Mapa de sensores em Santander, no norte costeiro da Espanha
Fonte: SmartSantander Map*.

* http://smartsantander.eu/map/

O projeto tem como objetivo a instalação de mais de 20 mil dispositivos nas cidades europeias, utilizando soluções desenvolvidas e manufaturadas pela empresa espanhola Libelium. A empresa desenvolve e suporta mais de 50 aplicações de IoT e *Machine to Machine* (M2M), incluindo o contexto de *Smart Cities*, com foco em plataformas de sensores sem fio (LIBELIUM, 2015).

2.2 SISTEMAS EMBARCADOS

2.2.1 Aspectos Gerais

Um sistema embarcado, ou embutido, é um sistema eletrônico que inclui um ou mais microcontroladores (*MicroController Units* – MCUs) ou microprocessadores (*MicroProcessor Units* – MPUs), os quais são programados para realizarem um conjunto de funções específicas, dedicadas e predefinidas. A palavra "sistema" refere-se ao fato da existência de vários componentes que atuam com o propósito de atingir um objetivo em comum. Já as palavras "embarcado" ou "embutido", possuem neste contexto, o significado da existência de um "computador escondido no interior" (VALVANO, 2014).

O que define um sistema embarcado são os dispositivos externos de entrada e saída, os quais permitem a interação do sistema com o mundo real. Pode-se dizer que os sistemas embarcados possuem pelo menos quatro características (VALVANO, 2014):

a) Sistemas embarcados tipicamente realizam uma única função. Cada sistema embarcado é único, e consegue realizar apenas as tarefas para as quais foi especificamente construído. Consequentemente, resolvem um limitado número de problemas;

b) Sistemas embarcados são constringidos quanto às especificações técnicas e de desempenho. Se um sistema embarcado não conseguir atingir suas especificações, não atingirá as expectativas do mercado;

c) Muitos sistemas embarcados necessitam controlar sistemas de tempo real. Um sistema de tempo real precisa garantir o pior caso no tempo de resposta entre o momento que surge uma nova entrada no sistema e o momento em que a entrada termina de ser processada. Outro requisito de tempo real que existe em muitos sistemas embarcados é a execução de tarefas periódicas;

d) Memória extremamente limitada, quando comparado com os computadores genéricos atuais. Existem exceções, como por exemplo sistemas que processam vídeo e áudio, mas na maioria dos sistemas embarcados pode-se contar com alguns milhares de *bytes* de memória, enquanto que em computadores genéricos são bilhões de *bytes* de memória.

Alguns exemplos de sistemas embarcados são:

- Automotivos: sistema de injeção eletrônica, sistema de controle de tração, sistema frenagem antitravamento (*Anti-lock Braking System* – ABS), sistema de alarme e travas, sistema de sensoriamento, etc.;
- Domésticos: Videogames, fornos micro-ondas, aparelhos de TV, lavadoras de louça, lavadoras de roupa, sistemas de segurança, geladeiras, etc.;
- Comunicação: Telefones celulares e *smartphones*, roteadores, *switches, hubs*, interfones prediais, equipamentos de GPS, etc.;
- Robótica: robôs industriais e móveis (aéreos, terrestres e subaquáticos);
- Aeroespacial e militar: Sistemas de gerenciamento de voo, controle de armas de fogo e rota de mísseis, sistemas de radar, etc.;
- Controle de processos: Processamento de alimentos, controle de plantas químicas, controle de manufaturas em geral.

Além das quatro características apresentadas, por serem sistemas dedicados à realização de tarefas específicas e predefinidas, os sistemas embarcados são projetados de forma a se otimizar o custo de produção, reduzindo o tamanho, peso, recursos computacionais e o consumo de energia, consequentemente restringindo os recursos do sistema a basicamente o mínimo necessário para o exercício de sua função.

À medida em que os sistemas embarcados crescem em complexidade, como o aumento da quantidade e complexidade de periféricos, bem como o aumento de poder computacional e de memória, a programação usual em laço infinito (sistemas *foreground*/*background*) para sistemas embarcados se torna inviável (LI; YAO, 2003). A programação de sistemas *foreground*/*background* possui uma série de deficiências as quais os sistemas operacionais de tempo real vêm a suprir.

2.2.2 Sistemas Operacionais de Tempo Real

Sistemas operacionais de tempo real, ou *Real-Time Operating Systems* (RTOSs), são sistemas operacionais multitarefas que suportam requisitos específicos de lógica e de tempo, e normalmente são aplicados em sistemas em que há pouca ou nenhuma flexibilidade quanto ao atraso de execução de tarefas e à inconsistência de dados, que se houverem, acarretará no mal funcionamento do sistema (LI; YAO, 2003).

Existem duas classificações de sistemas de tempo real, de acordo com os requisitos de tempo aos quais necessitam atender. Existem os chamados *soft*, cujas tarefas são realizadas pelo sistema tão rápido quanto possível, porém, não existem restrições rígidas de tempo para o término de uma tarefa. E existem os sistemas de tempo real *hard*, nos quais as tarefas precisam ser executadas de forma a atender explicitamente aos requisitos de tempo, ou seja, seus prazos precisam ser deterministicamente atendidos. Os sistemas de tempo real normalmente são combinação de *hard* e *soft* (LI; YAO, 2003).

Embora sistemas *foreground*/*background* sejam satisfatórios para aplicações mais simples, existem algumas desvantagens. Se uma tarefa muito complexa existir, a mesma poderá ocupar os recursos do processador por um tempo muito longo até ser concluída, e consequentemente atrasar as respostas de outras tarefas. Ainda, na programação em laço infinito, todas as tarefas são sequenciais e consequentemente possuem a mesma prioridade, o que em sistemas críticos é inapropriado. Os sistemas operacionais de tempo real foram criados com o objetivo de suprir estas e outras deficiências deste modelo.

Pode-se dizer que RTOS é um sistema operacional enxuto e robusto, de forma a se ocupar apenas alguns milhares de *bytes* de código, o que permite ser facilmente armazenado na memória *flash* de microcontroladores e microprocessadores. Em sistemas operacionais genéricos, como sistemas Unix por exemplo, existem processos e *threads*, que ficam concorrentemente disputando pela unidade de processamento (*Central Processing Unit* – CPU). Em um RTOS, existe o conceito de tarefas, semelhantes aos processos e *threads*, que ficam concorrentemente disputando pelo uso da MCU/MPU. Além do tamanho, os RTOSs também se diferenciam dos sistemas operacionais genéricos por serem construídos para atender aos requisitos específicos de sistemas de tempo real, podendo atender deterministicamente aos *deadlines* das tarefas.

O núcleo de um RTOS é responsável pelo gerenciamento das tarefas e pelas trocas de contexto, e existe nele um escalonador de tarefas, que funciona de acordo com um algoritmo de escalonamento. Em projetos típicos, uma tarefa pode ter os seguintes estados: em execução,

pronta para execução, e bloqueada (esperando por um evento). As tarefas geralmente são classificadas e escalonadas de acordo com prioridades. Como em sistemas operacionais genéricos, no RTOS o escalonamento das tarefas pode ser cooperativo ou preemptivo.

O Quadro 1 apresenta alguns RTOSs multiplataforma.

RTOS	Desenvolvedor(es)	Licença de *Software*	Modelo de Código Fonte
BRTOS (Brasil)	Gustavo Weber Denardin e Carlos Henrique Barriquello	MIT *License*	*Open Source*
TI-RTOS	Texas Instruments	BSD *License*	*Open Source*
RT-Linux	Wind River Systems	GNU GPLv2	*Open Source*
FreeRTOS	Real Time Engineers Ltd.	GNU GPL	*Open Source*
µC/OS-II e III	Micrium	Proprietário	Disponível sob licença
embOS	SEGGER Microcontroller Systems	Proprietário	*Closed Source*
ThreadX	Express Logic, Inc.	Proprietário, *Royalty-free*	Disponível para clientes
Windows CE	Microsoft	Proprietário	*Shared Source*
QNX Neutrino	BlackBerry Ltd.	Proprietário	*Shared Source*
VxWorks	Wind River Systems	Proprietário	*Closed Source*

Quadro 1 – Pequena lista de RTOSs multiplataforma
Fonte: Adaptado de Comparison (2015).

Em suma, um RTOS dá ao desenvolvedor de sistemas embarcados, além da maior portabilidade de código, sérios benefícios como o escalonamento de tarefas, execução de multitarefas, comportamento determinístico de eventos e interrupções, códigos de interrupções mais curtos, comunicação entre tarefas do sistema, melhor gerenciamento do uso de memória, e também, permite que o desenvolvedor foque mais no desenvolvimento da aplicação ao invés do gerenciamento de recursos. Embora existam diversos RTOSes e a maioria possuir os mesmos recursos, o BRTOS foi escolhido neste trabalho de forma a promover um sistema que é inteiramente desenvolvido por brasileiros.

2.2.3 BRTOS

O BRTOS é um sistema operacional de tempo real brasileiro, que foi desenvolvido com o intuito de ser um RTOS simples com mínimo de consumo de memória de dados e programa para microcontroladores de pequeno porte. O sistema possui um escalonador preemptivo por prioridades, e suporte para um total de 32 tarefas instaladas associadas a prioridades.

O BRTOS oferece como recursos de gerenciamento: semáforos, *mutex*, caixas de mensagens, filas e temporizadores por *software* (*timers*). Com relação ao *mutex*, este utiliza o protocolo *priority ceiling*, em que cada recurso é associado a um teto de prioridade, que é igual a maior prioridade entre as tarefas que podem bloquear tal recurso. Desta forma, o protocolo eleva a prioridade de tarefas em determinadas situações, com o intuito de evitar *deadlocks* e inversões de prioridade.

O sistema é escrito praticamente todo em linguagem C de programação, possuindo algumas chamadas de código em Assembly na camada de abstração de *hardware* (*Hardware Abstraction Layer* – HAL). Já existem vários *ports* oficiais lançados para diversas plataformas e famílias de microcontroladores de diferentes arquiteturas.

O BRTOS pode ocupar menos de 100 *bytes* de memória RAM e 2 KB de memória de programa, quando utilizando os recursos mínimos do sistema, e pode chegar a 1 KB de RAM e 8 KB de memória de programa caso sejam utilizados todos os serviços do sistema e o número máximo de tarefas instaladas. A quantidade de RAM necessária para as tarefas depende da arquitetura escolhida, do número de interrupções habilitadas e das necessidades da tarefa (variáveis e chamadas de funções). Microcontroladores de arquitetura com conjunto reduzido de instruções (*Reduced Instruction Set Computer* – RISC) necessitam maior espaço em RAM, pois possuem maior número de registradores.

2.3 REDES DE SENSORES SEM FIO

Avanços nos campos de estudo sobre eletrônica digital (microcontroladores e microprocessadores), comunicação sem fio e microssistemas eletromecânicos (*Micro Electro-Mechanical Systems* – MEMS) permitiram o desenvolvimento de pequenos e baratos nós sensoriais, multifuncionais e de baixo consumo de energia, e que podem se comunicar entre si dentro de curtas distâncias, formando as redes de sensores sem fio (AKYILDIZ et al., 2002).

2.3.1 Introdução e *Overview*

Redes de sensores são estruturas compostas de elementos de medição, computação e comunicação, que dão aos administradores a habilidade de instrumentar, observar, e tomar ações quanto a eventos e fenômenos em um ambiente específico (SORAHBY; MILONI; ZNATI, 2007). Elas podem ser vistas como um tipo especial de rede móvel *ad hoc* (*Mobile Ad hoc Network* – MANET), na qual a topologia da rede está em constante mudança, e como uma das vertentes da computação ubíqua (LOUREIRO et al., 2003). As aplicações das redes de sensores são inúmeras, e por muitas vezes também há interesse em realizar ações de controle.

As tecnologias de sensoriamento incluem sensores de campo elétrico e eletromagnético; sensores de frequência de ondas de rádio; sensores óticos, eletro óticos, e infravermelho; radares; lasers; sensores de localização/navegação; sensores sísmicos e de ondas de pressão; sensores de parâmetros ambientais (vento, umidade, temperatura); sensores bioquímicos; etc. (SORAHBY; MILONI; ZNATI, 2007). Os nós sensores ou nós sensoriais, da atualidade, podem ser descritos como dispositivos "inteligentes" e baratos, equipados com múltiplos elementos sensitivos (transdutores) acoplados. Estes dispositivos sensoriais sem fio são também chamados de *motes* (SORAHBY; MILONI; ZNATI, 2007).

De acordo com Sorahby, Miloni e Znati (2007, p. 1), existem quatro componentes em uma rede de sensores: um conjunto de sensores distribuídos em uma determinada área; uma rede de interconexão (tipicamente sendo sem fio); um nó central para aglomerar os dados; e um conjunto de recursos de computação no nodo central ou além da rede, para realizar a correlação dos dados e eventos, as requisições de estado, e mineração de dados.

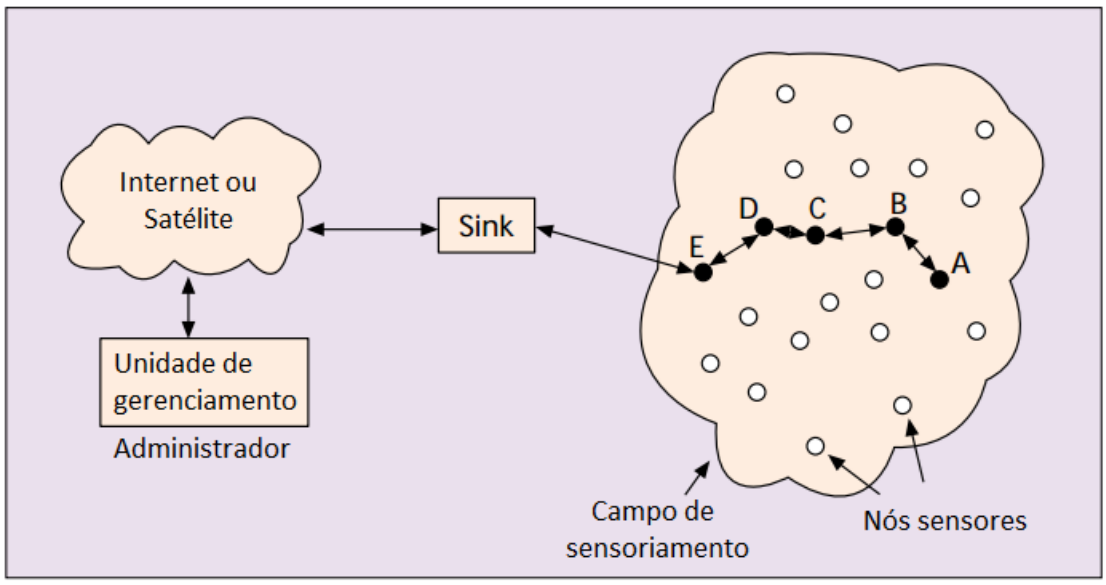

Figura 2 – Representação de uma rede de sensores
Fonte: Adaptado de Akyildiz et al. (2002, p. 103).

Nas RSSFs, os nós sensores são interconectados via séries de *links*, ou enlaces, sem fio de curta distância, com baixo consumo de energia e capacidade *multi-hop*. Os enlaces sem fio podem ser formados por rádio, infravermelho ou por mídia ótica, mas o contexto deste trabalho se limita a enlaces de radiofrequência (*Radio-Frequency* – RF). Normalmente as RSSFs utilizam a Internet ou alguma outra rede para a entrega de informações a um ponto (ou múltiplos pontos), para agregação de dados e análise (SORAHBY; MILONI; ZNATI, 2007, p.3). Quando um nó sensorial serve como ponte de acesso para a Internet ou alguma outra rede, objetivando a entrega dos dados adquiridos, ele é chamado de sorvedouro, ou *sink*, no contexto de RSSF, ou então *gateway* no contexto geral de rede (RUIZ et al., 2014).

2.3.2 Componentes e Organização do Nó Sensor

A estrutura de um protótipo genérico de nó sensor é composta basicamente de seis componentes principais: fonte de energia, comunicação, unidade de processamento, memória de armazenamento, interface para depuração e sensores (VIEIRA, 2004). A Figura 3 ilustra em blocos a estrutura de um protótipo de nó sensor.

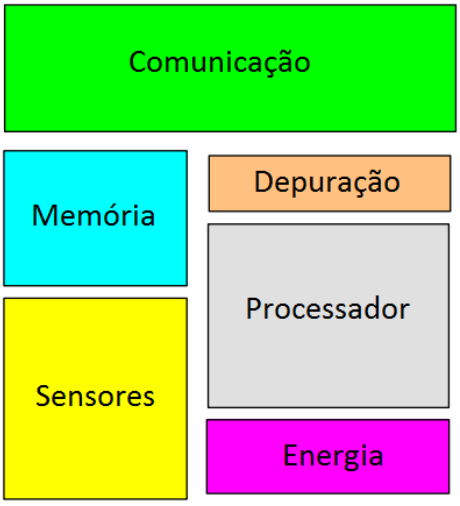

Figura 3 – Diagrama de blocos de protótipo de nó sensor
Fonte: Adaptado de Vieira (2004, p. 15).

O bloco de comunicação consiste de um canal de comunicação sem fio bidirecional pelo qual o nó sensor envia e recebe dados. O bloco de memória refere-se a um dispositivo de armazenamento externo, para armazenar dados, códigos de programa e registros de eventos. O bloco de depuração incorpora uma interface para realização de testes, não sendo necessário no produto real. O bloco de sensores corresponde aos transdutores para medição ou detecção de

fenômenos e eventos. O bloco de processamento refere-se ao microprocessador, ou microcontrolador, e é responsável pelo gerenciamento e integração do sistema como um todo, desde o gerenciamento de pilha de protocolos até o gerenciamento de energia, se houver, e de tarefas.

Uma visão mais detalhada da estrutura de um nó sensor genérico pode ser visto na Figura 4. Nesta figura pode-se ver a correlação entre os blocos vistos anteriormente na Figura 3, agora compostos com os componentes de *hardware* como a bateria, módulo RF, microcontrolador, conversor analógico-digital (ADC), sensores, memória, entre outros, e também com a inclusão de componentes de *software* como o sistema operacional, os protocolos de rede, *software* de gerenciamento de energia, filtragem de sinal, e outros algoritmos.

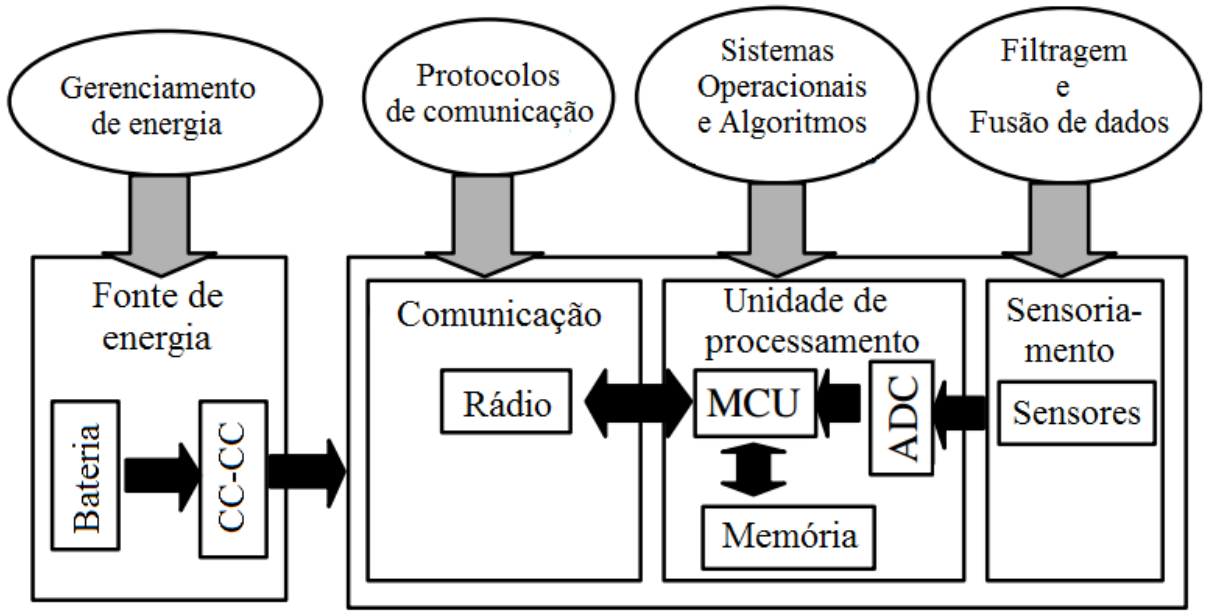

Figura 4 – Diagrama detalhado da arquitetura genérica do nó sensor
Fonte: Adaptado de Vieira (2004, p. 19).

A bateria fornece energia por meio de um conversor CC-CC aos circuitos do nó sensor, representados pelos blocos de comunicação, processamento e sensoriamento. O módulo de sensoriamento está diretamente ligado ao módulo de processamento por meio do ADC, o qual traduz o sinal elétrico dos sensores para o domínio digital. O módulo de comunicação está intimamente ligado com o módulo de processamento através do módulo de rádio. A unidade de processamento gerencia todas as tarefas do nó sensor através dos elementos de *software* como sistema operacional, e algoritmos como protocolos de rede, protocolos de aplicação, protocolos de roteamento, protocolos de transporte, filtragem e processamento de sinais, gerenciamento de energia, e entre outros.

2.3.3 Requisitos de Sistema e Características

Existem alguns requisitos chave para o projeto de nós sensores e redes de sensores sem fio, bem como algumas características predominantes. Dependendo da aplicação, outras características e requisitos, além dos apresentados neste tópico, podem complementar a RSSF e seus nós sensores. Neste contexto, pode-se definir alguns requisitos e características principais das redes de sensores sem fio:

- Durabilidade: Uma rede de sensores, como em qualquer projeto envolvendo sistemas embarcados, tem como um dos principais requisitos funcionar sem a necessidade de intervenção humana por um longo período de tempo;
- Eficiência energética: Nós sensores devem ser eficientes energeticamente. Tipicamente, os nós sensores possuem uma limitada quantidade de energia para funcionarem, quando não conectados à uma fonte contínua de energia, o que normalmente determina seu tempo de vida, que por sua vez pode também determinar o tempo de vida da rede de sensores em si. Por isso, energia é o recurso chave em um nó sensor, sendo uma das principais métricas de eficiência das RSSFs. As operações de comunicação, processamento, sensoriamento e atuação devem ser eficientes energeticamente (POTDAR; SHARIF; CHANG, 2009). Podem ser incorporados também sistemas de captação de energia aos nós sensores para que se prolongue seu tempo de vida. É altamente custoso fazer a troca e recarga de baterias de um sistema de grande escala como as RSSFs;
- Escalabilidade: O sistema deve suportar uma grande quantidade de nós sensores, que pode ir de centenas e milhares de nós sensores até extremos milhões de dispositivos, dependendo da aplicação (AKYILDIZ, 2002). O sistema também deve suportar o dinâmico aumento da rede, pela adição de novos nós sensores a qualquer momento;
- Tolerância à falha: O sistema deve ser robusto quanto à falha dos nós sensores (término de energia, destruição física, problemas de *hardware* ou *software*, etc.) (POTDAR; SHARIF; CHANG, 2009). Ou seja, o mal funcionamento de alguns nós sensores não deve afetar a tarefa principal da rede de sensores (AKYILDIZ, 2002). Para isso o sistema deve ter a habilidade de manter as funcionalidades da rede de sensores sem nenhuma interrupção devido às falhas, sejam elas estocásticas ou determinísticas, de seus nós sensores;

- Baixo custo: O nó sensor deve ter baixo custo de produção, já que uma rede de sensores pode abranger milhares de nós sensores (VIEIRA, 2004). O custo de cada nó sensor é muito importante para justificar o custo total da rede (AKYILDIZ, 2002). Os custos de instalação e manutenção também entram neste requisito, para que a implantação de um sistema desta escala seja realística (POTDAR; SHARIF; CHANG, 2009);
- Recursos restritos: Os nós sensores são nada além de pequenos sistemas embarcados. Por este fato, um nó sensor possui memória limitada, largura de banda de comunicação limitada, energia limitada, e capacidade de computação limitada (AHMED; SHI; SHANG, 2003). Por isto deve-se ter em mente a otimização de algoritmos e protocolos, de acordo com a aplicação da rede de sensores sem fio;
- Sensoriamento distribuído: Utilizando uma RSSF, muito mais dados podem ser coletados comparado à apenas o uso de um único sensor. Mesmo que o sensor tenha um longo alcance, pode haver obstruções ambientais (VIEIRA, 2004). Assim, o sensoriamento distribuído provê maior robustez, quantidade de dados, e consequentemente maior qualidade de informação (variação do fenômeno no espaço geográfico), do que um sistema com um único nó sensor;
- Meio de comunicação: Os nós sensores devem comunicar-se por meio sem fio. Em muitas aplicações, o ambiente a ser monitorado não tem infraestrutura instalada para comunicações. Instalar fiação pode ser muito difícil e custoso em largas escalas, assim, nós sensores devem utilizar canais de comunicação sem fio. Além disso, a taxa de dados em RSSFs é tipicamente baixa, justificando o uso de protocolos de comunicação sem fio de baixa taxa de dados;
- *Multi-hop*: Um nó sensor pode não alcançar a base central, ou *sink*. A rede deve suportar múltiplos saltos para que os dados consigam chegar ao *sink*, a partir de um nó que não o alcança diretamente (VIEIRA, 2004).

Com base nestas características, as RSSFs podem ser classificadas dentro de um grupo de redes de baixa potência e com perdas (*Low-Power and Lossy Networks* - LLNs), que é um termo comumente utilizado para se referir às redes compostas por nós que possuem limites de processamento, memória e energia, e que são interconectados por uma variedade de *links* com perdas, devido ao uso de rádios de baixa potência e taxa de transmissão.

2.3.4 Pilha de Protocolos

Como em qualquer outra rede, uma RSSF requer a existência de protocolos que padronizem o comportamento dos dispositivos da rede, de forma a formar a rede e mantê-la funcional. Especificamente, deve existir uma pilha de protocolos a qual define, por exemplo, a forma como os *motes* acessarão o meio de comunicação sem fio, como serão feitos o encaminhamento de pacotes de dados e o roteamento através de roteadores intermediários, se o serviço de comunicação entre os *motes* será orientado à conexão ou não, e etc. A pilha de protocolos utilizada pelo *sink* e pelos nós sensores da Figura 2 pode ser vista na Figura 5.

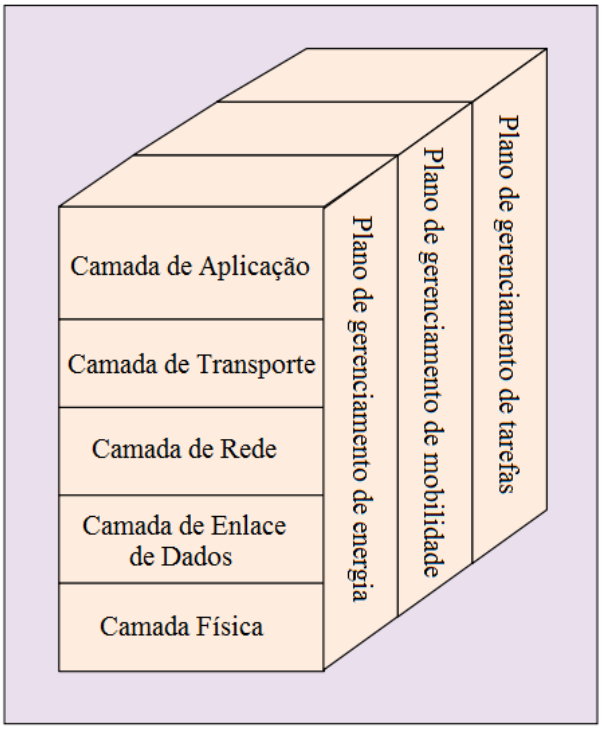

Figura 5 – Pilha de protocolos em RSSFs
Fonte: Adaptado de Akyildiz et al. (2002, p. 105).

A pilha de protocolos para redes de sensores consiste em cinco camadas de protocolos de comunicação e três planos de gerenciamento. As camadas consistem em:

- Camada de aplicação: Compreende as aplicações, incluindo aplicações de processamento, agregação de dados, atendimento às requisições externas, e aplicações de banco de dados externo. A camada de aplicação pode desde definir, por exemplo, o formato de amostra dos dados, até entender protocolos da camada de aplicação de Internet (protocolos baseados no modelo ReST, por exemplo);
- Camada de transporte: As principais funcionalidades da camada de transporte são a disseminação e acumulação de dados, *caching* e armazenamento (MAHALIK,

2007). A camada visa a transmissão confiável de dados fim-a-fim e redução de congestionamento de rede. *Transport Control Procotol* (TCP) e *User Datagram Protocol* (UDP) são protocolos típicos da camada de transporte. O TCP consegue efetuar transmissões confiáveis através da retransmissão de pacotes e checagem de *timeout*. Ele também reduz o congestionamento de rede através de controle de taxa de dados. Porém, em RSSFs, o TCP não é adequado para a camada de transporte devido ao seu largo *overhead* (HU; CAO, 2010);

- Camada de rede: A camada inclui protocolos de gerenciamento de topologias adaptativas e roteamento de pacotes. Ela é responsável pelo roteamento e encaminhamento de dados entre um grande número de nós sensores (*multi-hop*). Envolve algoritmos de busca de caminhos ótimos, almejando baixo consumo de energia, baixo *delay*, e outras melhorias. Uma vez que o caminho, ou rota, esteja estabelecida, os dados sensitivos podem ser retransmitidos de um nó sensor a outro, até chegarem ao destino (HU; CAO, 2010). Esta camada mantém e atualiza os percursos no caso de as condições de rede mudarem com o tempo (caso um *mote* drene suas baterias);
- Camada de enlace de dados: Enquanto a camada de transporte é responsável pelo controle de transmissão fim-a-fim, a camada de enlace de dados é responsável apenas pelas questões de comunicação entre vizinhos (1 *hop* de distância). Um nó sensor pode determinar se deve ou não ajustar a taxa de envio baseado nas configurações de *upstream* e *downstream* de *buffer* do nó sensor (HU; CAO, 2010). Algumas vezes, a camada de enlace de dados é chamada de camada *Medium Access Control* (MAC). Na verdade, MAC é uma subcamada da camada de enlace de dados. Outra subcamada é a *Logical Link Control* (LLC). Esta subdivisão da camada de enlace de dados em duas subcamadas serve para acomodar a lógica necessária para gerenciar o acesso ao meio de comunicação compartilhado (MAHALIK, 2007). A função do MAC é garantir que todos os nós sensores vizinhos não causem conflitos de transmissão de sinal, enquanto a camada de enlace de dados pode encarregar-se de detecção de erros, enquadramento de dados, e outras tarefas (HU; CAO, 2010);
- Camada física: Corresponde aos canais do meio físico de comunicação. Esta camada é responsável pela conversão de dados significativos em sinais de radiofrequência, através de *encoding*/modulação, e outros módulos de comunicação sem fio (HU; CAO, 2010). Esta camada apenas entende os "sinais" como níveis de

tensão (0 ou 1). A camada não compreende nenhuma questão relacionada às camadas superiores, como roteamento, conteúdo dos dados, confiabilidade, etc.

Os planos de gerenciamento de energia, gerenciamento de mobilidade, e gerenciamento de tarefas monitoram os recursos de energia do nó sensor, mobilidade/atuação e a distribuição de tarefas entre os nós sensores. Estes planos visam ajudar os nós sensores a coordenar tarefas de sensoriamento e diminuir o gasto de energia total da rede (AKYILDIZ, 2002).

2.3.5 Tecnologias de Comunicação de Radiofrequência

Dentro do campo de sensoriamento (área de implantação dos nós sensoriais), as RSSFs tipicamente empregam em suas camadas física e de enlace apresentadas na Figura 5, técnicas incorporadas à família IEEE 802 de padrões, desenvolvidas no final dos anos 60 e 70, que realizam o gerenciamento dos canais de transmissão. No entanto, outras técnicas também podem ser empregadas (SORAHBY; MILONI; ZNATI, 2007, p. 1).

Alguns dos padrões relevantes para comunicação via RF são:

- IEEE 802.11a/b/g/n/ah (Wi-Fi);
- IEEE 802.15.1 Bluetooth/BLE (Bluetooth Low-Energy);
- IEEE 802.15.3 Ultrawideband (UWB);
- IEEE 802.15.4 LR/LP-WPAN (*Low-Rate/Low-Power* WPAN);
- IEEE 802.16 WiMAX;
- IEEE 1451.5 (Wireless Sensor Working Group);
- Mobile IP.

Hoje em dia, a maioria das soluções comerciais em RSSF são baseadas no padrão IEEE 802.15.4, que define as camadas física e de enlace de dados para comunicações de baixa potência e baixa taxa de *bits* em *Wireless Personal Area Networks* (WPANs) (ATZORI; IERA; MORABITO, 2010).

2.4 A INTERNET DAS COISAS

2.4.1 Introdução

A Internet das Coisas é um conceito e um paradigma tecnológico que considera a presença difusa de uma variedade de coisas/objetos no ambiente, que através de conexões sem fio e com fio, e esquemas de endereçamento único, são capazes de interagir uns com os outros e cooperar com outras coisas/objetos para criar novas aplicações/serviços e atingir objetivos em comum (VERMESAN; FRIESS, 2014). Para Dunkels e Vasseur (2010, p. 112), IoT é a Internet onde redes de objetos inteligentes, como as RSSFs, pertencem realmente à Internet, como se fossem qualquer outra rede.

A IoT está rapidamente ganhando chão no cenário das telecomunicações modernas sem fio. No futuro, haverá aplicações que não serão muito diferentes das aplicações existentes nos dias de hoje, como e-mail e outros serviços de *web*, os quais deverão ser acessíveis pela comunidade da Internet (DUNKELS; VASSEUR, 2010). Qualquer usuário da Internet terá acesso às informações providas pelos objetos inteligentes, como por exemplo telemetria, através do acesso direto ao dispositivo ou por meio de servidores intermediários, como na Figura 6. Já existem muitas formas simples de ter acesso aos objetos inteligentes através da Internet, e o número destas aplicações continuará aumentando (DUNKELS; VASSEUR, 2010).

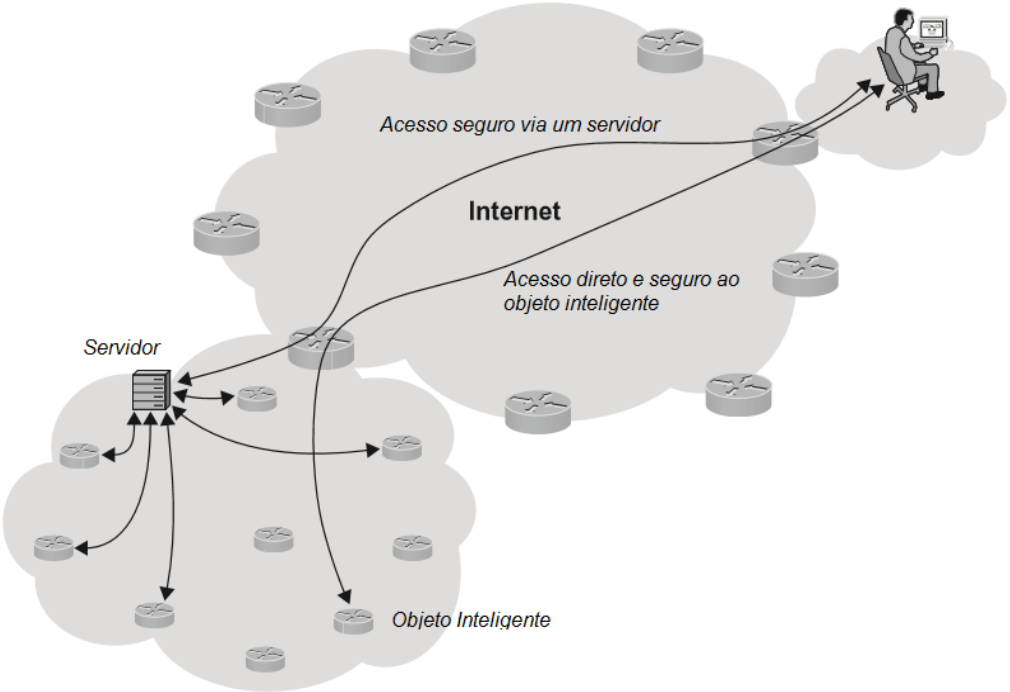

Figura 6 – A Internet das Coisas
Fonte: Adaptado de Dunkels e Vasseur (2010, p. 113).

Existem variadas aplicações possíveis com a evolução da IoT. Um dos contextos englobados por ela, e que é o foco deste trabalho, são as cidades inteligentes. Elas proverão informações úteis para seus cidadãos, melhorando sua qualidade de vida e ajudando-os a tomarem decisões importantes do dia-a-dia. Dados ambientais como a qualidade do ar, informação de transporte em tempo real, assistência emergencial, risco de ataques e desastres, e assim por diante (DUNKELS; VASSEUR, 2010). Outras aplicações proverão dados explorados pelos departamentos civis, para melhor gerenciar as cidades e seus recursos, como por exemplo, o gerenciamento da iluminação pública, detecção de vazamentos de água/gás, ou gerenciamento de tráfego (DUNKELS; VASSEUR, 2010).

2.4.2 Visões e Desafios

A IoT é um paradigma de visões diferentes, que trazem consigo desafios diferentes. Uma destas visões é direcionada aos simples objetos, ou coisas, buscando formas de identificá-los de forma única (RFIDs, µIDs, etc.), e como estes podem se conectar uns com os outros, como em RSSFs, por exemplo. Outra visão é a orientada à Internet, que propõe formas de conectar o mundo virtual com os objetos físicos no mundo real. E por último, mas não menos crítica, é a visão orientada à semântica, que se concentra nos desafios advindos do grande número de itens conectados, como por exemplo, formas de representar, armazenar, interconectar, procurar e organizar as informações geradas (ATZORI; IERA; MORABITO, 2010).

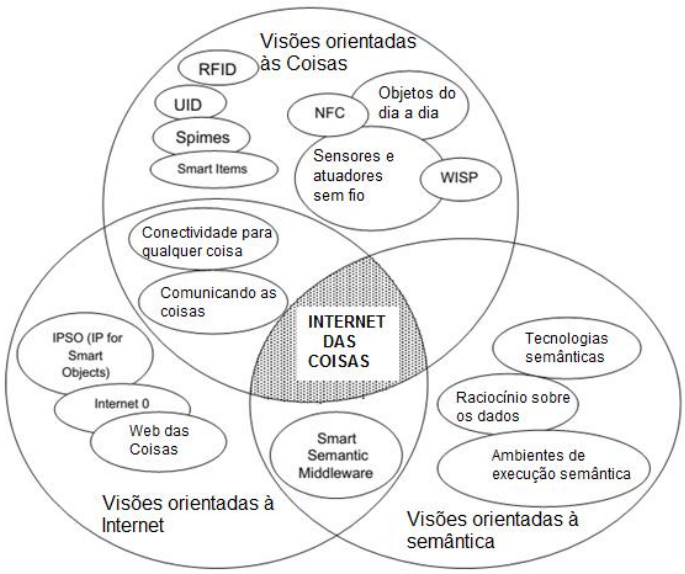

Figura 7 – Paradigma IoT como resultado da convergência de diferentes visões
Fonte: Adaptado de Atzori, Iera e Morabito (2010, p. 3).

De fato, a IoT é a convergência destas visões, como mostrado na Figura 7. A ideia convergente acabou se consolidando na seguinte frase, que faz apelo para a computação ubíqua: “conectividade a partir de qualquer tempo e qualquer lugar, para qualquer pessoa e para qualquer coisa” (ITU, 2005).

Embora as tecnologias existentes tornem o conceito de IoT algo realizável, ainda existe uma enorme demanda por pesquisa e desafios a serem vencidos. É por ser um paradigma convergente de várias visões, que concentram tecnologias diversas, que um dos principais desafios para que venha a se tornar real é a padronização. Outros desafios importantes envolvem privacidade de informação, protocolos de transporte eficientes, integridade de dados, caracterização de tráfego e suporte a *Quality of Service* (QoS), e eficiência energética e de utilização de recursos nos dispositivos.

2.4.3 Tecnologias Facilitadoras e Padronização

Várias contribuições para a padronização, modelagem e implantação do paradigma estão vindo da comunidade científica. Na Figura 8, pode se ver uma pequena amostra das tecnologias facilitadoras existentes e em desenvolvimento.

Como pode ser visto na Figura 7, dentro da visão orientada à Internet está a IP *for Smart Objects* (IPSO) *Alliance*, que é um fórum formado por 25 companhias para promover o *Internet Protocol* como a tecnologia de redes para conectar os diversos objetos inteligentes ao redor do mundo (ATZORI; IERA; MORABITO, 2010). A pilha do protocolo IP é leve e já conecta uma grande quantidade de dispositivos, funcionando até mesmo em dispositivos embarcados. Isto garante que o IP tem todas as qualidades para tornar a IoT realidade (IPSO, 2015).

Uma das relevantes tecnologias atuais em IP vem sendo desenvolvida e adaptada por um Grupo de Trabalho da IETF, chamado 6LoWPAN. O Grupo desenvolveu um conjunto de protocolos adaptados para integrar dispositivos com recursos restritos, e que utilizam o padrão IEEE 802.15.4 de comunicação, em redes IPv6 (ATZORI; IERA; MORABITO, 2010). Outra relevante tecnologia é o protocolo IPv6 *Routing Protocol for Low-Power and Lossy Networks* (RPL, lê-se “*ripple*”), desenvolvido por outro Grupo de Trabalho da IETF, chamado *Routing Over Low-Power and Lossy Networks* (ROLL), que será a base para o roteamento nas redes de objetos inteligentes (ATZORI; IERA; MORABITO, 2010). Para Atzori, Iera e Morabito (2010,

p. 12), 6LoWPAN e ROLL são os melhores candidatos para a emergente ideia de padronização, como parte da Internet do futuro.

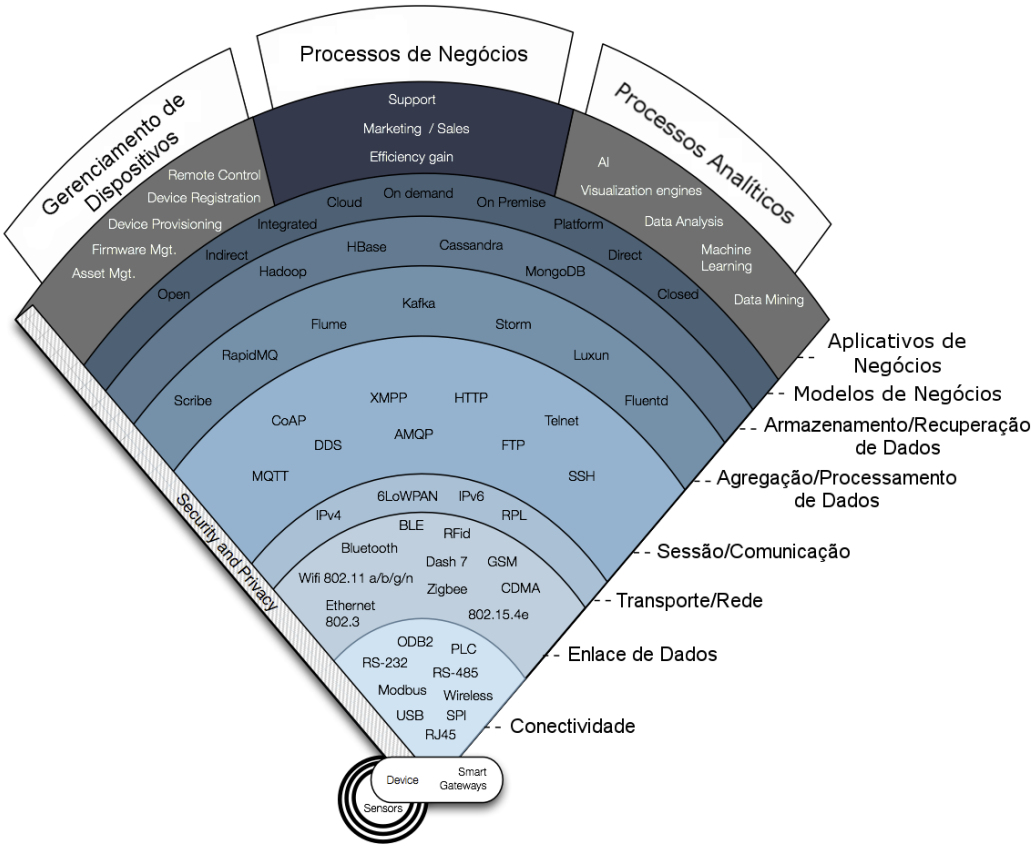

Figura 8 – Panorama de algumas tecnologias existentes que habilitam a IoT
Fonte: Adaptado de Passemard (2014).

Dunkels e Vasseur (2010, p. 198 e 312) afirmam que a padronização é absolutamente crítica e sinônimo de abertura e interoperabilidade. Também afirmam que o protocolo IP é, por excelência, uma tecnologia aberta e padronizada, e que sem dúvida alguma é o candidato ideal como infraestrutura unificada de comunicação, suportando uma infinidade de dispositivos interconectados.

2.5 CONTIKI-OS

O Contiki é um sistema operacional de código aberto para a Internet das Coisas, desenvolvido por um time composto por desenvolvedores do mundo inteiro, com contribuições da Atmel, Cisco, ETH, Redwire LLC, SAP, Thingsquare, e muitos outros, liderado por Adam Dunkels. O sistema provê comunicação através de protocolos de Internet, com suporte abrangente da pilha de protocolos TCP/IP, juntamente com emergentes protocolos e tecnologias

para LLNs, entre eles 6LoWPAN, RPL, CoAP, *Message Queuing Telemetry Transport* (MQTT), e outros (CONTIKI-OS, 2016).

2.5.1 Processos

O Contiki possui um *kernel* dirigido a eventos, e seus processos são executados em um contexto cooperativo, em que o escalonador não realiza a preempção dos processos. Cada um deles executa até terminar ou liberar o microcontrolador para o próximo processo. Um processo no Contiki consiste de duas partes: um bloco de controle de processo, o qual deve ser acessado apenas pelo *kernel* do sistema, e o código de execução, ou *protothread*, do processo. O bloco de controle contém informações sobre cada processo, como o estado do processo, ponteiro para o código do processo e o nome do processo (COLINA, 2016). Mais detalhes sobre o bloco de controle dos processos no Contiki podem ser vistos na Listagem 1.

```
struct process {
   struct process *next;
   const char *name;
   int (* thread)(struct pt *,
                  process_event_t,
                  process_data_t);
   struct pt pt;
   unsigned char state, needspoll;
 };
```

Listagem 1 – Estrutura dos blocos de controle dos processos no Contiki

O bloco de controle de um processo é uma estrutura interna usada pelo escalonador para invocar um processo, e também começar a sua execução. A declaração de um bloco de controle de um processo é feita utilizando a chamada PROCESS(), como pode se ver na Listagem 2. O primeiro parâmetro dessa chamada é o nome da variável de estrutura do processo e o segundo é a *string* do nome do processo. Já a *thread*, ou mais especificamente o código do processo, é declarada com a chamada PROCESS_THREAD(), como na Listagem 2, em que *ola_mundo* é o nome da variável de estrutura do processo, *ev* é o parâmetro que representa o número de um evento, e *data* é um ponteiro para um dado qualquer, podendo ser nulo.

```
PROCESS(ola_mundo, "Processo Olá Mundo"); // Declaração do processo
PROCESS_THREAD(ola_mundo, ev, data) // Declaração da protothread
 {
   PROCESS_BEGIN();
   printf("Olá, mundo!\n");
   PROCESS_END();
 }
```

Listagem 2 – Declaração de um processo e sua *protothread*

2.5.2 Eventos e *Protothreads*

No Contiki, uma aplicação qualquer consiste de um código de processo, que também é considerado como um *handler*, ou manipulador, de eventos. Os parâmetros do *handler* do processo permitem que o evento seja passado ao mesmo, desta forma podendo ser examinado e então tomada uma ação. Existem dois tipos de eventos, síncronos e assíncronos. Eventos síncronos são entregues imediatamente à um processo específico, e eventos assíncronos são colocados numa fila de eventos e repassados a todos ou à um processo específico (COLINA, 2016).

Os processos no Contiki são construídos com base em *protothreads*. *Protothreads* são funções em C, que através de macros funcionam como *threads*, porém extremamente leves e sem a necessidade de uma pilha própria, pois são dirigidas à eventos (COLINA, 2016). Essa técnica foi desenvolvida pensando em sistemas com pouquíssima memória e em simplificar o código. As *protothreads* são uma mistura de mecanismos de programação acionados por eventos, e implementam um controle de fluxo sequencial sem a necessidade de uma máquina de estados complexa.

Implemetação por Máquina de Estados

```
enum {ON, WAITING, OFF} state;

void eventhandler() {
  if(state == ON) {
    if(expired(timer)) {
      timer = t_sleep;
      if(!comm_complete()) {
        state = WAITING;
        wait_timer = t_wait_max;
      } else {
        radio_off();
        state = OFF;
      }
    }
  } else if(state == WAITING) {
    if(comm_complete() ||
       expired(wait_timer)) {
      state = OFF;
      radio_off();
    }
  } else if(state == OFF) {
    if(expired(timer)) {
      radio_on();
      state = ON;
      timer = t_awake;
    }
  }
}
```

Implementação baseada em Protothread

```
int protothread(struct pt *pt) {
  PT_BEGIN(pt);
  while(1) {
    radio_on();
    timer = t_awake;
    PT_WAIT_UNTIL(pt, expired(timer));
    timer = t_sleep;
    if(!comm_complete()) {
      wait_timer = t_wait_max;
      PT_WAIT_UNTIL(pt, comm_complete()
                || expired(wait_timer));
    }
    radio off();
    PT_WAIT_UNTIL(pt, expired(timer));
  }
  PT_END(pt);
}
```

Listagem 3 – Comparação de código baseado em máquina de estados e *protothread*

Como visto na Listagem 3, a organização de código é diferente para programas escritos com máquinas de estados e com *protothreads*. Programas baseados em máquinas de estados

tendem a consistir-se de uma única grande função contendo uma grande máquina de estados ou de pequenas funções onde a máquina de estados se torna difícil de entender. Ao contrário, programas baseados em *protothreads* tendem a ser construídos em torno de uma única função que contém a lógica de alto nível do programa. O sistema de eventos subjacente chama diferentes funções para cada evento de entrada. Esses programas tipicamente consistem de uma única função *protothread* e um pequeno número de manipuladores de eventos, que invocam a *protothread* quando um evento ocorre (COLINA, 2016).

Na Listagem 4, segue uma lista de macros de *protothread* disponíveis no Contiki e o que cada uma realiza. Essas funções devem ser chamadas apenas dentro de *threads* dos processos.

```
PROCESS_BEGIN(); // Declara o início da protothread do processo.
PROCESS_END(); // Declara o fim da protothread do processo.
PROCESS_EXIT(); // Sai do processo.
PROCESS_WAIT_EVENT(); // Espera por um evento.
PROCESS_WAIT_EVENT_UNTIL(); // Espera por um evento, mas com uma condição.
PROCESS_YIELD(); // Espera por qualquer evento, semelhante PROCESS_WAIT_EVENT().
PROCESS_WAIT_UNTIL(); // Espera por uma dada condição.
PROCESS_PAUSE(); // Temporariamente pausa o processo.
```

Listagem 4 – Macros de *protothreads* no Contiki

Estas *macros* foram adaptadas para o Contiki a partir da biblioteca de *protothreads* desenvolvida por Adam Dunkels. As principais operações de *protothread* se baseiam na seguinte implementação do pré-processador C, com algumas modificações (COLINA, 2016).

```
struct pt { lc_t lc };
#define PT_WAITING 0
#define PT_EXITED  1
#define PT_ENDED   2
#define PT_INIT(pt)           LC_INIT(pt->lc)
#define PT_BEGIN(pt)          LC_RESUME(pt->lc)
#define PT_END(pt)            LC_END(pt->lc);    \
                              return PT_ENDED
#define PT_WAIT_UNTIL(pt, c) LC_SET(pt->lc);    \
                              if(!(c))           \
                                  return PT_WAITING
#define PT_EXIT(pt)           return PT_EXITED
```

Listagem 5 – Principais operações de *protothreads*

As operações da Listagem 5 se estendem em continuações locais, vistas na Listagem 6, que utilizam *switch case* do C para execução da máquina de estados.

```
typedef unsigned short lc_t;
#define LC_INIT(c)   c = 0
#define LC_RESUME(c) switch(c) { case 0:
#define LC_SET(c)    c = __LINE__; case __LINE__:
#define LC_END(c)    }
```

Listagem 6 – Continuações locais das operações de *protothreads*

Os processos do Contiki podem também interagir uns com os outros através de postagens de eventos síncronos ou assíncronos, através de chamadas como *process_post()* ou *process_post_synch()*. Também pode ser feita a chamada *process_poll()*, que é um tipo especial de evento. Chamando esta função faz com que o processo passado como parâmetro da função venha a ser escalonado o mais rápido possível. Esta é a única função no módulo de processos do Contiki que é segura de se chamar a partir de uma interrupção (COLINA, 2016).

2.5.3 6LoWPAN e RPL

O Contiki fornece uma pilha de protocolos completa para a criação de redes de sensores sem fio baseadas em protocolos de rede IP, tanto IPv6 como IPv4, além da leve pilha de protocolos de comunicação *rime* (CONTIKI-OS, 2016).

Entre as camadas de protocolos de comunicação que o Contiki implementa em sua pilha de protocolos se encontra a 6LoWPAN, uma tecnologia que define padrões para comunicação via IPv6 através das tecnologias IEEE 802.15.4 de comunicação sem fio. 6LoWPAN é uma camada de adaptação entre o protocolo IPv6 e o protocolo IEEE 802.15.4, e é exclusivamente definida para redes IPv6. A pilha de protocolos para redes baseadas em 6LoWPAN do Contiki é estruturada como mostra a Figura 9.

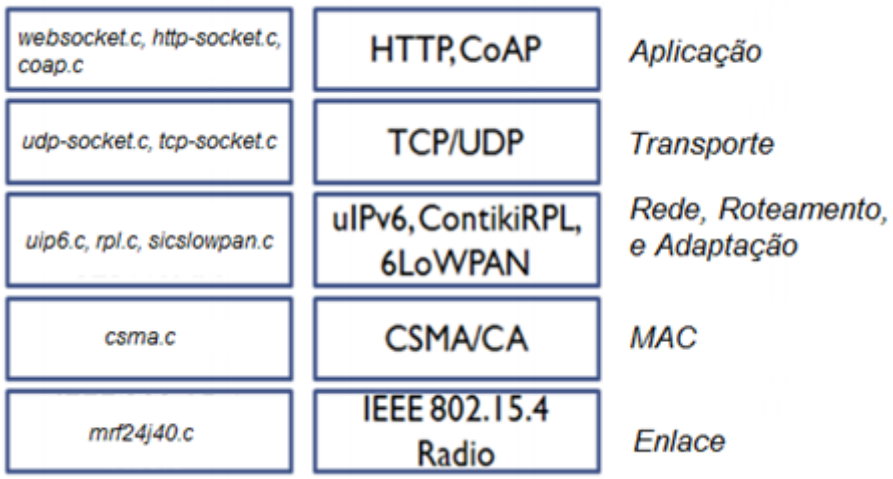

Figura 9 – Pilha de protocolos para redes 6LoWPAN no Contiki
Fonte: Adaptado de Protocols (2016).

O Grupo de Trabalho da IETF 6LoWPAN identificou uma série de desafios e metas para a utilização do IP nas redes IEEE 802.15.4, que foram descritos na RFC 4919. Seguem alguns objetivos e motivos para a camada 6LoWPAN (COLINA, 2016):

- Camada de fragmentação e desfragmentação de pacotes: as especificações do protocolo IPv6, definidas na RFC 2460, estabelecem que o maior tamanho de pacote (*Maximum Transmission Unit* – MTU) que uma camada de enlace deve suportar à camada IPv6 é 1280 *bytes*. No IEEE 802.15.4, as unidades de dados do protocolo podem ser tão pequenas quanto 81 *bytes*. Para resolver essa diferença entre essas camadas, uma camada de adaptação para fragmentação e desfragmentação de pacotes deve existir abaixo da camada IP;
- Compressão de cabeçalho: Dado que, no pior caso, o tamanho máximo disponível para transmitir pacotes IP através de um *frame* IEEE 802.15.4 é 81 octetos, e que o cabeçalho IPv6 tem tamanho de 40 octetos, sem cabeçalhos de extensão opcionais, sobram apenas 41 octetos para os protocolos das camadas superiores, como UDP e TCP. UDP utiliza 8 octetos no cabeçalho e TCP utiliza 20 octetos. Isto deixa 33 octetos para dados no UDP, e apenas 21 octetos para dados no TCP. Além do que foi apontado anteriormente, também é necessária uma camada de fragmentação e reconstituição, que irá utilizar mais alguns octetos do cabeçalho, deixando apenas alguns poucos octetos para os dados. Se os protocolos fossem utilizados como são, levaria à excessiva fragmentação e desfragmentação de pacotes. Tudo isso aponta para a necessidade da compressão do cabeçalho;
- Autoconfiguração de endereço: especificar métodos para a autoconfiguração *stateless* (ao contrário de *stateful*) de endereços IPv6, devido à grande quantidade de dispositivos normalmente presentes nas redes. Desta forma há a redução no tempo gasto para configuração nos *hosts*. Também, a necessidade de um método para criar Identificadores de Interface IPv6 (IPv6 IIDs) a partir do EUI-64 atribuído aos dispositivos compatíveis com IEEE 802.15.4;
- Protocolo de roteamento *mesh*: a necessidade de suporte à protocolos de roteamento simples para topologias *mesh* com múltiplos saltos. Os pacotes de roteamento devem caber em um único quadro IEEE 802.15.4;
- IEEE 802.15.4 define quatro tipos de quadros: quadros de *beacon*, quadros de comando MAC, quadros de reconhecimento (*acknowledgement*), e quadros de dados. Os pacotes IPv6 devem ser carregados em quadros de dados.

Como protocolo de roteamento, redes 6LoWPAN utilizam o RPL, que é documentado pela RFC 6550. Este protocolo é responsável por retransmitir pacotes em múltiplos saltos que

separam os nós de origem e de destino. O protocolo suporta tráfego de dados ponto-a-ponto (entre dispositivos dentro de uma rede), ponto-a-multiponto (de um ponto de controle central para um subconjunto de dispositivos dentro da rede), e multiponto-a-ponto (de dispositivos dentro da rede até um ponto de controle central) (COLINA, 2016).

O RPL é um protocolo do tipo vetor de distância (*distance vector*) adaptado para uma variedade de tipos de LLNs. O funcionamento do RPL se baseia em Grafos Acíclicos Dirigidos (*Directed Acyclic Graphs* – DAGs), e a organização dos nós se dá num conjunto de *Destination-Oriented* DAGs (DODAGs), que têm como raiz um nó (DAG *root*), normalmente este sendo o nó *sink*, ou *gateway*, na rede (COLINA, 2016).

Os DODAGs são otimizados utilizando Funções Objetivas (*Objective Functions* – OFs), indicando restrições dinâmicas e métricas, como por exemplo, contagem de saltos, latência, número estimado de transmissões (*Estimated Number of Transmissions* – ETX), conjunto de parentes, consumo de energia, nó com bateria ou não, etc (COLINA, 2016). As OFs definem de que maneira métricas de roteamento e funções relacionadas são empregadas pelos nós para computarem o seu *rank* dentro de uma versão do DODAG (DODAG *Version*) (ANTUNES, 2014). Este *rank* pode ser utilizado para determinar a posição relativa de um nó, e sua distância, quanto à raiz do DODAG.

Dentro de uma dada rede podem existir múltiplas instâncias RPL logicamente independentes, e cada uma delas pode ter diferentes objetivos de otimização. Todos os DODAGs dentro de uma mesma instância RPL devem usar uma mesma OF. Um nó da rede RPL pode pertencer à múltiplas instâncias RPL, podendo atuar como roteador em algumas e como nó folha em outras (COLINA, 2016).

Através do algoritmo Trickle (RFC 6206), o RPL regula as transmissões de mensagens DODAG *Information Object* (DIO), que são mensagens ICMPv6 usadas para construir e manter rotas em sentido à raiz do grafo (*upwards*). Nestas mensagens os nós anunciam sua instância RPL, identificador do DODAG, *rank* do nó, e o número de versão do DODAG. Um nó pode requisitar informações do DODAG através do envio de mensagens DODAG *Information Solicitation* (DIS), solicitando mensagens DIO de suas vizinhanças, para atualizar suas informações de rotas e se juntar à alguma instância RPL. Mensagens chamadas *Destination Advertisement Object* (DAO) são utilizadas para manter as rotas em sentido às folhas (*downward*) (COLINA, 2016).

Além disso, se um nó detecta a inexistência de uma rota em direção à raiz, um mecanismo de reparo local (*Local Repair*) é acionado para encontrar uma rota alternativa. Quando reparos locais sucessivos levarem eventualmente a uma topologia da árvore que não

seja mais eficiente, o nó raiz pode executar um mecanismo de reparo global (*Global Repair*), que atinge a topologia de toda a rede (ANTUNES, 2014).

2.6 O SOM

2.6.1 Definição

Onda sonora é genericamente qualquer onda longitudinal, em que as oscilações acontecem na direção da propagação. Uma onda sonora, como todas as ondas mecânicas, necessita de um meio material para se propagar (ar, água, etc.). O som pode ser descrito como uma sequência de ondas sonoras, com uma velocidade de aproximadamente 343 m/s no ar (HALLIDAY; RESNICK; WALKER, 2009). A Figura 10 apresenta como ocorre a propagação de uma onda sonora.

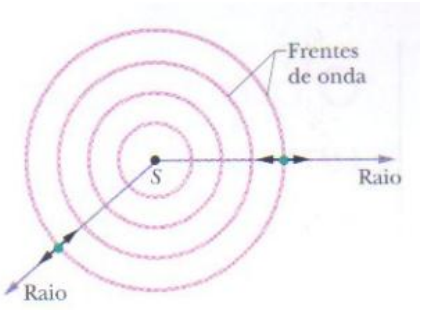

Figura 10 – Propagação de uma onda sonora
Fonte: Halliday, Resnick e Walker (2009, p. 150).

O ponto *S* representa uma fonte sonora, que emite ondas sonoras em todas as direções. As frentes de onda e os raios indicam a direção de propagação e o espalhamento das ondas sonoras. Frentes de onda são superfícies nas quais as oscilações geradas pelas ondas sonoras possuem o mesmo valor. Raios são retas perpendiculares às frentes de ondas que indicam a direção de propagação das frentes de onda. Quando próximas da fonte sonora, as frentes de onda são esféricas e se espalham em três dimensões, enquanto que, quando muito distante da fonte as frentes de onda são planas (HALLIDAY; RESNICK; WALKER, 2009).

2.6.2 Intensidade e Variação na Distância

A onda sonora possui uma intensidade *I*, que é a taxa média por unidade de área com a qual a energia contida na onda atravessa ou é absorvida por uma superfície. A intensidade do som pode ser definida pela seguinte equação (1) (HALLIDAY; RESNICK; WALKER, 2009):

$$I = \frac{P}{A}, \tag{1}$$

onde, *P* é a taxa de variação com o tempo da transferência de energia (potência) da onda sonora e *A* é a área da superfície que intercepta o som.

Em geral, a intensidade do som varia de acordo com a distância de uma fonte real de uma forma bastante complexa. Porém, pode-se supor uma fonte pontual isotrópica (que emite som de mesma intensidade em todas as direções) com uma potência P_s, e então calcular a intensidade *I* da onda sonora em função da distância *r* da fonte (HALLIDAY; RESNICK; WALKER, 2009):

$$I = \frac{P_s}{4\pi r^2} \tag{2}$$

A Equação 2 diz que a intensidade do som emitido por uma fonte pontual isotrópica diminui com o quadrado da distância *r* da fonte (HALLIDAY; RESNICK; WALKER, 2009).

2.6.3 A Escala de Decibéis

O ouvido humano consegue perceber uma faixa muito ampla de intensidades sonoras, aproximadamente uma razão de 10^{12} entre os limites do sistema auditivo humano. Para lidar com um intervalo tão grande de valores, recorre-se ao uso de logaritmos (HALLIDAY; RESNICK; WALKER, 2009). Assim, em vez de se falar da intensidade *I* de uma onda sonora, é muito mais viável utilizar o conceito de nível sonoro β, ou nível de pressão sonora, que é definido como (HALLIDAY; RESNICK; WALKER, 2009):

$$\beta = (10dB)\log\left(\frac{I}{I_0}\right), \tag{3}$$

Em que, dB é a abreviação de decibel, a unidade de nível sonoro. I_0 é uma intensidade de referência (= 10^{-12} W/m^2), cujo valor foi escolhido porque está próximo do limite inferior da

faixa de audição humana. Desta forma, o valor de β aumenta em 10 dB toda vez que a intensidade sonora aumenta de uma ordem de grandeza (um fator de 10) (HALLIDAY; RESNICK; WALKER, 2009).

A Tabela 1 apresenta diferentes níveis de pressão sonora, exemplos de onde podem existir tais níveis, e o tempo máximo de exposição por dia.

Tabela 1 – Tabela de níveis de pressão sonora, exemplos e exposição diária permissível

dB(A)	Exemplo	Exposição diária permissível
180	Lançamento de um foguete	0
160	Disparo de uma escopeta	Menos de 1 segundo
140	Decolagem de um jato	Menos de 30 segundos
130	Martelo pneumático	Menos de 2 minutos
120	**Limiar da dor**	Menos de 7 minutos
115	Show de Rock	15 minutos
110	Clube	30 minutos
105	Grito	1 hora
100	Fábrica	2 horas
95	Metrô	4 horas
90	Trânsito pesado	8 horas
80	Trânsito movimentado	-
70	Restaurante	-
60	Conversação normal	-
50	Residência suburbana	-
40	Auditório quieto	-
30	Sussurro	-
20	Estúdio de gravação	-
10	Câmara anecóica	-
0	**Limiar da audição**	-

Fonte: Adaptado de Occupational Safety & Health Administration[*].

[*] https://www.osha.gov/SLTC/noisehearingconservation/

3 MATERIAIS E MÉTODOS

Neste capítulo são apresentados os materiais de *hardware* e *software*, bem como os métodos que foram utilizados na realização deste trabalho.

3.1 MATERIAIS

3.1.1 *Hardware*

Os materiais de *hardware* utilizados na realização deste trabalho são:

- 3 FRDM-KL25Z – Plataforma de desenvolvimento Kinetis L com microcontrolador ARM® Cortex®-M0+ 32-bit *Ultra-Low Power*, que opera com 48MHz de *clock*, possuindo 128 KB de memória flash, 16 KB de memória SRAM, 16-bit SAR ADC, 2 módulos SPI, 3 módulos UART e 2 módulos I2C;
- 3 Microchip MRF24J40MA – Módulo transceptor de rádio frequência 2.4 GHz compatível com o padrão IEEE 802.15.4 com antena PCB e circuito de *matching* integrados, que utiliza interface SPI de comunicação;
- 1 Módulo transceptor SIM800L EVB para comunicação via GPRS/GSM, que utiliza interface UART de comunicação e responde à um conjunto de comandos AT;
- 1 Módulo receptor GPS ME-1000RW, que utiliza interface UART de comunicação;
- 3 Microfones de Eletreto e circuitos de pré-amplificação.

3.1.2 *Software*

Os materiais de *software* definidos para a realização deste trabalho são:

- CooCox: CoIDE e *tool-chain* livres para desenvolvimento em ARM Cortex M;
- Linguagem de programação C;
- Brazilian Real-Time Operating System (BRTOS);
- Contiki;

- WinPcapSlip6 – para captura de pacotes IPv6 via *Serial Line Internet Protocol* (SLIP);
- XAMPP – Ferramenta que integra servidores de Banco de Dados MySQL e Apache em um único *software*;
- MySQL;
- HTML;
- PHP;
- JavaScript;
- Google Maps API.

3.2 MÉTODO

Durante o desenvolvimento do trabalho foram realizados estudos sobre implementação de redes de sensores sem fio e protocolos de comunicação envolvendo IoT, visando a interoperabilidade entre os dispositivos embarcados ligados à rede. Foram estudados o Contiki e as tecnologias 6LoWPAN e RPL em específico, por serem tecnologias emergentes e intimamente ligadas à IoT e RSSFs, e de código aberto. Este trabalho seguiu um método iterativo e incremental de desenvolvimento e implementação, começando do nó sensor em sua essência até a integração parcial do sistema e, por fim, a integração completa do sistema.

Com o início do projeto, foi desenvolvida a base para um protótipo de nó sensor, utilizando um FRDM-KL25Z com o sistema BRTOS, com duas tarefas instaladas, uma de suporte para o GPS e outra para a aquisição de sinal do sensor. Um circuito de pré-amplificação foi montado, e então realizada a calibração do microfone utilizando um aplicativo de *smartphone*, devido à falta de um decibelímetro. Foram realizados alguns testes de localização e de suporte temporário do GPS, bem como de aquisição do sinal do sensor.

A partir disso, foi realizada a integração do Contiki com o BRTOS. Devido ao fato de o BRTOS permitir a execução de tarefas concorrentes e de tempo real, além da criação de semáforos e mutexes, e ao fato de o escalonador do Contiki funcionar apenas de forma colaborativa, o Contiki foi instalado como uma tarefa dentro do BRTOS. Desta forma pôde-se manter o escalonamento preemptivo prioritário e melhor controle das tarefas.

A partir da instalação do Contiki no BRTOS, foi feito um protótipo de nó sensor com a função de coordenador da rede. Com o uso da ferramenta WinPcapSlip6, foi testado o

exemplo de código *border-router*, que realizou uma ponte entre o computador e a rede IPv6 de nós sensores, que a princípio não tinha nenhum outro nó. Desta forma, pôde-se verificar o funcionamento das pilhas de protocolos implementadas dentro do Contiki. Foram realizados testes de *ping* e de acesso à página *web* embutida na camada de aplicação do nó coordenador.

Depois, foi realizada a adaptação do *driver* do módulo RF para o *hardware* dos nós sensores. Com o *driver* adaptado, foi realizada a prototipação de mais um nó sensor, executando uma *protothread* que executa uma rotina de cliente UDP. Ao mesmo tempo, foi adicionada uma *protothread* que executa uma rotina de servidor UDP no nó coordenador.

Com a etapa anterior concluída, foi feita a primeira comunicação de rede entre nós sensores, com nó coordenador sendo o UDP *server*, e o nó roteador sendo o UDP *client*. Com isso foi possível verificar, através da *protothread border-router* do nó coordenador, que a rede estava sendo formada e a tabela de roteamento do coordenador preenchida, e também, que os dados do cliente UDP do nó sensor estavam sendo recebidos pelo servidor UDP do nó coordenador. Após isso, foi replicado o mesmo código do UDP *client* para um outro protótipo de nó sensor, totalizando 3 nós na rede, o nó coordenador, e dois nós sensores.

A próxima etapa foi realizar a comunicação do nó *border-router* com a rede celular através do módulo GPRS, de forma a poder enviar os dados obtidos para a Internet. Para isso foi criada uma tarefa no BRTOS e as rotinas de inicialização, preparação e comunicação para o módulo. Foi verificada a aquisição de endereço IP a partir da rede celular.

Com a RSSF funcionando e *online*, um banco de dados foi modelado e implementado em um Servidor *Web*, para o armazenamento dos dados da RSSF. A partir disto, foi também implementada uma página *web* que, com acesso ao banco de dados, mostra as informações acústicas em formato de mapa, por meio da API do Google Maps.

Durante o desenvolvimento do trabalho, bem como no final, foram realizados testes envolvendo formação de rotas e encaminhamento de dados pelos nós sensores, e de tempo de envio de dados pelo coordenador da rede até o servidor na Internet, de forma a verificar o cumprimento dos objetivos deste trabalho.

4 DESENVOLVIMENTO

Este capítulo apresenta alguns aspectos da implementação do sistema proposto, dificuldades e problemas encontrados, e descrição de testes realizados durante o desenvolvimento.

4.1 ESCOPO DO SISTEMA

O sistema para o monitoramento e mapeamento em tempo real dos níveis acústicos em ambientes urbanos, obtido como resultado deste trabalho pode ser dividido em: uma rede sem fio de dispositivos embarcados baseada em 6LoWPAN, capazes de se comunicar entre si através do protocolo IP; um servidor de dados na Internet que armazena os dados provindos da rede de sensores; e uma página *web*, que mostra as informações para o usuário visualmente em formato de mapa sonoro. A Figura 11 apresenta uma visão geral do sistema desenvolvido.

O sistema é composto pelos seguintes subsistemas de *hardware*:

- **Sistema Embarcado Gateway** com microcontrolador, sensor de áudio, módulos GSM/GPRS e RF, e suporte temporário ao módulo GPS para aquisição de posição geográfica no momento de implantação. Atua como coordenador e nó primário da RSSF, e envia os dados adquiridos pela rede de sensores ao banco de dados na Internet, que é hospedado pelo Servidor *Web*;
- **Sistema Embarcado Mote** com microcontrolador, sensor de áudio, módulo RF e suporte temporário ao módulo GPS para aquisição de posição geográfica no momento de implantação. Como nó secundário da RSSF, é responsável pela aquisição e envio dos dados sensoriais ao nó principal da rede de sensores;
- **Servidor Web** contendo o banco de dados remoto e a página *web*. Responsável pelo armazenamento e apresentação das informações adquiridas em formato de mapa.

A rede de sensores sem fio tem sua camada de enlace baseada no padrão IEEE 802.15.4 de comunicação, e utiliza o protocolo IPv6 na camada de rede, juntamente com a camada de adaptação para dispositivos com recursos restritos 6LoWPAN, além do protocolo RPL para a topologia de rede e roteamento de pacotes.

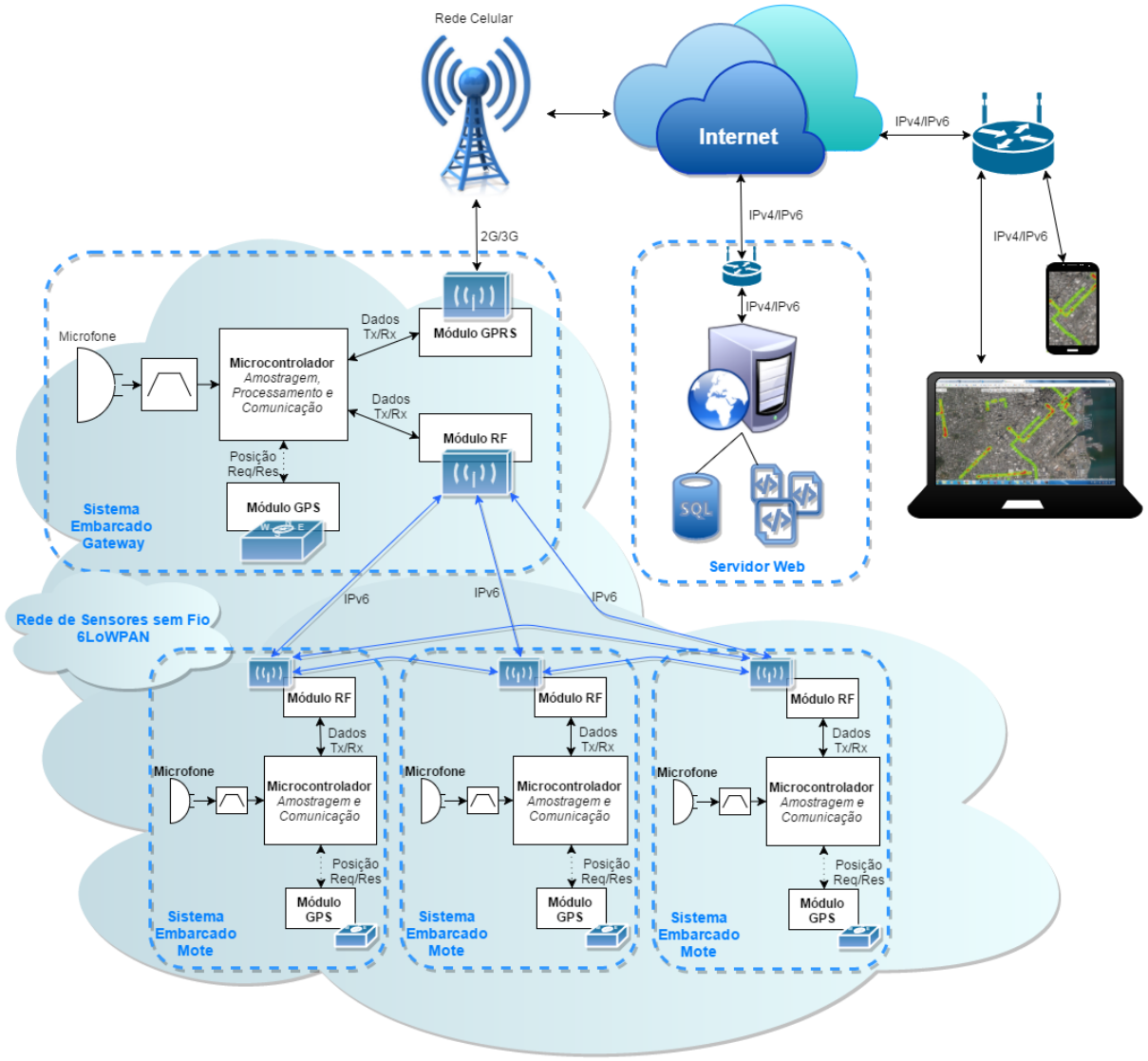

Figura 11 – Visão esquemática do projeto
Fonte: autoria própria.

A rede foi composta por três nós sensores espalhados em diferentes pontos, sendo um deles o coordenador da rede, o qual tem acesso à rede celular via GPRS. Todos os nós sensores têm sua posição determinada através de interface temporária com o GPS. Através de um microfone de eletreto e um circuito de pré-amplificação, os nós sensores realizam a medição da intensidade sonora do ambiente em que estão inseridos e enviam os mesmos para o coordenador da rede. Os dados são enviados através de pacotes UDP pela rede, e são encaminhados até que cheguem ao coordenador da rede.

O coordenador da rede se responsabiliza por enviar os dados, através da rede celular, para um servidor de dados na Internet, através do método GET do HTTP. Neste servidor, todos os dados são armazenados num banco de dados, o qual mantém as informações de todo o

período de vida da rede. O banco de dados é continuamente consultado por um *script* de uma página *web*, que apresenta visualmente as informações atuais no formato de mapa sonoro.

4.2 NÓS SENSORES

O sistema operacional base para gerenciar os recursos do *hardware* de cada um dos nós sensores é o BRTOS, que instala e gerencia tarefas, bem como recursos de memória, semáforos, mutex e filas utilizados no projeto. Uma destas tarefas é o Contiki, que implementa os protocolos de comunicação para LLNs, e encapsula o *driver* do módulo de rádio frequência. A Figura 12 mostra em detalhes a organização do sistema dos nós sensores. A organização geral do projeto no CoIDE pode ser vista no Apêndice A.

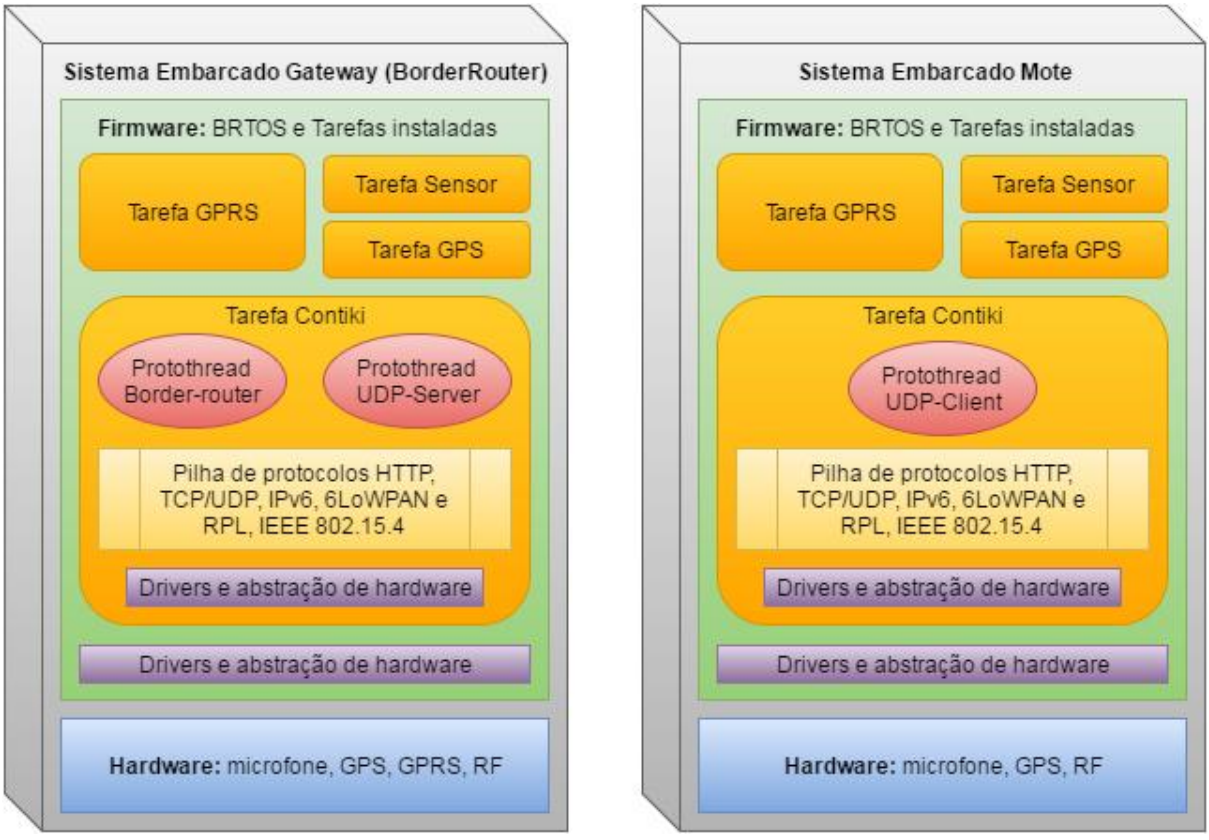

Figura 12 – Diagrama geral dos nós sensores
Fonte: autoria própria.

O BRTOS é configurado a partir de alguns arquivos *header*, e para funcionar em um *hardware* específico é necessário realizar o *port* nos arquivos de abstração de *hardware* do

sistema. O BRTOS já é portado para a plataforma de desenvolvimento utilizada neste trabalho, e não serão detalhados aspectos de *port* de *hardware* para o BRTOS.

A habilitação de recursos do BRTOS foi realizada no arquivo BRTOSConfig.h, bem como a configuração de velocidade de *clock*, tamanho de *heap* a ser alocado para as pilhas das tarefas do sistema e tamanho de *heap* para alocar as filas. Tais definições foram ajustadas com o decorrer do projeto. Neste arquivo também foram configurados aspectos do *hardware* do sistema quanto à ordem dos *bytes* de memória, se é de ordem *Big* ou *Little Endian*. No *hardware* utilizado neste trabalho, a ordem é *Little Endian*. Algumas das configurações existentes nesse *header* podem ser vistas na Listagem 7.

```
#define BOARD_FRDM_KL25Z              2
#define BRTOS_PLATFORM                BOARD_FRDM_KL25Z

/// Define a ordem dos bytes na memória
#define BRTOS_ENDIAN                  BRTOS_LITTLE_ENDIAN
...
/// Define se o watchdog está ativo
#define WATCHDOG                      1
...
/// Define número de prioridades
#define NUMBER_OF_PRIORITIES           32
/// Define o número máximo de tarefas a serem instaladas
#define NUMBER_OF_TASKS         (INT8U)8
...
// Define se a função TimerHook está ativa
#define TIMER_HOOK_EN                 1
...
/// Habilita ou desabilita serviço de temporização
#define BRTOS_TMR_EN                  1
/// Habilita ou desabilita semáforos
#define BRTOS_SEM_EN                  1
#define BRTOS_BINARY_SEM_EN           1
// Habilita ou desabilita mutexes
#define BRTOS_MUTEX_EN                1
// Habilita ou desabilita mailboxes
#define BRTOS_MBOX_EN                 0
// Habilita ou desabilita filas de mensagem
#define BRTOS_QUEUE_EN                1
// Habilita ou desabilita filas dinâmicas
#define BRTOS_DYNAMIC_QUEUE_ENABLED   0
// Habilita ou desabilita filas 16 bits
#define BRTOS_QUEUE_16_EN             0
// Habilita ou desabilita filas 32 bits
#define BRTOS_QUEUE_32_EN             1
// Define o número máximo de semáforos
#define BRTOS_MAX_SEM                 7
// Define o número máximo de mutexes
#define BRTOS_MAX_MUTEX               2
// Define o número máximo de
#define BRTOS_MAX_MBOX                0
// Define o número máximo de filas
#define BRTOS_MAX_QUEUE               2
/// Clock e contagem de Tick do sistema
#define configCPU_CLOCK_HZ        (INT32U)48000000  //< CPU clock in Hertz
```

```
#define configTICK_RATE_HZ          (INT32U)1000      //< Tick timer rate in
Hertz
...
// Tamanho da pilha para a tarefa “Idle”
#define IDLE_STACK_SIZE                (INT16U)192
// Tamanho da Pilha
// 4KB of RAM: 19 * 128 bytes = 2.4KB of Virtual Stack
#define HEAP_SIZE                      30*128
// Tamanho da Pilha para filas
#define QUEUE_HEAP_SIZE                256
```

Listagem 7 – Código de configuração do BRTOS

As tarefas do sistema são instaladas através de uma chamada de função *InstallTask()* do BRTOS, que recebe como parâmetros um ponteiro de função para o código da tarefa, uma *string* com o nome da tarefa, o tamanho em *bytes* a ser alocado para tal tarefa, a prioridade dela, e a identificação numérica, que é opcional. A função deve retornar ‘0’ (OK) para uma instalação bem-sucedida. A Listagem 8 apresenta como foram instaladas as tarefas do sistema.

```
if (InstallTask(&GPS_Temp_Task, "GPS Temporario", 156, 4, NULL) != OK) {
      while (1);
}
if (InstallTask(&Sensor_Task, "Sensor", 156, 4, NULL) != OK) {
      while (1);
}
#ifdef BORDER_ROUTER
if (InstallTask(&GPRS_Task, "GPRS", 512+256 , 3, NULL) != OK) {
      while (1);
}
#endif
if (InstallTask(&contiki_main, "Contiki", (1024*2), 2, NULL) != OK) {
      while (1);
}
```

Listagem 8 – Instalação das tarefas no BRTOS

4.2.1 Tarefa do GPS

Para adquirir a posição via GPS foi criada uma tarefa que continuamente verifica o recebimento de uma *string* através da UART2. O módulo de GPS segue os padrões do protocolo NMEA-0183, e envia automaticamente sequências de *strings*, cada uma com diferentes informações. A *string* analisada neste trabalho tem como início ‘$GPGGA’, que representa *Global Positioning System Fix Data*, e indica informações de latitude e longitude em formato DMS (*Degrees*, *Minutes*, *Seconds*), bem como hora e número de satélites encontrados. Como pode ser visto na Listagem 9, as informações de latitude e longitude são armazenadas pela tarefa, desde que exista comunicação com algum satélite de GPS.

```
while (!UARTGetChar(UART2_BASE, &buf_aux, 5)) {
	buff[p++] = buf_aux;
	if (buf_aux == '\n') {

		buff[p] = '\0';
		str = strstr(buff, "GPGGA");

		if (str != NULL) {
			strtok(str, ","); // $GPGGA
			strtok(NULL, ","); // Hora

			latitude_aux = strtok(NULL, ","); // Latitude

			strtok(NULL, ","); // N - North, S – South

			longitude_aux = strtok(NULL, ","); // Longitude

			strtok(NULL, ","); // W - West, E - East
			strtok(NULL, ","); // 0 - Inválido, 1 - GPS Fix, 2 - DGPS Fix

			nrosat = atoi(strtok(NULL, ",")); // Número de satélites

			if (nrosat > 0) {
				latitude = latitude_aux;
				longitude = longitude_aux;
			}
		}
		break;
	}
}
```

Listagem 9 – Código parcial da tarefa de aquisição do posicionamento

4.2.2 Tarefa do GPRS

A fim de enviar os dados obtidos pela rede de sensores, o coordenador cria uma conexão HTTP através do módulo GPRS e envia as informações por meio do método GET, a partir de uma URL (*Uniform Resource Locator*). Para realizar este procedimento o módulo requer uma sequência de passos de configuração, pois o mesmo funciona a partir de comandos AT, que são passados para o módulo através da UART1. Para realizar os procedimentos de inicialização, configuração e preparação do módulo GPRS, afim de enviar os dados para a Internet, foram criadas algumas funções, descritas a seguir.

A função *cmdSendGPRS()* foi criada para enviar os comandos AT para o módulo através da UART, e aguardar por uma resposta, retornando se foi recebida a resposta esperada ou se houve algum erro. Além do comando em si, essa função recebe como parâmetros um valor de *timeout* em milissegundos, que significa quanto tempo a função deve aguardar até receber um caractere, e também uma *string* que representa a resposta esperada. A função retorna

quando receber toda uma *string*, ou ao atingir o tempo de *timeout*. Como pode ser visto no código da Listagem 10, a função é utilizada para enviar um comando e verificar se o módulo respondeu como esperado antes de acabar um determinado tempo.

```
int initGPRS(void) {
	GPIOPinSet(GPIOD_BASE, GPIO_PIN_7);
	DelayTask(100);
	GPIOPinReset(GPIOD_BASE, GPIO_PIN_7);
	DelayTask(500);
	GPIOPinSet(GPIOD_BASE, GPIO_PIN_7);
	DelayTask(5000);
	cmdSendGPRS("AT\r", (uint16_t) 5000, "");
	DelayTask(100);
	if (!cmdSendGPRS("ATE0\r", (uint16_t) 50, "OK")) {
		return 0;
	}
	DelayTask(100);
	if (!cmdSendGPRS("AT+CMEE=2\r", (uint16_t) 50, "OK")) {
		return 0;
	}
	DelayTask(100);
	if (!cmdSendGPRS("AT+CFUN=1\r", (uint16_t) 50, "OK")) {
		return 0;
	}
	DelayTask(10000);
	return 1;
}
```

Listagem 10 – Função de inicialização do GPRS

A função de inicialização, vista na Listagem 10, realiza um *reset* no módulo a partir do pino 7 da porta D de entrada e saída. Após, é enviado o comando ‘AT’ para iniciar a comunicação com o módulo. Os comandos ‘ATE0’ e ‘AT+CMEE=2’ são para configurar o módulo para responder aos comandos sem enviar eco na resposta e descrever os erros ocorridos, respectivamente, a fim de ajudar no processo de *debug*. O comando ‘AT+CFUN=1’ habilita todas as funcionalidades do módulo, incluindo conexão de dados via Internet, SMS e chamadas. Após a inicialização do módulo é realizada a preparação da conexão do mesmo com a Internet, que pode ser vista em detalhes na Listagem 11.

```
int prepareGPRS(void) {
	cmdSendGPRS("AT+SAPBR=0,1\r", (uint16_t) 1000, "");
	DelayTask(1000);
	if (!cmdSendGPRS("AT+CGREG?\r", (uint16_t) 500, ",1")
			&& !cmdSendGPRS("AT+CGREG?\r", (uint16_t) 500, ",5")) {
		cmdSendGPRS("AT+CGREG=1\r", (uint16_t) 500, "OK");
		return 0;
	}
	DelayTask(2000);
	if (!cmdSendGPRS("AT+SAPBR=3,1,\"CONTYPE\",\"GPRS\"\r", (uint16_t) 500,
			"OK"))
		return 0;
	if (!cmdSendGPRS("AT+SAPBR=3,1,\"APN\",\"zap.vivo.com.br\"\r",
```

```
                    (uint16_t) 500, "OK"))
                return 0;
        DelayTask(100);
        if (!cmdSendGPRS("AT+SAPBR=3,1,\"USER\",\"vivo\"\r", (uint16_t) 500,
"OK"))
                return 0;
        DelayTask(100);
        if (!cmdSendGPRS("AT+SAPBR=3,1,\"PWD\",\"vivo\"\r", (uint16_t) 500,
"OK"))
                return 0;
        DelayTask(100);
        if (!cmdSendGPRS("AT+SAPBR=1,1\r", (uint16_t) 10000, ""))
                return 0;
        DelayTask(5000);
        if (cmdSendGPRS("AT+SAPBR=2,1\r", (uint16_t) 1000, "0.0.0.0")
                        || cmdSendGPRS("AT+SAPBR=2,1\r", (uint16_t) 1000, "ERROR")) {
                cmdSendGPRS("AT+SAPBR=0,1\r", (uint16_t) 1000, "");
                return 0;
        }
        DelayTask(2000);
        return 1;
}
```

Listagem 11 – Código de preparação do GPRS para conexão com a Internet

Os comandos ‘AT+SAPBR’ (Listagem 11) envolvem configurações de rede de dados do módulo, mais especificamente da portadora, ou *bearer*. Na preparação do módulo para se conectar à rede de dados, primeiramente é realizado o comando ‘AT+SAPBR=0,1’ para forçar o desligamento da portadora, caso esteja ligada. Após isso é realizado um teste, para verificar se o módulo já está registrado na rede celular. Caso esteja em *roaming* ou em rede doméstica, o código continua, senão é passado ao módulo o comando para habilitar o registro em rede. A sequência de comandos ‘AT+SAPBR=3,1’ correspondem às configurações do tipo de conexão, que neste caso é GPRS, da APN (*Access Point Name*) a ser utilizada, do usuário e da senha da mesma. Em seguida, através dos comandos ‘AT+SAPBR=1,1’ e ‘AT+SAPBR=2,1’, é ativada a portadora e verificado se o módulo adquiriu um endereço IP na rede celular, respectivamente.

Após realizada a inicialização e preparação do módulo para envio dos dados da RSSF, através dos comandos ‘AT+HTTPINIT’ e ‘AT+HTTPPARA’. Então, é inicializado o serviço HTTP do módulo e passados os parâmetros de portadora, bem como a URL com os dados a serem enviados ao servidor. Uma vez feito isso, submetesse o comando ‘AT+HTTPACTION=0’ para o módulo executar o método GET, e realizar o envio dos dados. Foram realizados alguns testes de envio de dados através do método GET, e o mesmo leva em torno de meio segundo nos melhores casos, até retornar confirmação de sucesso.

Caso haja erro nos comandos HTTP durante os envios de dados, após um determinado número de tentativas, ou caso ocorra algum problema no envio, é realizada a verificação se o

módulo ainda possui endereço IP cadastrado na rede, e caso necessário, é realizada a reinicialização do módulo, repetindo o processo de inicialização e preparação.

Além disso, os *delays* no código foram colocados à fim de evitar problemas quanto à demora do módulo para atender à certos comandos AT, que levam um tempo para serem executados e confirmados.

4.2.3 Tarefa do ADC e Microfone

Para ler o nível acústico foi utilizado um circuito simples de pré-amplificação, como o da Figura 13. Este circuito amplifica o sinal para que o mesmo seja possível de ser digitalizado pelo ADC do microcontrolador. O microfone de eletreto tem uma tensão de saída de aproximadamente 30 mVpp, juntamente com um *offset*, que variam de acordo com a tensão de polarização do circuito, que é de 3 Volts. Com este circuito de pré-amplificação, o sinal de saída fica estabelecido entre 0 e 3 V.

Foi criada uma tarefa que realiza um ciclo de amostragem do sinal através do ADC a cada 10 milissegundos. Neste ciclo são realizadas 400 amostragens do sinal à uma taxa de aproximadamente 400 ksps. O valor lido do ADC é convertido em tensão, e então é transformado para uma escala em decibéis de acordo com uma função adquirida através da calibração do sensor. A calibração foi realizada utilizando um aplicativo de *smartphone* que realiza a leitura da pressão sonora do ambiente, devido à falta de um decibelímetro. Esta tarefa armazena e atualiza o valor de pico do nível sonoro numa variável global, que é acessada e zerada a cada envio de dados.

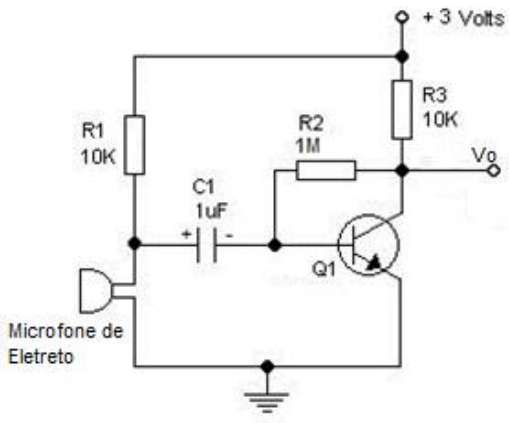

Figura 13 – Circuito de pré-amplificação do sinal do microfone de eletreto
Fonte: autoria própria.

Tal circuito não realiza compensação para variações de ganho devido à temperatura, e foi utilizado neste trabalho para fins de simplificação e redução de custos.

4.2.4 Tarefa do Contiki

Para possibilitar a integração com o sistema operacional Contiki, seu *clock* foi gerado a partir do *Timer Hook* (Listagem 12) do BRTOS, que é habilitado dentro do arquivo BRTOSConfig.h. O *Timer Hook* nada mais é que uma função temporizadora utilizada para contagem de *ticks* de sistema, e foi implementada dentro do código fonte CLOCK.c do Contiki, que faz parte dos arquivos de *port* do Contiki. Sua chamada ocorre toda vez que o BRTOS conta um *tick*, dentro da função OS_TICK_COUNT() que é implementada no arquivo BRTOS.c.

```
unsigned long clock_seconds(void){
  return (unsigned long)(clock_time_t)(clock / configTICK_RATE_HZ);
}

static unsigned long long clock;
void BRTOS_TimerHook(void){
	clock_time_t next_event;
	clock++;
	next_event = etimer_next_expiration_time();
	if(next_event){
		next_event -= clock_time();
		if (next_event == 0)
		{
			etimer_request_poll();
		}
	}
}
```

Listagem 12 – *Clock* do Contiki gerado através do *Timer Hook* do BRTOS

Para realizar alguns testes com o Contiki que auxiliaram no desenvolvimento deste trabalho foi necessário o uso do protocolo SLIP, que é dependente da comunicação serial da plataforma. *Serial Line Internet Protocol* (SLIP) é um encapsulamento dos protocolos de Internet feito para funcionar através de portas seriais e conexões de modems. O SLIP é documentado pela RFC 1055.

O protocolo já vem implementado pelo Contiki, e bastou realizar a adaptação para o *hardware*, através do arquivo *slip-arch.c*, e modificar o código de interrupção da UART do BRTOS para ser utilizada pelo SLIP, dentro do arquivo *uart.c* do projeto. Desta forma, quando o SLIP é iniciado, inicializa-se também a UART. A função de envio de *bytes* utilizada pelo

SLIP chama dentro de si a função de envio de *bytes* implementada no BRTOS, como pode ser visto na Listagem 13.

```
#include "uart.h"
...
void slip_arch_init(unsigned long ubr){

	(void)ubr;
	Init_UART0(115200, 0);
}

void slip_arch_writeb(unsigned char c){
	(void)UARTPutChar(0x4006A000, (char)c);
}
```

Listagem 13 – Código de adaptação de *hardware* para a interface SLIP

O *handler* de interrupção da UART0, como pode ser visto na Listagem 14, é responsável por repassar diretamente o *byte* recebido ao *buffer* do protocolo.

```
unsigned long USART0IntHandler(void *pvCBData, unsigned long ulEvent, unsigned
long ulMsgParam, void *pvMsgData){
	char receive_byte;
	if ((ulEvent & UART_EVENT_RX) == UART_EVENT_RX)
	{
		receive_byte = xHWREGB(UART0_BASE + UART_012_D);
		slip_input_byte(receive_byte);
	}
	...
	// Interrupt Exit
	OS_INT_EXIT_EXT();
	return 0;
}
```

Listagem 14 – Código de interrupção da UART0 para a interface SLIP

Através do SLIP pôde ser validada a integração do Contiki com o BRTOS e, também, a comunicação entre os nós da rede. Com o SLIP, é possível visualizar informações de DEBUG dos protocolos do Contiki em um terminal, bem como realizar testes de *ping* para com os nós da rede. Para conseguir isto, também foi necessário criar um túnel IPv6 na interface de rede de *Loopback* do computador, utilizando um prefixo de rede *aaaa::1/64*, para que, através do *software* WinPcapSlip6, pudesse ser feita a ponte com a rede de sensores via porta serial.

Para definir os protocolos da pilha de comunicação do Contiki foram realizadas algumas definições gerais, que estão resumidas na Listagem 15.

```
// Habilita UDP
#define UIP_CONF_UDP                    1
// Habilita TCP
#define UIP_CONF_TCP                    1
// Pilha de rede com IPv6
#define NETSTACK_CONF_WITH_IPV6         1
// Protocolo de roteamento RPL
```

```
#define UIP_CONF_IPV6_RPL               1
// Roteador
#define UIP_CONF_ROUTER                 1
// 6LoWPAN driver
#define NETSTACK_CONF_NETWORK           sicslowpan_driver
// Mecanismo de compressão 6LoWPAN HC06
#define SICSLOWPAN_CONF_COMPRESSION     SICSLOWPAN_COMPRESSION_HC06
// Habilita fragmentação dos pacotes
#define SICSLOWPAN_CONF_FRAG            1
// Framer 802.15.4
#define NETSTACK_CONF_FRAMER            framer_802154
// Medium Access Control driver
#define NETSTACK_CONF_MAC               nullmac_driver
// Radio Duty Cycle driver
#define NETSTACK_CONF_RDC               nullrdc_driver
// Driver do módulo RF
#define NETSTACK_CONF_RADIO             mrf24j40_driver
```

Listagem 15 – Configurações básicas da pilha de protocolos no Contiki

O coordenador da RSSF funciona com duas *protothreads* principais, uma que possibilita a interface via SLIP e realiza a definição do nó como uma raiz DODAG, que é a *border-router*, e outra que é responsável pelo servidor UDP e serve para o recebimento dos dados da rede, denominada *udp-server*.

Basicamente, o *border-router* aguarda por um prefixo ser recebido através da interface SLIP para definir-se como sendo uma raiz DODAG, através da função *rpl_set_root()* (Listagem 16). Uma vez que o prefixo for recebido, o *border-router* é estabelecido como a raiz DODAG, e a partir disso define também o prefixo do resto dos nós da rede automaticamente.

```
PROCESS_THREAD(border_router, ev, data)
{
  static struct etimer et;
  rpl_dag_t *dag;
  PROCESS_BEGIN();
  prefix_set = 0;
  NETSTACK_MAC.off(0);
  PROCESS_PAUSE();
#if 1
  while(!prefix_set) {
    etimer_set(&et, CLOCK_SECOND);
    request_prefix();
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
  }
#endif
  dag = rpl_set_root(RPL_DEFAULT_INSTANCE,(uip_ip6addr_t *)dag_id);
  if(dag != NULL) {
    rpl_set_prefix(dag, &prefix, 64);
  }
  NETSTACK_MAC.off(1);
  while(1) {
    PROCESS_YIELD();
    if (ev == sensors_event){
      rpl_repair_root(RPL_DEFAULT_INSTANCE);
    }
```

```
  }
  PROCESS_END();
}
```

Listagem 16 – *Protothread* do processo *border-router*

A *protothread* do servidor UDP pode ser vista na Listagem 17, e funciona como uma aplicação qualquer utilizando *sockets* UDP. Através de um *protosocket* UDP, a porta 3001 é aberta e ouvida para a comunicação vinda dos clientes para com o servidor, e a porta 3000 para comunicação do servidor com os clientes.

```
PROCESS_THREAD(udp_server, ev, data) {
	PROCESS_BEGIN();
	server_conn = udp_new(NULL, UIP_HTONS(3001), NULL);
	udp_bind(server_conn, UIP_HTONS(3000));
	while (1) {
		PROCESS_YIELD();
		if (ev == tcpip_event) {
			tcpip_handler();
		}
	}
	PROCESS_END();
}
```

Listagem 17 – *Protothread* do processo *udp-server*

Quando um evento *tcpip_event* ocorrer, a *protothread* chama a função *tcpip_handler()* (Listagem 17 e Listagem 18). Esta função verifica se há novos pacotes de dados recebidos através da função *uip_newdata()* (Listagem 18), e caso haja novos pacotes de dados realiza, então, o processamento dos mesmos.

```
static void tcpip_handler(void) {
	static int seq_id;
	if (uip_newdata()) {
		((char *) uip_appdata)[uip_datalen()] = 0;
		uip_ip6addr_t *addr = &UIP_IP_BUF->srcipaddr;
		char info[64];
		unsigned long seq_num;
		unsigned long mote_ident;
		sscanf((char *) uip_appdata, "%ld %ld %s", &mote_ident, &seq_num,
				&info[0]);
		sprintf(&bufsend[mote_ident][0],
		"ip6=%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x
		:%02x%02x&sqn=%ld&%s",
			((uint8_t *) addr)[0], ((uint8_t *) addr)[1],
			((uint8_t *) addr)[2], ((uint8_t *) addr)[3],
			((uint8_t *) addr)[4], ((uint8_t *) addr)[5],
			((uint8_t *) addr)[6], ((uint8_t *) addr)[7],
			((uint8_t *) addr)[8], ((uint8_t *) addr)[9],
			((uint8_t *) addr)[10], ((uint8_t *) addr)[11],
			((uint8_t *) addr)[12], ((uint8_t *) addr)[13],
			((uint8_t *) addr)[14], ((uint8_t *) addr)[15], seq_num,
			info);
		uip_ipaddr_copy(&server_conn->ripaddr, &UIP_IP_BUF->srcipaddr);
```

```
            memset(&server_conn->ripaddr, 0, sizeof(server_conn->ripaddr));
        }
}
```

Listagem 18 – Função de manipulação de dados do processo *udp-server*

O cliente UDP envia informações de sequência de dados da camada de aplicação, latitude, longitude e o valor medido do microfone, bem como seu número de identificação atribuído na camada de aplicação MOTE_ID. O servidor UDP, durante o processamento da mensagem recebida, monta uma *string* que mais tarde é usada na tarefa do GPRS. Tal *string* contém o endereço IPv6 do nó sensor do qual a informação veio, e os dados vindos de tal nó. A *string* já é montada no formato que o GPRS utilizará para realizar o método GET.

A *protothread* do cliente UDP (Listagem 19) também funciona como uma aplicação comum utilizando *sockets*. É feita a definição do endereço de conexão através da função *set_connection_address()*. Então, o cliente UDP começa a ouvir a porta 3000, passando o endereço do servidor UDP. Após isso, abre-se a porta 3001 para envio dos dados. Como pode ser visto na Listagem 19, é definido um evento de *timer*, o qual ocorre periodicamente de acordo com a definição *SEND_INTERVAL*. Toda vez que ocorre esse evento de temporização, a *protothread* chama a função *timout_handler()* (Listagem 20), que envia os dados.

```
PROCESS_THREAD(udp_client, ev, data)
{
  static struct etimer et;
  uip_ipaddr_t ipaddr;
  PROCESS_BEGIN();
  set_global_address();
  set_connection_address(&ipaddr);
  client_conn = udp_new(&ipaddr, UIP_HTONS(3000), NULL);
  udp_bind(client_conn, UIP_HTONS(3001));
  etimer_set(&et, SEND_INTERVAL);
  while(1) {
    PROCESS_YIELD();
    if(etimer_expired(&et)) {
      timeout_handler();
      etimer_restart(&et);
    }
  }
  PROCESS_END();
}
```

Listagem 19 – *Protothread* do processo *udp-client*

Quando o evento de temporização ocorre, o cliente UDP prepara os dados de sequência de pacotes, latitude, longitude, e o mais alto nível acústico medido entre os ciclos de envio, que neste trabalho são definidos com um período de 1 segundo, dentro de um *buffer*, que é passado

para a função de envio do pacote UDP. O pacote com os dados é então enviado de acordo com as configurações do *protosocket*.

```
static void timeout_handler(void){
  static int seq_id;
  sprintf(buf, "%ld %ld lat=%s&lng=%s&spl=%d", MOTE_ID, ++seq_id, latitude,
longitude, spl);
  spl = 0;
  uip_udp_packet_send(client_conn, buf, strlen(buf));
}
```

Listagem 20 – Função de manipulação de dados do processo *udp-client*

Durante a implementação desta parte do trabalho, houve vários problemas para fazer funcionar a comunicação entre os *motes*. Um destes problemas era por causa do *driver* do rádio. O *driver* foi escrito para um sistema com ordem de *bytes Big Endian*, enquanto que o microcontrolador utilizado neste trabalho é *Little Endian*. Através do *debug* da comunicação SPI do microcontrolador com o rádio, foi descoberto o problema. O rádio era programado para utilizar determinados PAN ID e endereço de camada de enlace, além de outros parâmetros. Tais parâmetros são configurados dentro do Contiki, e quando o mesmo realizava a desfragmentação dos pacotes, descartava-os pelo fato de os parâmetros não casarem, pois, o rádio foi configurado com os parâmetros de rede corretos, porém, com os *bytes* trocados. Com alguns ajustes feitos no *driver* do rádio o problema foi resolvido.

4.3 BANCO DE DADOS

O banco de dados foi modelado pensando nas informações que cada nó sensor provê, como a sua localização, seu endereço de IP, o valor medido a partir do microfone e a sequência de dados da aplicação. Também foi dividido o banco de dados em duas tabelas, em que uma armazena os dados em tempo real, e a outra armazena o histórico dos dados.

Primeiramente foi necessária a criação da instância do banco de dados e do usuário a ser responsável pelos *selects, inserts* e *updates* na tabela de dados atuais, como na Listagem 21, para que desta forma pudesse ser realizada a consulta e inserção/atualização dos dados pelos *scripts* PHP, que serão descritos no próximo tópico.

```
CREATE DATABASE IF NOT EXISTS spl_map;
CREATE USER IF NOT EXISTS 'datahandler'@'localhost' IDENTIFIED BY
'spl_map_123';
GRANT USAGE, SELECT, INSERT, UPDATE ON spl_map.* TO
datahandler@'localhost' IDENTIFIED BY 'spl_map_123';
```

Listagem 21 – Criação da instância do banco de dados, do usuário e seus privilégios

Em seguida, os seguintes tipos de informação foram considerados na criação das tabelas: seu IPv6 na rede, sua localização global em latitude e longitude, o valor medido do sensor, o número de sequência do dado no nível da aplicação, e o momento em que a informação entrou no banco de dados. A partir disso então foi criada a tabela *spl_map_rt* (Listagem 22).

```
CREATE TABLE spl_map_rt
(
ip6 CHAR(40) PRIMARY KEY NOT NULL,
sqn BIGINT NOT NULL,
lat DOUBLE NOT NULL,
lng DOUBLE NOT NULL,
spl DOUBLE NOT NULL,
ts TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
);
```

Listagem 22 – Criação da tabela de dados atuais

A outra tabela foi chamada *spl_map_hist*, e foi criada com o intuito de armazenar os dados antigos de forma que pudessem ser acessados e analisados em caso de interesse, de forma a levantar informações e relatórios diários, por exemplo. A criação da tabela pode ser visualizada na Listagem 23.

```
CREATE TABLE spl_map_hist
(
id BIGINT PRIMARY KEY NOT NULL AUTO_INCREMENT,
ip6 CHAR(40) NOT NULL,
sqn BIGINT NOT NULL,
lat DOUBLE NOT NULL,
lng DOUBLE NOT NULL,
spl DOUBLE NOT NULL,
ts TIMESTAMP NOT NULL
);
```

Listagem 23 – Criação da tabela de histórico de dados

Para que os dados históricos fossem armazenados toda vez que ocorresse uma atualização na tabela de dados atuais foi criada uma *trigger*, como na Listagem 24, que executa toda vez após uma linha da tabela *spl_map_rt* ser atualizada.

```
CREATE TRIGGER spl_update AFTER UPDATE ON spl_map_rt
FOR EACH ROW
BEGIN
IF NEW.ts <> OLD.ts THEN
INSERT INTO spl_map_hist (ip6, sqn, lat, lng, spl, ts) VALUES
(OLD.ip6, OLD.sqn, OLD.lat, OLD.lng, OLD.spl, OLD.ts);
END IF;
END;
```

Listagem 24 – *Trigger* de atualização sobre a tabela de dados atuais

Desta forma, sempre que houver um novo *update* na tabela que armazena os dados em tempo real, os dados antigos são salvos na tabela de histórico.

4.4 *SCRIPTS*

Para possibilitar o método GET, o qual permite ao coordenador da rede de sensores inserir os dados no banco de dados do servidor, foram criados dois *scripts* em PHP. Um deles é para realizar a conexão com a instância do banco de dados criado, utilizando os dados do usuário criado, como pode ser visto na Listagem 25. Como *hostname* foi utilizado *localhost*, pois o servidor de banco de dados do sistema foi criado no mesmo enlace em que o servidor Apache está.

```
$usuario = "datahandler"; // Usuário do Banco MySQL
$senha = "spl_map_123";   // Password do Usuário
$hostname = "localhost";  // Usualmente é "localhost"
$banco = "spl_map";       // Banco de Dados

$dbh = mysqli_connect($hostname, $usuario, $senha, $banco);
if (mysqli_connect_errno()){
    echo "Failed to connect to MySQL: " . mysqli_connect_error();
}
$selected = mysqli_select_db($dbh, $banco);
```

Listagem 25 – *Script* em PHP de conexão ao banco de dados

O outro *script* realiza a manipulação do método GET em si. Primeiro, o *script* realiza a conexão e seleção do banco de dados através do *script* da Listagem 25. Depois, através dos parâmetros passados pela URL, o *script* realiza uma *query* ao banco de dados para inserção de novos dados ou então atualização de dados existentes. Como pode ser visto na Listagem 26, o *script* também realiza uma conversão dos valores de latitude e longitude passados para o formato DD (*Decimal Degrees*), pois a API do Google Maps não suporta o formato DMS.

```
include("dbconn.php");
$IP6 = $_GET["ip6"];
$SQN = $_GET["sqn"];
$LAT = $_GET["lat"];
$LNG = $_GET["lng"];
$SPL = $_GET["spl"];
$LAT = (($LAT - fmod($LAT,100))/100 + fmod($LAT,100)/60)*-1;
$LNG = (($LNG - fmod($LNG,100))/100 + fmod($LNG,100)/60)*-1;
```

Listagem 26 – Aquisição dos dados passados pelo método HTTP GET

Para verificar o que deve ser feito, primeiro é realizada uma consulta ao banco na tabela de dados atuais para verificar a existência ou não do endereço IPv6 do nó sensor, ao qual as informações passadas pertencem. Se já existe este endereço de IP na tabela, significa que o cadastro do nó sensor já existia, bastando apenas atualizar seus dados atuais. Caso não exista, então é inserido como um novo registro na tabela de dados atuais, como se pode ver na Listagem 27.

```
$r = mysqli_query($dbh, "SELECT * FROM spl_map_rt WHERE ip6 =
'".$IP6."'");

if(mysqli_fetch_array($r) == true)
{
    $r = mysqli_query($dbh, "UPDATE spl_map_rt SET spl =
    '".$SPL."', lat = '".$LAT."', lng = '".$LNG."', sqn =
    '".$SQN."' WHERE ip6 = '".$IP6."'");
        echo "\nJá existe, então update!\n";
}
else
{
    $r = mysqli_query($dbh, "INSERT INTO spl_map_rt (ip6, sqn, lat,
    lng, spl, ts) VALUES ('".$IP6."', ".$SQN.", ".$LAT.", ".$LNG.",
    ".$SPL.", NOW())");
        echo "\nNão existe, então insert!\n";
}
mysqli_close($dbh);
```

Listagem 27 – Inserção ou atualização de tuplas na tabela de dados atuais

A página *web* do sistema que apresenta as informações foi construída em torno da API do Google Maps, utilizando *JavaScript*. A função *initMap()* (Listagem 28) faz a inicialização do mapa e realiza a chamada da função *getMarkers()*, que periodicamente executa o método *post()* via jQuery AJAX (*Asynchronous JavaScript and XML*), como pode ser visto mais adiante na Listagem 30.

A variável *map* representa a instância do mapa apresentado na página, e contém parâmetros como *zoom*, posição central inicial e o tipo de mapa a ser apresentado inicialmente. As variáveis *heatmaps* e *hm_check* são vetores auxiliares para armazenar e indexar dados que já estão inseridos no mapa, bastando, desta forma, apenas atualizar os dados dos pontos já existentes.

```
var map;
var heatmaps = new Array();
var hm_check = new Array();
function initMap() {
  map = new google.maps.Map(document.getElementById('map'), {
    zoom: 14,
    center: {lat: -26.222204, lng: -52.671463},
    mapTypeId: google.maps.MapTypeId.ROADMAP
```

```
  });
  getMarkers();
}
```

Listagem 28 – Função de inicialização do mapa e variáveis globais

O método *post()* realiza a execução do *script* em PHP chamado *update.php*, que retorna informações da tabela de dados atuais codificados em formato JSON (*JavaScript Object Notation*), como mostrado na Listagem 29. Desta forma, o mapa é atualizado com as informações mais atuais sem a necessidade de realizar *refresh* da página *web*.

```
require("dbconn.php");
$query = "SELECT spl, lat, lng, ip6 FROM spl_map_rt";
$result = mysqli_query($dbh, $query) or die(mysqli_error($dbh));
$returnArray = array();
while($row = mysqli_fetch_array($result)){
    $spl = $row['spl'];
    $latitude = $row['lat'];
    $longitude = $row['lng'];
    $ip6 = $row['ip6'];
    array_push($returnArray,array($spl,$latitude,$longitude,
$ip6));
}
$returnString = json_encode($returnArray);
echo $returnString;
```

Listagem 29 – *Script* para consulta ao banco de dados e retorno em formato JSON

A verificação da existência, ou não, dos pontos é feita através do retorno da função *inArray()* (Listagem 30), que recebe como parâmetros um endereço IPv6, e o vetor de indexação *hm_check*. A função retorna o índice do ponto, caso já exista, e -1 caso não exista. Se o endereço IPv6 já existir no vetor de indexação *hm_check*, então basta atualizar as informações do ponto, que está armazenado no vetor *heatmaps*, no mesmo índice retornado.

```
function getMarkers() {
  $.post('update.php', {},
    function(data) {
       for(var i=0; i < data.length; i++){
           var spl = data[i][0];
           var lat = data[i][1];
           var lng = data[i][2];
           var ip6 = data[i][3];
           var spl_rate = (spl-45)/55;
           if(spl_rate > 1) spl_rate = 1;
           else if(spl_rate < 0) spl_rate = 0;
           var found = $.inArray(ip6, hm_check);
           if(found == -1) {
             var heatmap = new
             google.maps.visualization.HeatmapLayer({
                data: [{location: new google.maps.LatLng(lat,lng),
                weight: 200}], radius: 50, map: map, dissapating:
```

```
                false, gradient: [getColor(spl_rate) + ', 0)',
                getColor(spl_rate) + ', 1)']
              });
              hm_check.push(ip6);
              heatmaps.push(heatmap);
            }else{
              heatmaps[found].set('data', [{location: new
              google.maps.LatLng(lat,lng), weight: 200}]);
              heatmaps[found].set('gradient', [ getColor(spl_rate)
              + ', 0)', getColor(spl_rate) + ', 1)']);
            }
        }
    },"json");
  setTimeout(getMarkers, 500);
}
```

Listagem 30 – Função de atualização periódica do mapa em JavaScript

5 RESULTADOS

Este capítulo apresenta o que foi obtido como resultado deste trabalho e as experiências do autor durante o desenvolvimento do sistema.

5.1 APRESENTAÇÃO

O sistema apresenta as informações atuais através da habitual interface gráfica do Google Maps. As informações são pontos no mapa que apresentam uma determinada cor, verde para baixa intensidade acústica e vermelho para alta intensidade acústica. A cor dos pontos vai mudando conforme os dados recebidos da RSSF em tempo real. Para níveis aproximados a 50 dB a cor é apresentada como verde, e segue uma função linear crescente de gradiente de cores conforme o aumento dos níveis acústicos, do verde ao amarelo, e do amarelo ao vermelho. Uma visão da interface *web* utilizando dados simulados pode ser vista na Figura 14.

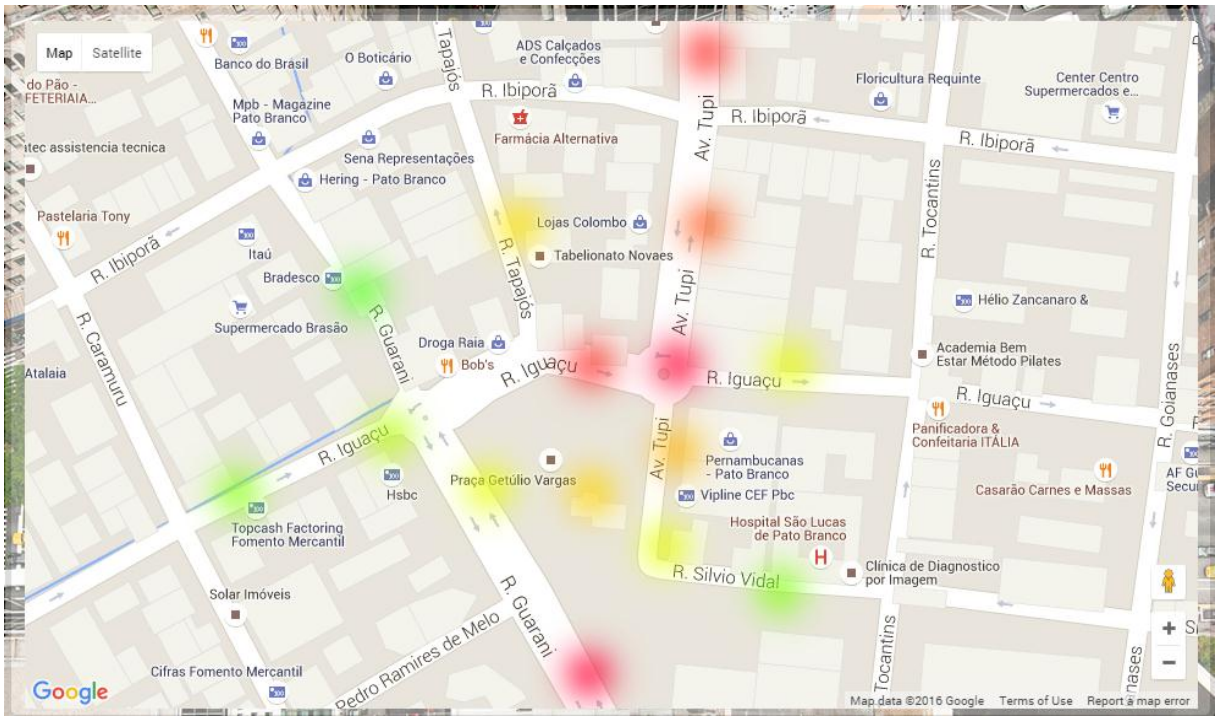

Figura 14 – Interface gráfica baseada no Google Maps com dados simulados
Fonte: autoria própria.

Através da interface SLIP do coordenador da rede, é possível visualizar informações da rede de sensores sem fio. Pode-se obter informações das rotas existentes e dos nós sensores

vizinhos pertencentes à rede, por meio de um *webserver* embutido no *border-router*. Na Figura 15, pode-se verificar a existência de uma rota com dois saltos de distância.

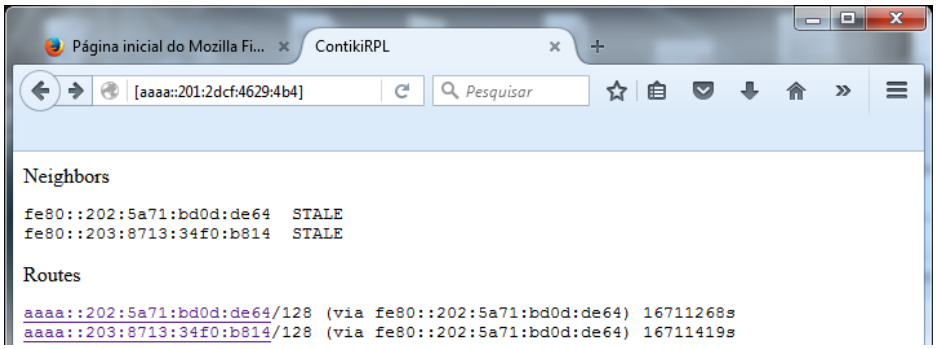

Figura 15 – Visualização da tabela de vizinhos e rotas
Fonte: autoria própria

Além disso, a interface SLIP permite analisar pacotes e eventos da rede de sensores através do terminal, como na Figura 16.

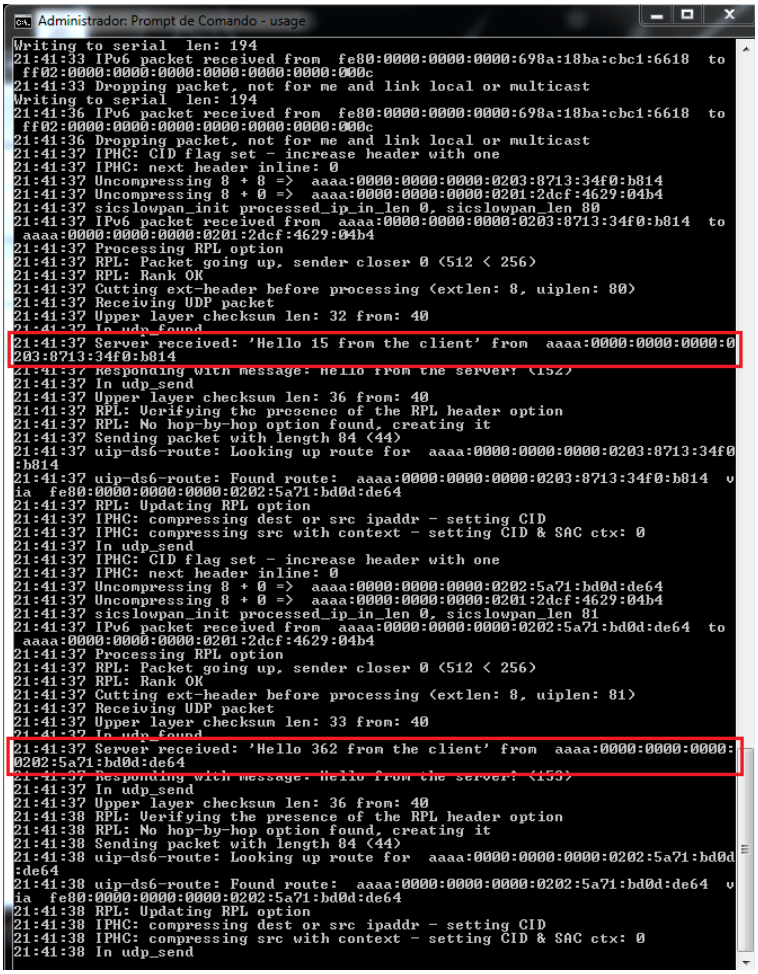

Figura 16 – Debug da rede via interface SLIP
Fonte: autoria própria.

5.2 DISCUSSÃO

Os resultados obtidos até o fim deste trabalho mostraram que a rede funciona, e que os pacotes são enviados efetivamente da origem até o destino, mesmo em múltiplos saltos. Na maior parte do tempo em que foi desenvolvida, a rede foi testada em um ambiente interno, mostrando alcance de sinal entre nós sensores, nos piores casos, de 10 a 15 metros de distância com uma parede no meio, apresentando uma pequena perda de eficiência na comunicação sem fio. Os protocolos para redes *lossy* se mostram confiáveis quanto a manter a rede funcional mesmo em situações críticas, quando a taxa de perdas é grande.

Quanto à recursos, o *firmware* dos nós sensores ocupou no melhor caso aproximadamente 13,5 KB de 16 KB de memória RAM nos nós sensores, e 15,7 KB para o coordenador da rede, mais do que 80% dos recursos de memória RAM. O Contiki foi responsável pela ocupação da maior parte dos recursos de memória em geral. Apesar de ter sobrado memória, para um sistema com recursos restritos essa quantidade de memória é grande.

A rede celular via GPRS se mostrou mais lenta do que se era esperado quando este trabalho foi proposto. No melhor caso, o coordenador da RSSF consegue enviar os dados de um único nó sensor para o servidor *web* em pouco menos de meio segundo. É uma latência elevada para uma aplicação que demanda por atualizações em tempo real. Desta forma, conforme a escala da rede sem fio de sensores aumenta, aumenta também o tempo para atualizar os dados de todos os nós sensores no servidor *web*.

A integração do Contiki com o BRTOS possibilitou um melhor controle das tarefas de comunicação com a rede celular e leitura do sensor. Num sistema puramente cooperativo, os atrasos no envio dos dados para o servidor *web* ocasionariam perdas em eficiência na comunicação entre o nó coordenador e os demais nós da rede de sensores sem fio.

Devido à falta de um decibelímetro, a calibração do microfone não foi adequada. Mesmo assim, o sistema cumpre com o objetivo do trabalho.

6 CONCLUSÃO

Este trabalho resumiu-se em desenvolver um sistema baseado em redes de sensores sem fio, com o objetivo de monitorar os níveis acústicos em meio urbano e mostrar as informações no formato de mapa sonoro. O trabalho buscou focar no uso dos conceitos de RSSFs e de tecnologias emergentes em IoT, as quais foram possíveis de serem utilizadas através do sistema operacional Contiki, que foi integrado ao BRTOS.

Pôde-se concluir que a rede de sensores sem fio, utilizando as tecnologias emergentes baseadas em protocolo IP para dispositivos com recursos restritos, atinge os objetivos quanto ao sistema. A formação da rede, comunicação e a troca contínua de pacotes IPv6 ocorreu como o esperado. Entretanto, apesar de as tecnologias empregadas serem para dispositivos com recursos restritos, o *firmware* utiliza uma relevante quantidade de memória RAM dos microcontroladores. Também, o fato de os módulos RF trabalharem numa frequência de 2.4 GHz não agrada *a posteriori*. Numa aplicação real onde o sistema abrangesse todo o bairro de uma cidade, por exemplo, sinais de WiFi poderiam causar problemas constantes de interferência na RSSF, prejudicando a sua eficiência. Para solucionar este e outros problemas de interferência, bem como melhorar o alcance de sinal dos nós sensores, módulos de RF que trabalhem em frequências sub GHz podem ser utilizados.

As vantagens de se utilizar os protocolos IP em redes de objetos inteligentes são evidentes, e a padronização que possibilita a IoT converge para os protocolos de Internet. Porém, a infraestrutura pode ser uma grande vilã contra o avanço tecnológico quando o objetivo é implantar um sistema em larga escala como a ideia que este trabalho traz. A solução escolhida neste trabalho para interligar a rede de sensores com a Internet funciona, porém, se mostra lenta quando comparada às capacidades da rede de sensores sem fio, mesmo em pequena escala. Para uma aplicação onde o objetivo é o monitoramento em tempo real, a baixa latência é requisito fundamental entre o berço dos dados e o berço da informação. O uso de módulos que suportem tecnologias de terceira e quarta geração poderia resolver este problema.

Devido ao processo inadequado de calibração, pela ausência de um decibelímetro, não é possível concluir se os valores em decibéis armazenados pelo sistema correspondem exatamente aos valores exatos na realidade, pois os dados adquiridos através dos microfones não podem ser considerados fielmente descritivos quanto aos níveis de pressão sonora. Entretanto, mesmo em condições de baixa acurácia e precisão, o sistema permite detectar níveis intensos de som, auxiliando, portanto, na detecção de fontes de som poluidoras em tempo real.

REFERÊNCIAS

AHMED, Ahmed A.; SHI, Hongchi; SHANG, Yi. **A Survey on Network Protocols for Wireless Sensor Networks**. IEEE International Conference on Information Technology: Research and Education (IEEE ITRE), p. 301-305, ago. 2003.

AKYILDIZ, Ian F.; SU, Weilian; SANKARASUBRAMANIAM, Yogesh; CAYIRCI, Erdal. **A Survey on Sensor Networks**. IEEE Communications Magazine, v. 40, n. 8, p. 102-114, ago. 2002.

ANTUNES, Vinicius B. **Roteamento Sensível ao Contexto em Redes de Sensores sem Fio: Uma abordagem baseada em Regras de Aplicação para o Protocolo RPL**. 2014. 80f. Dissertação (Mestrado em Informática) – Programa de Pós-Graduação em Informática, Universidade Federal do Espírito Santo, 2014. Disponível em: < http://portais4.ufes.br/posgrad/teses/tese_7981_Disserta%E7%E3o%20-%20Vinicius%20Barcellos%20Antunes20150602-133738.pdf>. Acesso em: 20 jun. 2016.

ATZORI, Luigi; IERA, Antonio; MORABITO, Giacomo. **The Internet of Things**: A survey. Computer Networks: The International Journal of Computer and Telecommunications Networking, v. 54, n. 15, p. 2787-2805, 2010.

BASENER, Mathias, et al. Auditory and non-auditory effects of noise on health. **The Lancet**, v. 383, n. 9925, p. 1325-1332, 2014.

BIELSA, Alberto. **The Smart City Project in Santander**. Libelium Sensors (Mar. 2013). Disponível em: <http://www.sensorsmag.com/wireless-applications/smart-city-project-santander-11152>. Acesso em: 21 nov. 2015.

BRAMBILLA, Giovanni. Noise and soundscape in Rome. **The Journal of the Acoustical Society of America**, v. 115, n. 5, p. 2591-2591, 2014.

CAVALCANTE, Waldek Fachinelli. Poluição sonora é crime. **Revista Jus Navigandi**, Teresina, ano 17, n. 3193, 29 mar. 2012. Disponível em: <http://jus.com.br/artigos/21382>. Acesso em: 31 ago. 2015.

CENEDESE, Angelo, et al. **Padova Smart City**: an urban Internet of Things experimentation. IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (IEEE WoWMoM), p. 19-19, jun. 2014.

CHRISTENSEN, Brad. **Building a Network for the Internet of Things with the Constrained Application Protocol**. Monografia (Bacharelado em Ciências da Computação e Matemática) – Universidade de Waikato, Hamilton, Nova Zelândia, 2014.

COLINA, Antonio L. et al. **IoT in five Days**. 1. ed. 2016. Disponível em: <http://github.com/marcozennaro/IPv6-WSN-book/releases>. Acesso em: 15 jun. 2016.

COMPARISON of real-time operating systems. In: Wikipédia: a enciclopédia livre. Disponível em: <http://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems>. Acesso em: 1 nov. 2015.

CONTIKI-OS, Thingsquare – Contiki: The Open Source OS for the Internet of Things. Disponível em: <http://www.contiki-os.org>. Acesso em: 15 jun. 2016.

DUNKELS, Adams; VASSEUR, Jean-Philippe. **Interconnecting Smart Objects with IP**: the next Internet. 1 ed. Burlington, MA: Elsevier, 2010.

HALLIDAY, David; RESNICK, Robert; WALKER, Jearl. **Fundamentos de Física, volume 2**: gravitação, ondas e termodinâmica. 8 ed. Rio de Janeiro, RJ: LTC, 2009.

HU, Fei; CAO, Xiaojun. **Wireless Sensor Networks**: Principles and Practice. 1 ed. Boca Raton, FL: CRC Press, 2010.

IPSO. **Enabling the Internet of Things**: White Papers. IPSO Alliance (2015). Disponível em: <http://www.ipso-alliance.org/white-papers>. Acesso em: 21 nov. 2015

ITU. **The Internet of Things**. International Telecommunication Union (Nov. 2005). Disponível em: <https://www.itu.int/net/wsis/tunis/newsroom/stats/The-Internet-of-Things-2005.pdf>. Acesso em: 22 nov. 2015.

LI, Qing; YAO, Caroline. **Real-Time Concepts for Embedded Systems**. 1 ed. San Francisco, CA, USA: Elsevier, 2003.

LIBELIUM. **50 Sensor Applications for a Smarter World**. Libelium (2015). Disponível em: <http://www.libelium.com/top_50_iot_sensor_applications_ranking>. Acesso em: 21 nov. 2015.

LOUREIRO, Antonio A. F. et al. **Redes de sensores sem fio**. Simpósio Brasileiro de Redes de Computadores, p. 179 – 226, 2003.

MACHADO, Anaxágora Alves. Poluição sonora como crime ambiental. **Revista Jus Navigandi**, Teresina, ano 9, n. 327, 2004. Disponível em: <http://jus.com.br/artigos/5261>. Acesso em: 10 set. 2015.

MAHALIK, Nitaigour P. **Sensor Networks and Configuration**: Fundamentals, Standards, Platforms, and Applications. 1 ed. Gwangju, República da Coréia do Sul: Springer, 2007.

PASSEMARD, Antony. **The Internet of Things Protocol stack**: from sensors to busines value. 2014. Disponível em: <https://entrepreneurshiptalk.wordpress.com/2014/01/29/the-internet-of-thing-protocol-stack-from-sensors-to-business-value>. Acesso em: 19 nov. 2015.

POTDAR, Vidyasagar; SHARIF, Atif; CHANG, Elizabeth. **Wireless Sensor Networks:** A Survey. IEEE Workshops of International Conference on Advanced Information Networking and Applications Workshops (IEEE WAINA), p. 636-641, mai. 2009.

PROTOCOLS stack. In: Contiki Development at ANRG, University of Southern California. Disponível em: <http://anrg.usc.edu/contiki/index.php/Protocols_stack>. Acesso em: 15 jun. 2016.

RUIZ, Linnyer Beatrys, et al. **Arquitetura para Redes de Sensores Sem Fio**. Minicursos do XXII Simpósio Brasileiro de Redes de Computadores (SBRC'04), Cap. 4, p. 167–218. Gramado, RS, Brasil. ISBN 85-88442-81-7.

SATOR. **Connected Smart Cities**: cidades do futuro no Brasil. Sator (2015). Disponível em: <http://issuu.com/connectedsmartcities/docs/informativo_connected_smart_cities/1>. Acesso em: 01 set. 2015.

SMARTSANTANDER. **SmartSantander**: Future Internet Research & Experimentation. Disponível em: <http://www.smartsantander.eu>. Acesso em: 21 nov. 2015.

SOHRABY, Kazem; MINOLI, Daniel; ZNATI, Taieb. **Wireless Sensor Networks**: Technology, Protocols, and Applications. 1. ed. Hoboken, NJ: John Wiley & Sons, 2007.

STANSFELD, Stephen A.; MATHESON, Mark P. **Noise pollution: non-auditory effects on health**. British Medical Bulletin, 68(5):243-257, 2003.

VALVANO, Jonathan W. **Real-Time Interfacing to ARM Cortex-M Microcontrollers:** Embedded Systems. 4 ed. San Bernardino, CA. 2014.

VERMESAN, Ovidiu; FRIESS, Peter. **Internet of Things Applications**: From Research and Innovation to Market Deployment. 1 ed. River Publishers, 2014.

VIEIRA, Marcos Augusto Menezes. **BEAN: Uma Plataforma Computacional para Redes de Sensores sem Fio**. 2004. Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Minas Gerais, Belo Horizonte, 2004.

WORLD HEALTH ORGANIZATION. **Burden of disease from environmental noise**: quantification of healthy life years lost in Europe. World Health Organization (WHO: 2011). Disponível em: <http://www.euro.who.int/__data/assets/pdf_file/0008/136466/e94888.pdf>. Acesso em: 01 set. 2015.

ZAJARKIEWICCH, Daniel F. B. **Poluição sonora urbana**: principais fontes. 2010. Dissertação (Mestrado em Direito) – Pontifícia Universidade Católica de São Paulo, São Paulo, 2010.

APÊNDICE A – Visão do Projeto no CoIDE