PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Universidade Federal do Rio de Janeiro

JOSÉ RENATO DA SILVA JUNIOR

PRISMA: middleware orientado a recursos para redes de sensores sem fio

DISSERTAÇÃO DE MESTRADO

Rio de Janeiro

2014

Instituto de Matemática

Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

JOSÉ RENATO DA SILVA JUNIOR

PRISMA: middleware orientado a recursos para redes de sensores sem fio

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Instituto de Matemática, Instituto Tércio Pacciti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, como requisito parcial à obtenção do título de Mestre em Informática.

Orientador: Profa. Flávia C. Delicato, Dsc.

Rio de Janeiro
2014

S586 Silva Junior, José Renato da
PRISMA: middleware orientado a recursos para redes de sensores sem fio. / José Renato da Silva Junior.—2014.
128 f.: il.

Dissertação (Mestrado em Informática)—Universidade Federal do Rio de Janeiro, Instituto de Matemática, Instituto Tércio Pacciti de Aplicações e Pesquisas Computacionais, Programa de Pós-Graduação em Informática, Rio de Janeiro, 2014.

Orientador: Flávia Coimbra Delicato

1. Middleware. 2. REST. 3. Comunicação Assíncrona. 4. Orientado a Recursos. 5. Arduino – Teses. I. Delicato, Flávia Coimbra. II. Universidade Federal do Rio de Janeiro, Instituto de Matemática, Instituto Tércio Pacciti de Aplicações e Pesquisas Computacionais, Programa de Pós-Graduação em Informática. III. Título.

CDD.

José Renato da Silva Junior

PRISMA: MIDDLEWARE ORIENTADO A RECURSOS PARA REDES DE SENSORES SEM FIO

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Instituto de Matemática, Instituto Tércio Pacciti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, como requisito parcial à obtenção do título de Mestre em Informática.

Aprovada em 10/12/2014.

_____

Prof.ª Flávia Coimbra Delicato, DSc., UFRJ

_____

Prof.ª Luci Pirmez, DSc., UFRJ

_____

Prof. José Ferreira de Rezende, Dr., UFRJ

Agradecimentos

Agradeço primeiramente a Deus por ter feito com que várias pessoas cruzassem o meu caminho e me ajudassem nessa caminhada. Agradeço especialmente aos colegas de laboratório, especialmente Thomaz e Jesús que me ajudaram mais do que eu esperaria, agradeço também aos outros colegas mesmo os que chegaram perto do meu término nesta jornada. E agradeço também aos meus pais pelo apoio incondicional, em todos os momentos, para que eu conseguisse atingir meus objetivos.

À minha orientadora, Profa. Flávia Delicato, meus agradecimentos especiais por confiar e acreditar no meu trabalho, principalmente pela paciência nas inúmeras correções e ajustes necessários para melhorar esta dissertação. Aos professores Luci e Rezende pelas palavras de incentivo e sugestões na defesa. E também aos colegas do LabNet que me ajudaram tanto durante este caminho, especialmente o Claudio Miceli.

Agradeço também aos meus amigos que compartilharam comigo todos os momentos durante esta jornada, tanto os positivos quantos os negativos. Especialmente Eduardo, Rodrigos, Thiago, Gabrieis, Renan, Jean, Leandros, Alyson, e a todos que possivelmente esqueci-me de citar. Desculpem-me são muitos nomes!

Agradeço a todos os professores por me proporcionar o conhecimento não apenas racional, mas a manifestação do caráter e afetividade da educação no processo de formação profissional, por tanto que se dedicaram a mim, não somente por terem me ensinado, mas por terem me feito aprender.

A todos que direta ou indiretamente fizeram parte da minha formação, o meu muito obrigado.

Resumo

SILVA JUNIOR, José Renato da. **PRISMA:** middleware orientado a recursos para redes de sensores sem fio. 2014. 128 folhas. Dissertação (Mestrado em Informática) – Programa de Pós-Graduação em Informática, Instituto de Matemática, Instituto Tércio Pacciti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2014.

A tecnologia de RSSF encontra-se em constante evolução, atraindo a atenção de diferentes comunidades de pesquisadores e, mais recentemente da indústria. Tal evolução levou à criação de uma ampla gama de aplicações, variando de monitoramento ambiental a detecção de danos em estruturas. Além disso, há também o aumento no número de infraestruturas físicas de RSSF e com isso surge à tendência de compartilhar e integrar dados produzidos por estas redes. Ao mesmo tempo em que podemos observar esta tendência, podemos observar que as aplicações para RSSF deixaram de ter requisitos simples e passaram a exigir mais de suas infraestruturas de sensoriamento, processamento e comunicação. Esta tendência torna impraticável uma abordagem monolítica para projetar RSSFs, já que não é possível conhecer os requisitos de suas potenciais aplicações em fase de pré-implantação. Surge então a necessidade de uma plataforma de software no nível de middleware, fundamentalmente fornecendo as quatro funcionalidades listadas na literatura como essenciais para um middleware de RSSF: (i) abstração de programação; (ii) serviços; (iii) suporte em tempo de execução; e (iv) mecanismos de qualidade de serviço. Diante dessas tendências, este trabalho apresenta o middleware PRISMA, um middleware orientado a recursos para rede de sensores sem fio. O PRISMA provê abstrações de programação através de interfaces REST para disponibilizar as capacidades da rede como recursos acessíveis via web; oferece um conjunto de serviços básicos, incluindo comunicação assíncrona e descoberta de recursos; e oferece suporte em tempo de execução através da criação de aplicações em tempo real e gerenciamento dos nós da rede. O PRISMA é agnóstico à plataforma de RSSF, porém neste trabalho foi instanciado para a plataforma Arduino. Uma avaliação quantitativa com relação às funcionalidades do middleware foi conduzida e com base nos resultados obtidos foi possível observar que, apesar de o Arduino não ser uma plataforma ideal para RSSF, é viável construir um *middleware* nessa plataforma

que atende várias das funcionalidades supracitadas e que possui um desempenho adequado em termos de tempo de resposta percebido pelas aplicações clientes.

Palavras-chave: Middleware; REST; comunicação assíncrona; orientado a recursos; Arduino.

Abstract

SILVA JUNIOR, José Renato da. **PRISMA:** middleware orientado a recursos para redes de sensores sem fio. 2014. 128 folhas. Dissertação (Mestrado em Informática) – Programa de Pós-Graduação em Informática, Instituto de Matemática, Instituto Tércio Pacciti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2014.

The WSN technology is constantly evolving, attracting the attention of researchers from different communities, and more recently from the industry. This constant evolution has led to a wide range of applications, ranging from environmental monitoring to damage detection in civil structures. In addition, there is also an increase in the number of deployed WSN infrastructures and with that comes the tendency to share and integrate data produced by these networks. While we can observe this recent trend, we can also notice that applications for WSN no longer have simple requirements and have demanded more of the underlying sensing, processing and communication infrastructure. This trend makes it impractical to design monolithic WSN applications, since it is not possible to meet the requirements of all potential applications that will execute in the infrastructure, at pre-deployment time. Then the need emerges for a software platform at the middleware level, providing essentially four features listed in the literature as essential for a WSN middleware: (i) programming abstraction to develop and execute applications; (ii) a set of services; (iii) support at runtime; and (iv) mechanisms for quality of service provision. In this context, this work presents PRISMA, a resource-oriented middleware for wireless sensor networks. PRISMA provides programming abstractions through REST interfaces to provide network capabilities as resources accessible via the Web; it offers a set of basic services, including asynchronous communication, and resource discovery; and provides runtime support by creating applications and management of network nodes in real-time. PRISMA is agnostic to the WSN platform, but in this work it has been instantiated for the Arduino platform. A quantitative evaluation with respect to the middleware functionality has been conducted and based on the results obtained we observed that, despite Arduino is not an ideal platform for WSN, it is feasible to middleware in such platform that meets several required

features and presents an acceptable behavior when compared with typical Web applications in terms of response time perceived by the client.

Keywords: middleware; resource-oriented; asynchronous communication; Arduino.

Lista de Figuras

Figura 2.1: Placa Arduino 26
Figura 2.2: Requisição HTTP sobre um recurso 40
Figura 3.1: Componentes lógicos do subsistema Gateway 48
Figura 3.2: Componentes lógicos do subsistema *Cluster Head* 51
Figura 3.3: Componentes lógicos do subsistema Nó Sensor 52
Figura 3.4: Requisição e Resposta - *Gateway* 56
Figura 3.5: Requisição e Resposta – *Cluster Head* 57
Figura 3.6: Requisição e Reposta – Nó Sensor 58
Figura 3.7: Descoberta de recursos – Nó Sensor 60
Figura 3.8: Descoberta de recursos – *Cluster Head* 61
Figura 3.9: Descoberta de recursos - *Gateway* 62
Figura 3.10: Modelo de Comunicação assíncrona – *Gateway* 66
Figura 3.11: Modelo de comunicação assíncrona – *Cluster Head* 67
Figura 3.12: Modelo de comunicação assíncrona – Nó Sensor 68
Figura 3.13: Arquivo de configuração de nova aplicação 71
Figura 3.14: Diagrama de atividades descrevendo a operação do sistema 72
Figura 3.15: Algoritmo de pré-seleção de clusters 75
Figura 3.16: Algoritmo de seleção de nós ativos que executa no subsistema *Cluster Head* 76
Figura 5.1: A) Arduino UNO; B) Arduino MEGA 88
Figura 5.2: Arduino UNO com a placa *XBee Shield* 89
Figura 5.3: Diagrama de implantação do subsistema *Gateway* 92
Figura 5.4: Diagrama de implantação do subsistema *Cluster Head* 92
Figura 5.5: Digrama de implantação do subsistema Nó Sensor 92
Figura 5.6: Principais tecnologias utilizadas no desenvolvimento do PRISMA 93
Figura 5.7: Página de login 94
Figura 5.8: Página para criação de novas aplicações 94
Figura 5.9: Resultados apresentados na interface web 95
Figura 5.10: Função para coleta de voltagem fornecida para o Arduino. 97
Figura 5.11: Consumo de energia do Arduino em diversos testes 98
Figura 5.12: Gráfico de calor do Arduino MEGA 100

Figura 6.1: Diagrama esquemático da topologia de rede utilizado no experimento 103
Figura 6.2: XML que representa a aplicação do cenário 1 108
Figura 6.3: XML que representa a aplicação do cenário 2 108
Figura 6.4: XML que representa a aplicação do cenário 3 109
Figura 6.5: XML que representa a aplicação do cenário 4 110
Figura 6.6: Aplicação baseada em eventos com 3 eventos a serem monitorados 113
Figura 7.1: Formato do pacote de controle.....127
Figura 7.2: Formato do pacote de mensagem recebida 127
Figura 7.3: Formato do pacote de transmissão de mensagem.....128
Figura 7.4: Modelo PRISMA.....129

Lista de Tabelas

Tabela 1: Operações (HTTP verbs) para acesso e/ou manipulação de recursos em REST. 40
Tabela 2: Comparação dos modelos de Arduino 88
Tabela 3: Resultados da avaliação do PRISMA.....111

Lista de Siglas

API	Application Programming Interface
CDR	Common Data Representation
CMOS	Complementary metal-oxide-semiconductor
CPU	Central Processing Unit
DAO	Data Access Object
DDS	Data Distribution Service
DVS	Dynamic Voltage Scaling
DYI	Do It Yourself
FTDI	Future Technology Devices International
GIOP	General Inter-ORB Protocol
GQM	Goal Question Metric
HEED	Hybrid Energy-Efficient Distributed clustering
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IOT	Internet of Things
J2EE	Java2 Platform Enterprise Edition
JSON	JavaScript Object Notation
LEACH	Low Energy Adaptive Clustering Hierarchy
LED	Light Emission Diode
MAC	Media Access Control
MARINE	MiddlewAre for Resource and mIssion oriented sensor NEtworks
MOM	Middleware orientado a mensagem
MufFIN	Middleware For the Internet of thiNgs
OERP	Overlay Event Routing Protocols
OMG	Object Management Group
PEGASIS	Power-Efficient. GAthering in Sensor Information Systems
PWM	Pulse-Width Modulation
QoS	Quality of Service
RAM	Random Access Memory

REST	REpresentational State Transfer
RPC	Remote Procedure Call
RSSF	Rede de Sensores Sem Fio
SGBD	Sistema de Gerenciamento de Banco de Dados
SM	Sleep Mode
SOAP	Simple Object Access Protocol
SP	Sleep Period
SWE	Sensor Web Enablement
TinyDDS	Toward Interoperable Publish/Subscribe Communication between Wireless Sensor Networks and Access Networks
UML	Unified Modeling Language
URI	Uniform Resource Identifier
USB	Universal Serial Bus
XML	eXtensible Markup Language

Sumário

1 Introdução 18
1.1 Delimitação do Escopo 22
1.2 Organização do Trabalho 25
2 Conceitos Básicos 26
2.1 Plataforma Arduino 26
2.2 Redes de Sensores Sem Fio 27
2.2.1 Otimização do Consumo de Energia em RSSF 29
2.3 Qualidade de Serviço (QoS) 33
2.3.1 Qualidade de Serviço em RSSF 33
2.4 *Middleware* para RSSF 35
2.5 REST 38
2.5.1 Princípios REST 39
2.6 Modelo de comunicação assíncrona 41
2.6.1 Modelos de comunicação assíncrona em RSSF 44
3 PRISMA 45
3.1 Arquitetura do PRISMA 46
3.1.1 Componentes do *Middleware* 47
3.1.2 Descrição Comportamental dos Componentes do Middleware 54
3.2 Operação do PRISMA 69
3.3 Controle de Topologia 72
4 Trabalhos Relacionados 77
4.1 TinyDDS 77
4.2 MiSense 79
4.3 MufFIN 81
4.4 Mires 83
4.5 MARINE 85
5 Implementação 87
5.1 Ambiente de implementação 87
5.1.1 Rádio XBee 90
5.2 Implantação 91
5.3 Interfaces gráficas 93
5.4 Lições aprendidas na Implementação do PRISMA 96

6 Avaliação 102
6.1 Abordagem Goal/Question/Metric (GQM) 103
6.2 Avaliações do PRISMA baseadas na abordagem GQM 104
6.2.1 Metodologia de avaliação e cenários 107
6.2.2 Análise dos resultados 111
7 Conclusão e Trabalhos Futuros 116
7.1 Direções futuras 117
Referências 118
Apêndices 126
APÊNDICE A – INFORMAÇÃO ADICIONAL SOBRE O RÁDIO XBEE 126
APÊNDICE B – Figura que exibe a modelagem do banco de dados 129

1 Introdução

Redes de Sensores sem Fio (RSSF) constituem uma nova forma de computação distribuída onde nós sensores são implantados em uma área alvo para coletar informações sobre fenômenos físicos, com pouco ou nenhum impacto sobre o ambiente em que estão implantados, e entregá-las a um ou mais pontos de saída da rede, chamados nós sorvedouros (ou *sinks*), através de comunicação sem fio. Um nó sensor é composto essencialmente por (i) um rádio para transmissão e recepção de dados, conectado a uma antena, (ii) um microcontrolador para controlar a entrada/saída de dados e efetuar o processamento dos sinais recebidos dos sensores através de conversores analógico-digitais, (iii) uma bateria, que fornece a energia necessária para o funcionamento de todo o nó e (iv) as unidades de sensoriamento, capazes de coletar informações sobre fenômenos físicos, como por exemplo: luminosidade, temperatura e umidade, entre outros. Já os nós sorvedouros são, geralmente, computadores com recursos abundantes para onde os dados coletados pela rede são encaminhados para análise e processamento adicionais.

A tecnologia de Redes de Sensores Sem Fio encontra-se em constante evolução, atraindo a atenção de diferentes comunidades de pesquisadores e, mais recentemente, da indústria. Existem atualmente inúmeras plataformas de hardware e software para tais redes. Exemplos de plataformas são os nós sensores fabricados pela Crossbow, atualmente MEMSIC [1](Mica, MicaZ, entre outros), que são suportados pelo sistema operacional TinyOS [2], nós sensores fabricados pela Sun [3](atualmente Oracle) chamados Sun Spots, que utilizam programação em Java e, mais atualmente, os nós sensores da plataforma Arduino [4](Arduino Uno, Arduino Mega, entre outros), que são foco deste trabalho.

Há também uma ampla gama de aplicações para RSSF, variando de monitoramento ambiental a detecção de danos em estruturas civis [5]–[8]. Como o número de infraestruturas físicas de RSSF na mesma área de cobertura tende a aumentar consideravelmente, há uma tendência em direção a se compartilhar e integrar os dados produzidos por tais redes entre diferentes aplicações, além de iniciativas no sentido de incluir dados de monitoramento como parte de aplicações Web, integrados a outros tipos de recursos disponíveis na Internet [9], [10].

Inicialmente, as aplicações para RSSF possuíam requisitos simples, não demandando a necessidade do uso de infraestruturas de software complexas. Além disso, as RSSF eram normalmente projetadas para atender aos requisitos de uma única aplicação alvo, ou seja, o código a ser instalado nos nós sensores era tipicamente monolítico, por estar altamente associado aos requisitos de uma única aplicação, a uma plataforma específica, e a pilha de protocolos disponíveis para tal plataforma. Além disso, desenvolvedores de software estavam atrelados ao uso direto de primitivas de baixo nível providas pelo sistema operacional para a construção de suas aplicações. Junto a isto, os desenvolvedores adotavam uma abordagem de projeto altamente voltada a melhorar a eficiência energética da aplicação, visando estender o tempo de vida dos dispositivos utilizados, dados os recursos limitados dos nós sensores. Tal abordagem gerou uma forte dependência entre a camada de aplicação e os níveis mais baixos de protocolos que precisavam ser utilizados para melhorar a eficiência energética de suas aplicações. Esse nível de dependência entre a camada de aplicação e níveis mais baixos de protocolos não é desejável segundo tendências atuais [11]–[14]. De acordo com estas tendências não é possível projetar a RSSF utilizando uma abordagem monolítica, já que não conhecemos os requisitos das suas potenciais aplicações em fase de pré-implantação. Além disso, esta forte dependência gerava sistemas difíceis de manter, estender e reusar.

A fim de atender as tendências citadas anteriormente, torna-se necessária uma plataforma de software no nível de middleware, fundamentalmente fornecendo abstrações para construção de aplicações e acesso aos dados produzidos pela rede, e oferecendo múltiplos serviços, genéricos ou específicos de domínio. Embora plataformas de middleware sejam amplamente utilizadas em ambientes de sistemas distribuídos tradicionais, o desenvolvimento de middleware no contexto de RSSF é uma área de pesquisa que tem crescido muito na última década. Há reportada na literatura uma série de pesquisas [15]–[18] propondo plataformas de middleware para RSSF, cada qual projetada para atender um conjunto de requisitos distintos. Por exemplo, algumas plataformas focam principalmente na questão de interoperabilidade entre dispositivos heterogêneos que servem diversos domínios de aplicações, enquanto outras tratam de adaptação e ciência de contexto em RSSF, outras ainda abordam questões de descoberta de serviços e de gerenciamento de dispositivos e tratam a escalabilidade em RSSF.

Um middleware pode ser definido como uma camada de software localizada geralmente entre o sistema operacional e as aplicações, no intuito de facilitar o desenvolvimento e a execução de aplicações. Este oculta dos desenvolvedores de aplicações complexidades e heterogeneidades referentes ao hardware e as plataformas de rede subjacentes, as diferenças de protocolos de comunicação, e dependências do sistema operacional, facilitando o gerenciamento de recursos do sistema e aumentando a previsibilidade de execução de aplicações [19].

Um middleware para RSSF é um artefato de software que reside entre a aplicação e a infraestrutura de comunicação, provendo através de interfaces, um conjunto de serviços que podem ser compostos e configurados para facilitar o desenvolvimento e a execução de aplicações de forma mais eficientes para o ambiente distribuído [20]. Além disto, um middleware para RSSF necessita contemplar algumas funcionalidades básicas específicas do ambiente dessas redes. Segundo [19], um middleware para RSSF deve oferecer quatro funcionalidades principais: (i) prover abstrações de programação, (ii) prover serviços, (iii) fornecer suporte em tempo de execução e (iv) fornecer mecanismos de provisão de QoS (*Quality of Service*). As **abstrações de programação** são responsáveis pela definição das interfaces do middleware expostas para o programador da aplicação. Já os serviços do sistema fornecem implementações para alcançar as abstrações; portanto estes serviços incorporam as funcionalidades a serem providas e compõem o núcleo do middleware. Dentre os **serviços** providos por um middleware para RSSF, pode-se identificar um conjunto básico, responsável por atender as necessidades comuns da maioria das aplicações, além de serviços específicos de domínio [21]. Dentre estes serviços básicos destaca-se a entrega das informações coletadas pelos nós sensores para o sorvedouro e do sorvedouro para as aplicações clientes (serviço de comunicação). A comunicação em RSSF pode ser classificada como síncrona ou assíncrona. Para alguns domínios, como por exemplo, aplicações de monitoramento em tempo real (domínio de segurança, com captura de vídeo para detecção de intrusos [5], [6] e aplicações de monitoramento contínuo de estruturas [7], [8]), o modelo síncrono de comunicação é mais adequado devido a grande taxa com que os dados são coletados e transmitidos. Porém, grande parte das aplicações para RSSF são baseadas em eventos (modelo de comunicação baseado em eventos) e, portanto, o esquema tradicional de requisição-resposta (*request-reply*), tipicamente síncrono, não é apropriado para uso

nessas redes. Então, seguindo as tendências citadas no inicio do Capítulo, onde as RSSFs tendem a crescer em cobertura e número de aplicações suportadas para então agregar e disponibilizar estes dados na Web, idealmente é necessário que um middleware proveja suporte para ambos os modelos de comunicação, de modo a atender a uma maior gama de aplicações que utilizem ambos os modelos.

Outro serviço básico implementado por plataformas de middleware para RSSF é o serviço de controle de topologia. Um dos maiores desafios de RSSFs ainda é a limitação energética dos nós sensores. Em geral, o mesmo nó é responsável por produzir dados e por encaminhar mensagens suas e de seus vizinhos e, gerando um grande impacto em seu consumo de energia, já que o rádio é o componente responsável pelo maior gasto de energia na maioria das RSSF [22]. Além disso, o tráfego de mensagens em geral não é balanceado na RSSF já que os nós mais próximos do sorvedouro são esgotados mais rapidamente, por servirem de rota para todos os nós da rede. Para utilizar os recursos escassos dos dispositivos de uma maneira mais eficiente e estender o tempo de vida global da RSSF, mecanismos de controle de topologia [23]–[26] são utilizados. A ideia do controle de topologia é limitar o número de vizinhos de certo nó ou limitar o número de mensagens trafegadas na rede, seja controlando a potência de transmissão, introduzindo hierarquias na rede ou, mesmo, desligando certos nós por um período determinado de tempo, objetivando, com isso, a redução do consumo de energia enquanto preserva-se a cobertura e conectividade da rede.

Outro serviço básico comumente provido por um middleware para RSSF é o serviço de descoberta de recursos. Neste serviço, os nós sensores anunciam suas capacidades ao *middleware* para que este tome ciência dos recursos que devem ser gerenciados e oferecidos para as aplicações.

No tocante, ao **suporte em tempo de execução e aos mecanismos de QoS**, por ambos atuarem no sistema e poderem ser providos de diversas maneiras em diferentes níveis da pilha de protocolo de RSSF não é raro vermos estas duas funcionalidades se sobrepondo. Para abstrair esta complexidade das aplicações é recomendado que estas funcionalidades sejam manipuladas pelo *middleware*. Para podermos ter uma melhor visão do escopo de atuação destas funcionalidades, é importante ressaltar que em ambas o escopo pode ser a RSSF (infraestrutura) ou as aplicações. No caso da funcionalidade de

suporte em tempo de execução, esta atua como uma extensão do sistema operacional embutido dos nós, provendo funcionalidades como, por exemplo: escalonamento de tarefas, comunicação entre processos, gerenciamento de memória, e controle de energia em termos de alteração de voltagem de funcionamento, ativação/desativação de componentes e (re)configuração de componentes. A necessidade de suporte em tempo de execução em RSSF se origina do fato de que nem sempre o hardware e firmware do nó sensor possuem suporte suficiente para a implementação de determinados serviços requeridos para a operação e otimização da rede. Já os **mecanismos de QoS** são utilizados para atender restrições de QoS impostas pelo sistema, tanto do ponto de vista da rede como da aplicação. Exemplos típicos são restrições impostas pelos seguintes parâmetros de QoS: tempo de vida da rede, cobertura de sensoriamento, acurácia, latência, largura de banda, dentre outros. Alguns mecanismos de provisão de QoS reportados na literatura são: mecanismos que alteram a potência de transmissão dos nós (a energia utilizada durante a transmissão é influenciada pela potência de transmissão), mecanismos que alteram o modo de operação dos nós (entre “dormindo” e ativos), mecanismos que alteram o papel dos nós entre líderes de cluster e nós sensores/roteadores, mecanismos de escalonamento de tarefas, entre outros. Por existirem sobreposições entre o suporte em tempo de execução e os mecanismos de QoS, na literatura podemos encontrar mecanismos de QoS sendo providos como serviços, como, por exemplo, a alteração do papel dos nós sensores e a alteração do modo de operação dos nós sensores; e podemos também encontra-los como sendo parte do suporte em tempo de execução, como, por exemplo, os mecanismos de escalonamento de tarefas e a alteração da potência de transmissão dos nós sensores.

Apesar de já haver várias propostas de middleware para RSSF, no entanto poucos dos trabalhos existentes abordam mais de um dos quatro requisitos mencionados ao mesmo tempo.

1.1 Delimitação do Escopo

Nesta seção são discutidas as questões que são abordadas neste trabalho e as que estão fora do escopo deste trabalho.

Dentro do contexto supracitado, o objetivo geral deste trabalho é construir um middleware orientado a recursos para redes de sensores sem fio. Como citado

anteriormente, um *middleware* deve prover quatro funcionalidades principais. Dentre estas, o PRISMA visa atender as seguintes funcionalidades:

- abstrações de programação - o PRISMA provê interfaces REST [27] para acessar os recursos da RSSF e para criar novas aplicações.
- serviços - além do serviço básico de comunicação síncrona, o PRISMA provê o serviço de comunicação assíncrona, descrito na Seção 2.6.
- suporte em tempo de execução - Através das interfaces REST é possível criar e implantar uma nova aplicação na RSSF em tempo de execução.

Com relação à quarta funcionalidade citada, a saber, a provisão de mecanismos de QoS, a mesma não se encontra no escopo desta proposta e será abordada como um trabalho futuro.

O mecanismo de QoS planejado para ser oferecido pelo PRISMA será implementado sob a forma de um serviço de controle de topologia, responsável por controlar a atividade dos nós sensores, mais especificamente alterando seu ciclo de trabalho (ativando e desativando certo conjunto de nós em dado tempo). Esta abordagem foi incentivada pelas tendências atuais segundo as quais as RSSFs tendem a crescer em cobertura e prover uma grande redundância nos dados (que pode ser traduzido em uma grande densidade de nós). O mecanismo de QoS baseado em controle de topologia possibilita um consumo homogêneo da energia na rede através a ativação de somente um conjunto de nós que seja suficiente para atender aos requisitos das aplicações que estão executando na rede enquanto os outros nós poupam energia e prolongam o tempo de vida útil do sistema. No restante deste documento iremos considerar esta funcionalidade como tendo sido planejada e, portanto, já projetada e incluída na arquitetura lógica do PRISMA, mas por restrição de tempo e dificuldades tecnológicas, ainda não implementada na versão atual do protótipo construído. Também devido à restrição de tempo o PRISMA em sua atual implementação foi concebido para atuar com somente um nó sorvedouro para coleta de informações. Ou seja, apenas uma RSSF é tratada pelo *middleware*, não havendo a necessidade de gerência de múltiplas redes. Porém, o *middleware* ainda é relevante no contexto de redes de sensores compartilhadas já que em uma mesma infraestrutura executam N aplicações simultaneamente. No PRISMA, aplicações são definidas por um conjunto de tarefas de

sensoriamento que são solicitadas por um cliente com a finalidade de coletar dados do ambiente em que os dispositivos estão inseridos.

A decisão de utilizar o estilo arquitetural REST como abstração de programação foi motivada pela necessidade crescente de interoperabilidade e integração de RSSF entre si e com outros sistemas, como preconizado por paradigmas recentes como o de redes de sensores compartilhadas [9], [10] e da Internet das Coisas [do inglês *Internet of Things*, IoT - [28]]. Além disso, uma das principais vantagens do estilo arquitetural REST é ele só depender do protocolo HTTP para se comunicar. Grande parte das plataformas e dispositivos possuem acesso a este protocolo, tornando o REST uma melhor opção em termos de interoperabilidade. Da mesma forma, para aplicações de IoT que requerem sensoriamento de tipos diversos e por áreas extensas, muitas vezes será necessário extrair dados de múltiplas RSSFs para atender a uma única aplicação; mais ainda, tais dados poderão ser integrados com outros recursos disponíveis na Web ou mesmo com sofisticados sistemas de informação. Portanto, prover uma interface de acesso uniforme aos dados de sensores e mecanismos para especificar aplicações de sensoriamento de forma ágil, torna-se uma necessidade. Em uma abordagem baseada em REST as capacidades de sensoriamento da RSSF podem ser acessadas como recursos, da mesma forma que outros recursos disponíveis na Web. Os dados produzidos são providos para as aplicações através de uma camada de abstração que esconde dos usuários, ou aplicações clientes, os detalhes de hardware dos dispositivos e/ou suas plataformas de *software*. Através desta abstração de recursos, é possível integrar os dados produzidos por diferentes sensores, redes, ou sistemas Web, desde que todos sigam a mesma abstração e façam uso da mesma interface uniforme.

Para tirar o maior proveito possível da camada de abstração oferecida e do modelo de comunicação assíncrona utilizado neste trabalho, interfaces REST são usadas para criar as aplicações em tempo real. Uma aplicação é descrita (em forma de requisitos funcionais e não funcionais) em um arquivo XML e enviada para o *middleware* através de uma interface REST. As aplicações no PRISMA são configuradas nos nós sensores através da (re)configuração de parâmetros pré-definidos nestes nós. Então, depois de configurados, os nós sensores começarão a coletar segundo os requisitos das aplicações atualmente em execução. Estes dados coletados serão então encaminhados via serviços de comunicação do *middleware* para os clientes que submeteram suas aplicações.

Apesar de sua arquitetura lógica ser agnóstica em relação às plataformas de RSSF subjacentes, o projeto físico e a implementação do PRISMA foram customizados para a plataforma Arduino. A única restrição que deve ser atendida por uma plataforma de RSSF para ser utilizada juntamente com o PRISMA é ter a possibilidade da alteração dos parâmetros de configuração de seu rádio em tempo de execução. A plataforma Arduino foi selecionada por ser relativamente nova, lançada em 2005, possuir hardware livre e atender a esta restrição. Por ser uma plataforma recente, ainda foi pouco explorada pela comunidade acadêmica, principalmente na área de *middleware*, onde não há na literatura pesquisada nenhum *middleware* para RSSF implementado até o momento. Além disto, a plataforma possui uma linguagem de alto nível que possibilita um desenvolvimento de aplicações rápido e de mais fácil compreensão.

1.2 Organização do Trabalho

Este trabalho encontra-se organizado da seguinte maneira: o Capítulo 2 apresenta os conceitos básicos dos temas relacionados com o presente trabalho. No Capítulo 3 é apresentada a arquitetura do PRISMA, e no Capítulo 3.3 são apresentados trabalhos relacionados. No Capítulo 5 são apresentados os detalhes da implementação e as lições aprendidas durante o desenvolvimento do trabalho. No Capítulo 6 é apresentada a avaliação conduzida e no Capítulo 7 são apresentados à conclusão e os trabalhos futuros.

2 Conceitos Básicos

Este Capítulo discorre sobre os conceitos básicos pertinentes aos temas apresentados neste trabalho. A Seção 2.1 apresenta a plataforma Arduino. A Seção 2.2 aborda brevemente os conceitos de redes de sensores sem fio, otimização energética e qualidade de serviço nessas redes. A Seção 2.4 aborda conceitos de plataformas de middleware específicos para redes de sensores sem fio. A Seção 2.5 introduz o estilo arquitetural REST. A Seção 2.6 aborda o modelo de comunicação assíncrona.

2.1 Plataforma Arduino

A plataforma de prototipagem rápida Arduino [4], utilizada neste trabalho, é baseada em *hardware* aberto, ou seja, é possível desenvolver produtos utilizando-a como base sem o pagamento de *royalties*, permitindo assim a customização e adaptação da rede de sensores de acordo com a necessidade da pesquisa. O módulo Arduino é composto por um microcontrolador ATMega 328, um chip de comunicação USB *Future Technology Devices International* (FTDI) para interligação com um microcomputador e reguladores de voltagem que permitem a utilização de fontes de energia entre 5 e 12 volts, como pode ser visto na Figura 2.1.

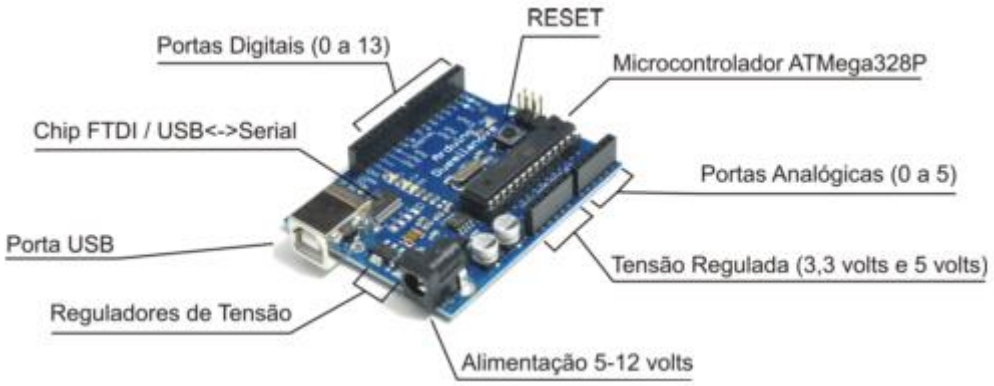

Figura 2.1: Placa Arduino

O Arduino possui 14 portas digitais programáveis, enumeradas de 0 a 13. As portas 0 e 1 são utilizadas para comunicação serial, como por exemplo, com um rádio de comunicação. As portas 2 e 3, além da função normal de Entrada e Saída (E/S) podem ser utilizadas para disparar eventos de interrupção no microcontrolador. Por exemplo, caso um sensor de presença esteja conectado a esta porta, quando ocorrer um evento detectado por

esse sensor um tratamento especial pode ser dado, como, por exemplo, enviar uma mensagem de emergência ou tirar o microcontrolador do estado de dormência.

As outras portas podem ser utilizadas para sensores que retornem valores booleanos, (como verdadeiro ou falso), para ativar e desativar relés/transistores a fim de controlar dispositivos de maior potência ou tensão, além de ligar e desligar lâmpadas *Light Emission Diode* (LED - Diodo Emissor de Luz). As portas digitais 3, 5, 6, 9, 10 e 11 podem ser utilizadas como portas *Pulse Width Modulation* (PWM - Modulação por Largura de Pulso).

Além das portas digitais, o Arduino possui 6 portas analógicas, cada porta com um conversor analógico-digital de 10 bits. Estes conversores funcionam medindo a tensão existente em cada porta, fornecendo um valor entre 0, sem sinal, e 1023, equivalente a 3,3 volts ou 5 volts, de acordo com o nível de tensão utilizado. Podem ser acopladas a estas portas sensores como luminosidade, umidade, temperatura, gás e sísmico/acelerômetro. Outros componentes importantes da placa Arduino são o botão de reset, para a reinicialização do módulo, e os conectores com tensão regulada em 3,3 volts e 5 volts, para utilização em sensores e/ou placas acopladas à placa principal do Arduino.

A plataforma oferece o conceito de *shields*, denotando placas que podem ser adicionadas ao Arduino para aumentar sua funcionalidade. Existem *shields* para conexão do Arduino com módulos *Bluetooth*, *Ethernet*, entre outros. A placa utilizada neste trabalho, denominada *XBee Shield*, permite a interligação do Arduino ao rádio *XBee*. O fabricante do Arduino também fornece um *software* para o desenvolvimento de aplicações, baseadas na linguagem C, permitindo a programação em alto nível.

2.2 Redes de Sensores Sem Fio

Redes de Sensores sem Fio (RSSF) constituem uma nova forma de computação distribuída onde nós sensores são implantados no ambiente para coletar informações sobre fenômenos físicos, com pouco ou nenhum impacto sobre o ambiente em que está implantada, e entregá-las a um ou mais pontos de saída da rede, chamados nós sorvedouros (ou *sink*), através de comunicação sem fio. Estes nós sensores transmitem dados coletados por unidades de sensoriamento, permitindo sensoriar os ambientes nos quais tais dispositivos estão inseridos. As RSSFs diferem de redes tradicionais *ad hoc* e sem fio por diversos fatores, como por exemplo: energia limitada dos dispositivos, capacidade de auto-organização e autoconfiguração (adaptação), operação de forma colaborativa, comunicação

em múltiplos saltos (ou seja, nós sensores encaminham as informações para outros nós sensores até que essa informação chegue a algum ponto de saída da rede), capacidade de processamento *in-network* [29], uso de endereçamento centrado em dados [30], dentre outros.

Um nó sensor é composto essencialmente por (i) um rádio de comunicação para transmissão e recepção de dados, conectado a uma antena, (ii) um microcontrolador para controlar a entrada/saída de dados e efetuar o processamento dos sinais recebidos dos sensores através de conversores analógico-digitais, (iii) uma bateria, que fornece a energia necessária para o funcionamento de todo o nó e (iv) as unidades de sensoriamento, sensores capazes de coletar informações sobre fenômenos físicos, como por exemplo: luminosidade, temperatura e umidade, entre outros. Uma unidade de sensoriamento é capaz de realizar medições de variáveis físicas, tais como vibrações, mudança de pressão, temperatura e velocidade, convertendo-as para sinais digitais. Os dados medidos pelos sensores e transformados em sinais digitais são traduzidos em informações a respeito de um fenômeno em particular, de mais alto nível, e de interesse da aplicação. Um nó de uma RSSF pode também possuir dispositivos atuadores, como por exemplo, relés, lâmpadas LED ou motores, podendo ativá-los e desativá-los de acordo com os dados recebidos pelo rádio. Nesses casos, em geral passa-se a denomina tal infraestrutura como rede de sensores e atuadores sem fio [31].

Quando um dado é coletado pelo sensor, este passa pelo conversor analógico digital e é preparado para o envio para o rádio. Os rádios dos nós possuem buffers para envio e para recepção de dados, permitindo controlar o fluxo de transmissão e recepção. O rádio converte os sinais recebidos pela antena Rádio Frequência em *bytes* para serem transmitidos de forma serial. Este dado irá então trafegar pela rede de para então chegar a um ponto de saída da rede, chamados sorvedouros. Em geral os dados trafegam por diversos sensores através de múltiplos saltos até atingirem seu destino final. Para esta comunicação ser efetiva, ela deve ser governada por um algoritmo de roteamento que irá definir o caminho a ser percorrido pela mensagem para que atinja seu destino da forma mais eficiente de acordo com os requisitos impostos, seja com o intuito de reduzir o atraso, fazendo com que a mensagem tenha o menor número de saltos possível ou com o intuito de economizar energia, fazendo com que a mensagem trafegue por nós que possuem uma maior energia

residual para finalmente atingir o nó sorvedouro, o ponto de saída da rede. Um sorvedouro é, geralmente, um computador com recursos abundantes (muitas vezes considerados ilimitados) para onde os dados coletados pela rede são encaminhados para análise e processamento adicional. O sorvedouro pode agir como gateway ou ponte entre a RSSF e outras redes ou sistemas.

Em geral, o componente que consome mais energia em uma RSSF é o rádio de comunicação. Transmitir um bit de informação, em termos de consumo de energia, é equivalente ao processamento de 3000 instruções [22], [32]. Possíveis soluções para economia de energia em RSSFs são descritas na Seção 2.2.1.

Na Seção a seguir são discutidos o conceito de otimização de consumo de energia em RSSF e algumas técnicas utilizadas nesse contexto. Também será apresentado o conceito de qualidade de serviço no contexto de RSSF.

2.2.1 Otimização do Consumo de Energia em RSSF

Nesta seção é discutido o consumo de energia em RSSF e são apresentadas algumas técnicas para a otimização desse consumo.

Redes de sensores sem fio são capazes de monitorar uma grande variedade de fenômenos físicos e podem ser utilizadas por uma ampla gama de aplicações. Na maioria das aplicações, é essencial que a rede tenha um tempo de vida operacional o mais longo possível. Por exemplo, aplicações de monitoramento ambiental [33], [34] ou de monitoramento de estruturas civis [8], [35] requerem operação contínua da e de vários meses ou anos. No entanto, o tempo de vida da rede é severamente limitado pela capacidade de bateria dos seus dispositivos, dado o grande número de nós e visto que a rede é frequentemente instalada em áreas de difícil acesso ou remotas, tornando a substituição dos sensores que tenham suas baterias esgotadas inviável ou extremamente custosa. Uma forma possível para minimizar tal problema intrínseco as RSSF é utilizar estratégias para captação de energia a partir do ambiente [36]. No entanto, a energia obtida a partir de fontes naturais é instável e pouco confiável para o pleno funcionamento da RSSF. Portanto, é fundamental que RSSFs sejam cientes de sua própria energia e possam lidar com o seu consumo de forma inteligente, a fim de maximizar o seu tempo de vida operacional. Para estender a vida útil de uma RSSF, a eficiência energética deve ser abordada em todos os

níveis da rede, desde componentes individuais do *hardware* do nó sensor até protocolos de eficiência energética que regem o funcionamento da rede como um todo.

O tempo de vida de uma RSSF pode ser medido pelo tempo decorrido antes de todos os nós (ou uma parte significativa destes) terem suas baterias drenadas ou pelo não cumprimento dos requisitos de conectividade, de cobertura ou qualquer parâmetro específico da aplicação que afete diretamente a utilidade da rede. Técnicas atuais para melhorar a eficiência energética incluem projetos de *hardware* de baixa potência [37], [38], que incidem sobre o consumo de energia nos níveis de circuito e de arquitetura de um único nó [39] e estratégias e protocolos energicamente eficientes [40]–[42], que atuam em diferentes camadas da pilha de protocolos da RSSF, como roteamento, escalonamento de tarefas, controle de acesso ao meio, entre outros. No nível do nó, uma vez que uma grande quantidade de energia é consumida pelos seus componentes (CPU, rádio, etc.), mesmo que eles estejam ociosos, sistemas de gestão de energia podem ser utilizados para desligar componentes que não sejam temporariamente necessários.

Segundo [43], o subsistema de comunicação é a origem do maior gasto de energia em um nó sensor, consumindo muito mais energia que o subsistema de processamento que é a segunda maior fonte do consumo de energia. Portanto, sempre que possível, a comunicação deve ser substituída pelo processamento dos dados. A maior fonte de consumo de energia do subsistema de comunicação é o rádio, consumindo energia enquanto envia, recebe ou está ocioso. Então, sempre que possível o rádio deve “dormir”, gastando menos energia do que no estado ocioso (*idle*), ou deve ser totalmente desligado, onde a economia energética é maior. Em terceiro lugar em termos de gasto de energia está o subsistema de sensoriamento, onde dependendo da aplicação este subsistema pode consumir uma quantidade considerável de energia (podendo em alguns casos ser até maior que o subsistema de comunicação [44]).

Para efetivamente reduzir o consumo de energia da RSSF e prolongar seu tempo de vida, algumas estratégias de otimização energética devem ser aplicadas, muitas vezes simultaneamente. Estas estratégias atuam em diferentes níveis: (i) no nível do nó, chamado intra-nó; (ii) no nível que representa o *link* de comunicação entre nós vizinhos (muitas vezes presentes a um salto de distancia), chamado inter-nó e (iii) no nível de toda a rede. É

importante mencionar que os níveis intra e inter-nó possuem uma relação de compromisso que deve ser explorada para fins de economia de energia.

A maioria das estratégias intra-nó são implementadas no sistema operacional do nó, embora algumas delas sejam implementadas na camada MAC ou como algoritmos no topo da camada MAC. Foram identificadas quatro principais diferentes técnicas que permitem a implementação das estratégias no nível intra-nó descritas em seguida.

- ***duty cycling*** - técnica em que os nós alternam entre estados ativos e "dormindo" para economizar energia (pode ser aplicado no subsistema de sensoriamento e no rádio)
- **sensoriamento adaptativo** - técnica em que a taxa de sensoriamento é adaptada dinamicamente para evitar coletar dados que podem não ser necessários
- **escala de tensão dinâmica (*Dynamic Voltage Scaling* - DVS)** - técnica que visa ajustar a tensão e a frequência de funcionamento da CPU com base na carga computacional de seu nó
- **escalonamento de tarefas do sistema operacional** - técnica responsável por escalonar um conjunto de tarefas a serem executadas no sistema e garantir que as restrições de tempo de cada tarefa sejam cumpridas.

As estratégias inter-nós atuam no enlace de comunicação entre nós vizinhos e são baseadas em três técnicas principais descritas em seguida.

- **adaptação da potência de transmissão do rádio** - a potência do rádio é adaptada para transmitir na potência necessária para atingir seus vizinhos e evitar gasto de energia por utilizar potência excessiva
- **escala de modulação dinâmica** - esta técnica consiste em adaptar dinamicamente o nível de modulação do rádio para corresponder à carga de tráfego instantânea, como parte da gestão de energia do rádio
- **otimizações na camada de enlace** - estratégias que incluem o gerenciamento da topologia da rede e estratégias que atuam no *link* de comunicação entre nós vizinhos (como técnicas de detecção e correção de erros).

Já as estratégias que são aplicadas na rede como um todo são implementadas como softwares que atuam em diferentes camadas da pilha de protocolos. Algumas características dessas estratégias descritas em seguida.

- **camada de aplicação** - Inclui algoritmos que requerem colaboração inter-nós e exploram estratégias de gerenciamento de energia intra-nó, portanto apresentando um comportamento *crosslayer*. Existem principalmente dois tipos de algoritmos nesta camada: (i) algoritmos de alocação de tarefas, cujo objetivo é fragmentar a aplicação em tarefas menores, destinando-as a nós sensores diferentes tentando paralelizar a sua execução de forma eficiente energeticamente (minimizando o custo de comunicação); e (ii) algoritmos de fusão/agregação de dados, que exploram a redundância dos dados coletados pelos nós sensores, a fim de minimizar a transmissão de dados, assim trocando custos de comunicação por processamento
- **camada de transporte -** provê mecanismos para (i) mitigar o congestionamento que surge da variação de trafego de mensagens da rede, (ii) recuperar da perda de pacotes devido a congestionamento e transbordamento de filas, (iii) prover alocação de banda justa, e (iv) ordenar entrega de pacotes no caso de fragmentação.
- **camada de rede** - estratégias para eficiência energética no nível da camada de rede são basicamente implementadas como protocolos de roteamento, como parte do algoritmo de roteamento ou como restrições a serem utilizadas nas estratégias de roteamento
- **camada de enlace** - esta camada é responsável por multiplexar *streams* de dados, fazer controle de acesso ao meio e controle de erros, garantindo conexões confiáveis ponto-a-ponto e ponto-a-multiponto, O gerenciamento de energia nesta camada pode ser alcançado principalmente através das seguintes formas listadas a seguir:
    - **protocolos de controle de topologia** - controle de topologia pode ser definido como uma técnica que utiliza qualquer parâmetro controlável da rede para gerar e manter uma topologia lógica com o intuito de reduzir o consumo de energia e alcançar uma propriedade desejada para a rede como um todo, já os possíveis parâmetros controláveis que podem ser modificados para se obter a topologia desejada são: potência de transmissão([45], [46]), modo de operação do nó (ativo, “dormindo” e desligado, [47], [48]) e papel dos nós[49].
    - **protocolos MAC energeticamente eficientes** - estes protocolos especificam como nós compartilham o canal de comunicação e

controlam diretamente a atividade do rádio do nó sensor, podendo evitar gastos de energia no nível MAC, como por exemplo, por colisão de pacotes, por receber pacotes que não são de interesse do receptor, ouvir continuamente o canal enquanto está ocioso, entre outros.

2.3 Qualidade de Serviço (QoS)

O termo qualidade de serviço (QoS) pode ser interpretado de uma forma diferente por diferentes comunidades técnicas/acadêmicas. Por exemplo, na comunidade de aplicações, QoS geralmente se refere à qualidade percebida pelo usuário final ou pela aplicação. Já na comunidade de redes, QoS é aceito como a medida da qualidade do serviço que a rede oferece às aplicações ou usuários; portanto nesta perspectiva o objetivo é prover o serviço de QoS enquanto maximiza o uso dos recursos da rede. Esta última é a perspectiva que discutiremos nesta Seção, em particular, iremos focar nos desafios de suporte à Qualidade de Serviços em RSSF.

2.3.1 Qualidade de Serviço em RSSF

Fornecer suporte a QoS em RSSF ainda é uma questão em aberto devido à ampla gama de possíveis aplicações, cada uma destas com possíveis requisitos de QoS diferentes [50]. Os mecanismos de QoS em geral cruzam múltiplas camadas e componentes e estão incorporados em vários serviços funcionais de *middleware*. Por exemplo, o serviço de gerenciamento de dados exige alta acurácia e confiabilidade.

A provisão de QoS em RSSFs pode ser implementada como um serviço oferecido por um *middleware*, já que o *middleware* atua como uma camada intermediária entre as aplicações e a infraestrutura de comunicação. Dessa forma, o suporte a QoS pode traduzir e controlar as métricas de QoS entre o nível da aplicação e o nível da rede [51]. Devido às características diferenciadas das RSSFs em [50], [52] e [53] são apresentados alguns desafios de suporte à QoS em RSSF que são descritos em seguida.

- **limitações severas de recursos de energia, largura de banda, memória, tamanho de *buffer*, capacidade de processamento, e poder limitado de transmissão** - dentre essas limitações, a energia, fornecida aos nós sensores por baterias, é um dos aspectos mais cruciais, uma vez que pode ser inviável substituir ou recarregar a bateria dos nós sensores e, como consequência, essa limitação impõe uma exigência que é requisito de

qualquer mecanismo de QoS em RSSFs, a simplicidade, já os algoritmos que fazem uso intensivo de computação, protocolos caros de sinalização, ou que exigem manutenção de estados da rede não são praticáveis nas RSSF

- **tráfego desbalanceado** - na maioria das aplicações de RSSFs, o tráfego flui principalmente de um grande número de nós sensores (fontes de dados) para um subconjunto pequeno de nós sorvedouros. Os mecanismos de QoS devem ser projetados para operar nessas condições de desbalanceamento e assimetria da comunicação
- **redundância de dados** - as RSSFs são caracterizadas pela alta redundância de dados gerados a qual, apesar de muitas vezes se explorada em mecanismos de suporte à segurança e tolerância a falhas, gera um maior consumo de energia. A fusão de dados ou a agregação de dados é uma solução utilizada para redução da redundância sem afetar a robustez da rede, porém este mecanismo introduz um aumento na latência e uma maior complexidade ao projeto de QoS em RSSFs
- **dinamismo da rede** - a dinamicidade da rede pode surgir como decorrência de falhas em nós sensores, falhas da conexão sem fio, mobilidade de nós, e de transições de estado dos nós devido ao uso de esquemas eficientes de energia, que podem desligar/ligar nós em tempos diferentes, ou de reconfigurações da rede, já uma RSSF altamente dinâmica aumenta muito a complexidade de uma solução de QoS
- **balanceamento do uso de energia** - a fim de obter longevidade de RSSF, a carga de energia deve ser distribuída uniformemente entre todos os nós sensores, de modo que a energia de um nó sensor ou de um conjunto pequeno de nós não seja esgotada rapidamente
- **escalabilidade** - em uma RSSF genérica, tipicamente centenas ou milhares de nós sensores podem ser distribuídos na área alvo, gerando uma alta densidade de nós. Em consequência, mecanismos de QoS projetados para RSSFs devem escalar para um grande número de nós sensores, isto é, os mecanismos de provisão de QoS não devem degradar rapidamente quando o número dos nós aumenta
- **múltiplos sorvedouros** - principalmente considerando as tendências atuais de RSSF compartilhadas, heterogêneas e de larga escala, será comum à existência de muitos nós sorvedouros, os quais impõem exigências adicionais na rede: por exemplo, um sorvedouro pode pedir aos nós sensores situados ao noroeste da RSSF para enviar

informações de temperatura a cada 10 minutos, enquanto outro nó sorvedouro pode estar interessado em informações de pressão alta na área sudoeste, as RSSFs devem ser capazes de atender diferentes níveis de QoS associados a diferentes sorvedouros

- **múltiplos tipos de tráfego** - algumas aplicações podem requerer diferentes tipos de informação que, consequentemente, podem demandar diferentes tipos de sensores, com diferentes taxas de transmissão e diferentes modelos de comunicação (temperatura, pressão, umidade, campo magnético etc., podem gerar tráfego com características distintas entre si), através desta multiplicidade de tipos de tráfego são gerados desafios no projeto de mecanismos de QoS
- **importância do pacote de dados** - o conteúdo de um pacote de dados que trafega pela RSSF deve refletir a importância do fenômeno físico capturado pelo nó sensor para que seja possível priorizar a sua entrega, os mecanismos de QoS devem ser capazes de identificar essas prioridades e de ajustar o sistema para atender a aplicação.

2.4 *Middleware* para RSSF

O principal objetivo de um *middleware* é possibilitar a comunicação entre componentes distribuídos, escondendo das aplicações a complexidade do ambiente de rede subjacente e livrando-as da manipulação explícita de protocolos e serviços de infraestrutura.

Em [54] é apresentado um modelo de referência que classifica os sistemas de *middleware* entre fixos e *ad hoc* ou móveis. Essa classificação leva em consideração três aspectos, o tipo de carga computacional, o paradigma de comunicação e a representação de contexto.

Middleware para sistemas distribuídos tradicionais ou fixos possuem algumas limitações que faz o uso deles impraticável nas RSSFs. Esses sistemas demandam recursos computacionais (carga computacional pesada), escondem informações de contexto das aplicações tanto quanto possível (transparência), e suportam comunicação síncrona entre componentes, com exceção do *middleware* orientado a mensagem (MOM). A comunicação síncrona não é adequada para ambientes com muitas desconexões. Outro problema é a carga computacional das plataformas de middleware tradicionais, para os poucos recursos dos dispositivos de uma RSSF. A transparência apresentada nem sempre é adequada para as aplicações nas RSSFs, que necessitam de alguma informação sobre o contexto de execução para uma melhor adaptabilidade [54].

As RSSFs são uma categoria de redes sem fio *ad hoc*, e devem ser projetadas obedecendo a alguns dos requisitos de middleware para redes ad-hoc ou móveis. Os principais requisitos incluem carga computacional leve para suportar os dispositivos, prover comunicação assíncrona para contornar os problemas causados pelas frequentes desconexões e tornar as aplicações cientes de contexto. Dessa forma, novos tipos de *middleware* têm sido projetados e desenvolvidos para atender diversos tipos de aplicações em RSSFs [54].

Segundo [55], o principal propósito de *middleware* para RSSF é suportar o desenvolvimento, manutenção, distribuição, e execução de aplicações de sensoriamento. Isso inclui mecanismos para formulações complexas de tarefas de sensoriamento de alto nível, a comunicação dessas tarefas com a RSSF, coordenação de nós sensores para distribuição das tarefas, agregação e/ou fusão de dados para unir as leituras dos sensores em um resultado de alto nível e reportar os resultados das tarefas de volta para o emissor.

Em [19] são descritas três formas que podem ajudar os desenvolvedores de aplicações a utilizar um *middleware* para RSSF. Primeiro, o *middleware* pode fornecer abstrações de sistemas apropriados, de forma que o programador da aplicação pode focar na lógica da aplicação sem se preocupar sobre detalhes de implementação de baixo nível. Segundo, o *middleware* pode prover o reuso de código de serviços tais como atualização, serviços de dados, assim como filtragem de dados, dessa forma programadores de aplicações podem distribuir e executar aplicações sem se preocupar com complexas e tediosas funções. Terceiro, o *middleware* pode ajudar os programadores no gerenciamento da infraestrutura da rede e na adaptação, fornecendo recursos eficientes de serviços, como gerenciamento de energia, além de suportar integração de sistemas, monitoramento e segurança.

Em [56] foram propostos princípios a serem adotados em um projeto de *middleware* para RSSFs.

- o *middleware* deve fornecer mecanismos centrados em dados para o processamento e a consulta de dados no interior da rede
- algoritmos localizados devem ser usados para alcançar um desejável objetivo enquanto fornecem boa escalabilidade e robustez ao sistema

- plataformas de *Middleware* tradicionais são projetadas para suportar uma ampla variedade de aplicações na rede, mas devido aos recursos limitados disponíveis, *middleware* para RSSFs não podem ser generalizados dessa forma
- a disponibilidade de recursos dos nós sensores é baixa, o *middleware* então deve ser leve em termos de requisitos de comunicação e computação
- devido aos recursos limitados, é muito provável que os requisitos de desempenho de todas as aplicações em execução não possam ser simultaneamente satisfeitos, portanto, é necessário que o *middleware* negocie de forma inteligente a QoS entre várias aplicações.

O projeto e desenvolvimento de *middleware* impõem vários desafios devido as característica das RSSFs, por exemplo, restrições de recursos, disponibilidade e diversidade de hardware de sensor. Em [57] foi elaborada uma descrição de vários desafios associados com *middleware* para RSSFs.

- **suporte na abstração** - as RSSFs consistem de um grande número de sensores heterogêneos, os sensores são desenvolvidos por empresas diferentes e talvez tenham diferentes plataformas de *hardware*, então esconder as plataformas de *hardware* subjacentes para oferecer uma visão global da rede é um dos principais desafios das plataformas de *middleware* para RSSFs
- **fusão/agregação de dados -** sensores são usados para coletar dados do ambiente, então coletar dados de vários sensores, unir, agregar e apresentar os dados em mais alto nível é outro importante desafio
- **restrições de recursos** - um middleware para RSSF deve ser leve para funcionar sobre *hardware* limitado em recursos
- **topologia dinâmica** - um *middleware* para RSSF deve ser capaz de lidar com a topologia dinâmica da rede, devido à mobilidade, falha de nós, e falha na comunicação entre os nós
- **conhecimento da aplicação** - um *middleware* para RSSF deve integrar conhecimento da aplicação nos serviços disponíveis, as otimizações de rede podem ser alcançadas com o conhecimento do nível da aplicação, por exemplo, características da aplicação podem influir tanto na infraestrutura da rede, quanto nos protocolos utilizados, já o conhecimento da aplicação pode ser aproveitado pela rede para que ela possa alcançar

uma maior eficiência em termos de consumo de energia prolongando, assim, o tempo de vida da rede

- **paradigma de programação** - os paradigmas de programação para RSSFs são diferentes de estilos de programação tradicional, devido às restrições de recursos, topologia dinâmica da rede, e dificuldades envolvidas na coleta e processamento de dados do sensor
- **adaptabilidade** - um *middleware* para RSSF deve suportar algoritmos que tenham desempenho adaptativo
- **escalabilidade** - um *middleware* para RSSF deve ser escalável em termos de números de nós, números de usuários, etc. para operar por longos períodos de tempo
- **segurança** - redes de sensores têm que tratar questões de segurança no processamento e comunicação de dados, mas devido às limitações de recursos e ao baixo poder computacional, a maioria dos algoritmos existentes e modelos de segurança não são adequados para as redes de sensores
- **suporte de QoS** - um *middleware* para RSSF deve também resolver várias questões de QoS, como por exemplo, tempo de resposta, disponibilidade, largura de banda, alocação, etc.

2.5 REST

REST (*REpresentational State Transfer*) [27] pode ser definido como um estilo arquitetural para a construção de sistemas distribuídos, provendo uma abstração da arquitetura da *World Wide Web* e cujos elementos fundamentais são chamados de recursos. Um recurso pode ser qualquer componente de uma aplicação que seja importante o suficiente para ser endereçável na Web por, no mínimo, um URI (*Uniform Resource Identifier*) [58], que é uma espécie de endereço que identifica unicamente tal recurso. Dessa forma, a abordagem REST envolve uma filosofia simples na qual um URI pode ser atribuído a qualquer recurso disponível em um servidor, recurso esse que pode então ser acessado e/ou manipulado por um cliente através de um dado conjunto de operações.

Nessa perspectiva, serviços *RESTful* [59], i.e. serviços que adotam o estilo REST, são menos acoplados, mais leves, eficientes e flexíveis do que os sistemas baseados em serviços Web usando chamadas remotas de procedimentos (RPCs – *Remote Procedure Calls*) e o protocolo SOAP (*Simple Object Access Protocol*), que introduzem uma camada adicional de

complexidade e dependência da linguagem XML [60].Através do uso do REST, é possível atender aos princípios do paradigma *Internet of Things* (IoT). Na IoT qualquer objeto físico, como os objetos do cotidiano (por exemplo, geladeira, fogão e um carro), pode estar conectado à internet. Neste cenário, a internet passa a conter dados do mundo físico e os objetos físicos se tornam os maiores emissores e receptores de trafego da rede. Ao utilizar REST é criada uma forma de acesso uniforme aos recursos da rede, da mesma forma que outros recursos disponíveis na Web. Isto é feito através de uma camada de abstração que esconde os detalhes de *hardware* e da plataforma de *software* utilizada na rede. No caso de RSSFs, cada serviço ou recurso disponibilizado pelo *middleware* pode ser mapeado em uma URI e com isto esse serviço ou recurso estará disponível na Web.

2.5.1 Princípios REST

O estilo arquitetural REST define um conjunto de princípios básicos para a construção de sistemas denominados *RESTful*. Tais princípios podem ser facilmente empregados utilizando o protocolo HTTP (*HyperText Transfer Protocol*) [58] e, por isso, esse protocolo tem sido amplamente utilizado no desenvolvimento de sistemas *RESTful*, visto que ele fornece o suporte necessário para a realização dos princípios REST, apresentados nas subseções a seguir.

2.5.1.1 Identificação única e global de recursos

Recursos são identificados única e globalmente através de um endereço URI [58], através do qual clientes podem interagir com tais recursos. URIs também são utilizados para indicar o escopo da informação, provendo meios que permitem a navegação entre recursos que interagem entre si, de modo que um URI pode identificar os sub-recursos relacionados a um recurso em um dado momento. Por exemplo, no URI “http://www.example.org/sensor-1469/light” light é um sub-recurso de sensor-1469.

2.5.1.2 Interface uniforme de acesso aos recursos

Recursos podem ser acessados e/ou manipulados através de um dado conjunto de operações definidas no protocolo HTTP, também chamadas de HTTP *verbs*, que são utilizadas para indicar ao provedor do recurso a ação que deve ser realizada, conforme apresentado na Tabela 1.

Tabela 1: Operações (HTTP verbs) para acesso e/ou manipulação de recursos em REST.

Operação (*HTTP verb*)	**Descrição**
GET	Recupera o estado atual de um recurso em uma dada representação
POST	Cria um novo recurso
DELETE	Remove um recurso
PUT	Atualiza o estado atual de um recurso; se o recurso ainda não existe, ele é criado

Por exemplo, considere-se a requisição HTTP representada na Figura 2.2. A operação GET especificada nessa requisição indica ao servidor (http://www.example.org) que o cliente espera receber do servidor uma representação do recurso light, que é sub-recurso de sensor-1469, conforme especificado pelo URI da requisição.

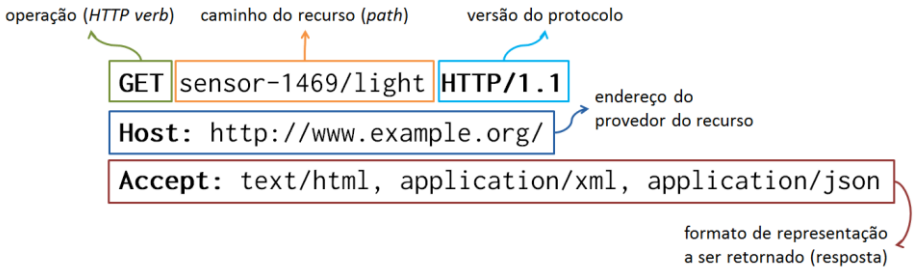

Figura 2.2: Requisição HTTP sobre um recurso

Apesar da simplicidade, essa característica é extremamente poderosa, pois define uma interface unificada para todos os recursos/serviços. Caso um determinado consumidor conheça os recursos oferecidos por um determinado serviço, ele passa a conhecer os processos de recuperação, criação, modificação e remoção de tais recursos devido à existência de uma interface unificada. Adicionalmente, essa característica fortalece o caráter interoperável dos serviços.

2.5.1.3 Múltiplas representações de recursos

Recursos são desacoplados de sua respectiva representação, de modo que o seu conteúdo pode ser acessado em uma variedade de formatos como, por exemplo, HTML (*HyperText Markup Language*), XML (*eXtended Markup Language*), JSON (*JavaScript Object*

Notation), etc., visto que o protocolo HTTP possibilita a utilização de diferentes representações e, com isso, as aplicações podem ser construídas independentemente da forma através da qual os dados são transferidos entre cliente e servidor. De maneira adicional, metadados acerca de um recurso podem estar disponíveis e serem utilizados, por exemplo, para negociação do formato de representação apropriado para uma dada requisição.

A especificação das representações que o cliente deseja receber podem ser passadas no campo Accept do cabeçalho (*header*) da requisição HTTP. Por exemplo, o campo Accept da requisição apresentada na Figura 2.2 indica três opções de representações que o cliente espera receber: text/html, indicando formato HTML; application/xml, indicando formato XML, e; application/json, indicando formato JSON.

2.5.1.4 Interações sem estado

Em serviços *RESTful*, toda interação com um dado recurso é dita *stateless*, i.e. sem manutenção de estado, de modo que as mensagens de requisição são autocontidas, ou seja, não pode haver manutenção de informações sobre o cliente em sessões no lado do servidor, e, com isso, cada requisição enviada ao provedor do recurso deve ter todas as informações necessárias ao seu processamento. Se alguma informação deve ser mantida sobre um dado recurso, esta deve ser mantida do lado do cliente, visto que, se um servidor mantivesse as sessões com informações de cada cliente, seu desempenho poderia ser afetado quando estivesse tratando de múltiplas requisições concorrentes.

Serviços *RESTful* não mantêm estado visto que o próprio protocolo HTTP sobre o qual eles são estruturados não oferece essa capacidade. Além disso, é possível ter uma redução da dependência do cliente com relação ao provedor do recurso, pois as requisições submetidas são independentes de um servidor específico, tornando possível a substituição transparente de um servidor em caso de falha do mesmo [61]. Por fim, esse tipo de interação *stateless* é baseada no conceito de transferência explícita de estado, que pode ser embutido nas mensagens para indicar estados futuros válidos da interação.

2.6 Modelo de comunicação assíncrona

O aspecto típico de uma comunicação assíncrona é que um emissor continua sua execução imediatamente após ter apresentado sua mensagem para transmissão. Isso

significa que a mensagem deve ser temporariamente armazenada para sua eventual entrega a um ou mais destinatários, os quais, portanto não precisam estar disponíveis/ativos simultaneamente ao emissor. Um dos modelos mais comuns de implementar o paradigma de comunicação assíncrona é o publish-subscribe [62].

No modelo de comunicação baseado no paradigma *publish-subscribe*, um ou mais *publishers* (publicadores) publicam eventos estruturados para um serviço de eventos e um ou mais *subscribers* (assinantes) expressam seu interesse em eventos particulares através de *subscriptions* (assinaturas). No contexto de RSSF, um assinante pode expressar interesse em dados de temperatura de uma determinada sala. O papel deste modelo de comunicação é fazer a ligação entre os eventos publicados e as assinaturas dos usuários garantindo a entrega das notificações (ocorrência dos eventos solicitados). Um mesmo evento pode ser distribuído para muitos assinantes e, portanto, este modelo de comunicação é fundamentalmente de um para muitos. Sistemas deste tipo são utilizados em uma grande variedade de domínios de aplicações, particularmente nos relacionados à disseminação em larga escala de eventos. Alguns exemplos são: sistemas de informação financeira, áreas com *feeds* em tempo real de informação, suporte à computação ubíqua (por exemplo, eventos de localização) e aplicações de monitoramento em geral.

Este modelo de comunicação possui duas principais características, que são a heterogeneidade e o inerente assincronismo. Com relação à primeira, quando a notificação de eventos é utilizada como meio de comunicação, componentes físicos ou lógicos em um sistema distribuído que não foram projetados para se comunicarem podem passar a trabalhar juntos. O necessário é que o componente responsável por gerar o evento publique os tipos de eventos que ele é capaz de gerar e que os componentes que consumirão estes eventos se subscrevam a estes eventos e, além disso, ofereçam uma interface para que as notificações possam ser entregues. Já com relação ao assincronismo, notificações são enviadas pelos *publishers* para todos os *subscribers* que expressaram interesse no evento gerado, sem a necessidade de haver sincronização entre as partes. *Publishers* e *subscribers* podem estar desacoplados no tempo.

Os elementos chave neste modelo de comunicação assíncrona são o serviço de notificação e o *buffer (Message Broker)* no qual as mensagens são enfileiradas antes de serem encaminhadas para seus assinantes. O serviço de notificação assume a

responsabilidade de informar aos assinantes quando uma nova mensagem está disponível. O *buffer* de mensagens é o responsável por enfileirar as mensagens de cada assinante, retirando assim a necessidade de conexão ponto-a-ponto entre produtor e consumidor, já que ambos se conectam ao *buffer* para adicionar/consumir mensagens. Deste modo é possível a comunicação assíncrona já que produtores e consumidores são completamente desacoplados. Este fraco acoplamento é a principal vantagem deste tipo de comunicação para ambientes *ad hoc* e ambientes pervasivos, como as redes de sensores sem fio.

A expressividade deste modelo de comunicação pode ser determinada de acordo com a abordagem de filtragem de assinaturas utilizado por ele. Algumas abordagens definidas são brevemente descritas a seguir.

(i) *channel-based* - nesta abordagem, publicadores publicam seus eventos em determinados canais e os assinantes devem assinar estes canais e filtrar os dados de acordo com sua necessidade

(ii) *topic-based* - nesta abordagem, assume-se que cada notificação é expressa em termos de um conjunto de campos, onde cada campo denota um tópico, esta abordagem é equivalente à abordagem *channel-based* onde cada tópico pode ser relacionado a um canal, mas ela pode ser melhorada ao se adicionar hierarquia aos tópicos; por exemplo, assinantes assinam o tópico de temperatura, mas poderiam ter assinado o tópico de temperatura na sala A (temperatura/salaA)

(iii) *content-based* - esta abordagem é uma generalização da abordagem *topic-based* permitindo a expressão de assinaturas sobre um conjunto de atributos de um evento, mais especificamente, um filtro baseado em conteúdo (*content-based*) pode ser considerado como uma consulta em termos de composição de restrições a ser aplicada sobre os atributos de certo evento

(iv) *type-based* - nesta abordagem, assinaturas são definidas em termos de tipos de eventos e a correlação entre o evento e as assinaturas é feita através dos tipos e subtipos gerados pelo evento.

Quanto ao sistema de filtragem que correlaciona os eventos gerados com as assinaturas presentes, este pode ser implementado basicamente de duas formas: (i) centralizado ou (ii) distribuído. Em (i) o *Message Broker* (*buffer* de mensagens) fica centralizado em um único ponto. Publicadores então publicam seus eventos para este ponto

que é responsável por distribuir para os devidos assinantes. Já em (ii), ao invés de um único ponto atuando como o *Message Broker,* há vários *Network Brokers* que cooperam para prover a mesma funcionalidade levando em conta a escalabilidade e eliminando o problema de um único ponto de falha.

2.6.1 Modelos de comunicação assíncrona em RSSF

No contexto de RSSF a maioria das aplicações segue um dos seguintes modelos de entrega de dados, que ditam o modelo de comunicação a ser adotado na rede: (i) consultas *one-shot*, que segue o esquema *request-reply*, onde é feita uma requisição do tipo "qual a temperatura média na área A ?"; (ii) consultas de longa duração, onde é feita uma requisição do tipo "qual a temperatura média nas próximas 24 horas na área A ?"; e (iii) consulta sobre a ocorrência de um evento, onde é feita uma requisição do tipo "A temperatura média na área A ultrapassou 30ºC?". Enquanto o primeiro modelo de entrega é tipicamente síncrono, esses dois últimos tipos de consultas tipicamente refletem um modelo assíncrono de comunicação.

Considerando, portanto, os requisitos da maioria das aplicações, a comunicação em redes de sensores sem fio é essencialmente assíncrona e baseada em eventos. Sabendo que os nós sensores da rede possuem atributos bem definidos (por exemplo, sua localização espacial, capacidades de sensoriamento, como luminosidade, umidade, temperatura, dentre outros), o modelo de comunicação assíncrona pode ser utilizado para requisitar e extrair dados da rede a partir de seus atributos. Neste modelo, cada nó sensor (que desempenha o papel de *Publisher*) anuncia um conjunto de atributos que descrevem os tipos de dados que ele pode oferecer. Então, a aplicação cliente (no papel de *Subscriber*) pode solicitar um subconjunto dos atributos oferecidos pelos nós sensores. Somente os dados coletados pelos nós sensores (*publishers*) que forem correspondentes aos atributos solicitados pela aplicação cliente (*subscriber*) são encaminhados a esta e outras possíveis aplicações, através da comunicação de um para muitos.

3 PRISMA

Este trabalho propõe uma plataforma de *middleware* orientado a recursos para redes de sensores sem fio, chamado PRISMA [63]. No sistema proposto, considera-se que várias aplicações podem executar na mesma rede, ao mesmo tempo ou em momentos diferentes. Cada aplicação pode configurar a rede de forma diferente, de acordo com suas necessidades (ou restrições). As necessidades indicadas pelas aplicações são usadas pelo *middleware* para configurar a operação dos nós sensores e para selecionar qual parte da rede deve ser utilizada para uma dada aplicação. Para acomodar a natureza dinâmica das RSSFs, o *middleware* permite que as aplicações sejam especificadas em tempo de execução.

O PRISMA assume uma topologia hierárquica para a RSSF. Este tipo de topologia lógica foi adotado por facilitar o gerenciamento da rede, especialmente em redes de larga escala, e por permitir que o controle de topologia seja realizado em dois níveis: (i) controle no nível de rede, com uma visão global desta e (ii) controle de nível de *cluster*, com a visão local do *cluster*. Além de uma topologia hierárquica, o PRISMA assume uma RSSF heterogênea, onde os nós sensores ordinários e líderes de clusters (*cluster heads)* possuem *hardwares* diferentes. Esta diferença se dá pela necessidade de um maior poder computacional nos *cluster heads,* que devem ser capazes de armazenar informações dos nós sensores de seu *cluster.* Este cenário de redes heterogêneas já tem sido explorado na literatura como pode ser visto, por exemplo, em [64]–[66].

Como visto no Capítulo 1, um *middleware* deve oferecer principalmente quatro funcionalidades: (i) abstrações de programação, (ii) serviços, (iii) suporte em tempo de execução e (iv) mecanismos de QoS (*Quality of Service*). Destas, o PRISMA em sua versão atual foca nas três primeiras, embora sua arquitetura já esteja preparada para contemplar um mecanismo de QoS baseado em controle de topologia.

A primeira funcionalidade, abstrações de programação, é atendida pelo PRISMA através da utilização de interfaces REST para abstrair a RSSF como um conjunto de recursos ou serviços, escondendo dos clientes os detalhes da infraestrutura de rede subjacente. Ou seja, o PRISMA provê uma abstração de programação baseada na ideia de *recursos*.

Os principais serviços (segunda funcionalidade oferecida) providos pelo PRISMA são o serviço de descoberta de recursos, o monitoramento de contexto e o serviço básico de comunicação, que provê suporte tanto ao modelo de comunicação síncrona quanto assíncrona. Com a RSSF já implantada e em operação é possível que usuários submetam aplicações em tempo de execução através das interfaces REST providas pelo *middleware*. Estas aplicações serão processadas e terão seus requisitos extraídos. Estes requisitos são analisados pelo *middleware* e só então as aplicações serão associadas aos respectivos nós sensores responsáveis por atendê-las. Nós estes que devem ser capazes de atender aos requisitos funcionais e não funcionais impostos, ou seja, os nós devem ser capazes de executar durante o tempo requisitado pela aplicação e devem ter a capacidade de suprir a aplicação com os dados de sensoriamento desejados. A possibilidade de submeter requisitos para a rede de forma dinâmica e, com isto, após análise por parte do *middleware*, um usuário poder configurar uma nova aplicação após a rede estar implantada e funcional (em tempo de execução), caracteriza a terceira funcionalidade provida (suporte em tempo de execução).

A seguir na Seção 3.1 é apresentada a arquitetura do PRISMA. É inicialmente dada uma visão dos componentes lógicos do *middleware* e uma breve descrição comportamental dos componentes de *software* que provêm às funcionalidades oferecidas. Na Seção 3.2 são detalhados todos os passos de operação do PRISMA. E, finalmente, na Seção 3.3 é apresentado o mecanismo de controle de topologia previsto pela arquitetura proposta.

3.1 Arquitetura do PRISMA

Esta seção descreve a arquitetura do middleware PRISMA proposto neste trabalho. Sua arquitetura é composta por componentes de *software* projetados de modo a serem implantados em 3 diferentes subsistemas de *hardware* existentes na rede: (i) *Gateway*, (ii) *Cluster Head* e (iii) Nó Sensor. Na Subseção 3.1.1, são descritos os componentes lógicos presentes em cada um dos subsistemas especificados para o *middleware*. Em seguida, na Seção 3.1.2 é apresentada a visão comportamental dos componentes, ou seja, sua operação para prover cada uma das seguintes funcionalidades providas: (i) modelo de entrega de dados baseado em requisição-resposta, (ii) serviço de Descoberta de recursos, (iii) modelo de comunicação assíncrona.

3.1.1 Componentes do *Middleware*

Considerando as características supracitadas assumidas pelo PRISMA de uma RSSF heterogênea e hierárquica, podemos dividir o *middleware* em três subsistemas, de acordo com os tipos de nós físicos onde os componentes de software serão instalados: (i) *Gateway*, (ii) *Cluster Head* e (iii) Nó Sensor. Dessa forma, a arquitetura lógica apresentada foi dividida em três partes, sendo uma para cada elemento físico da hierarquia.

O subsistema *Gateway* é responsável principalmente por intermediar a interação da RSSF com as aplicações clientes. O *Gateway* processa as requisições submetidas pelos clientes através de interfaces REST. Requisições essas que podem ser, por exemplo, a criação de uma nova aplicação na rede, consulta a dados históricos, consulta a dados atuais e verificação da disponibilidade de serviços na RSSF. Os dados coletados na RSSF são disponibilizados via web através de interfaces REST, podendo ser entregues de modo assíncrono (seguindo o modelo de comunicação assíncrona descrito na Seção 2.6) ou sincronamente (através do modelo de requisição e resposta). Além disto, o *Gateway* gerencia a rede com uma visão global (visão da rede como um todo). As decisões tomadas no *Gateway* são no nível do conjunto de *clusters* disponíveis na rede, por exemplo, em qual/quais *clusters* determinada aplicação deve ser implantada (os respectivos nós são configurados para atender aos requisitos por ela desejados). Já o subsistema *Cluster Head* é responsável por gerir o *cluster* específico em que está inserido, tratando da topologia local e encaminhando para o *Gateway* dados coletados pelos nós sensores do respectivo *cluster*. O subsistema *Nó Sensor* consiste basicamente de um componente de *software* responsável pela aquisição de dados, através de uma interface com as unidades de sensoriamento presentes no nó físico. Este componente de *software* é encarregado de coletar os dados de determinado fenômeno sendo monitorado.

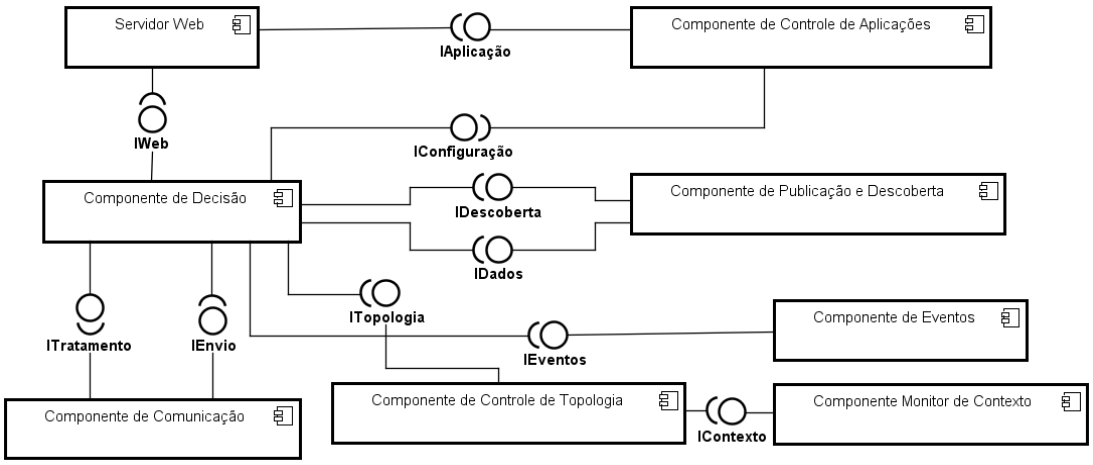

Figura 3.1: Componentes lógicos do subsistema Gateway

A seguir cada um dos componentes lógicos do subsistema *Gateway* é descrito com maior detalhamento. A Figura 3.1 apresenta os componentes de *software* (lógicos) a serem instalados no *Gateway*.

- **Componente de Controle de Aplicações** - responsável por gerenciar a criação de novas aplicações e o **Repositório de Aplicações**, as informações como: (i) tipos de dados desejados, (ii) frequência com que esses dados devem ser coletados, (iii) requisitos de QoS, tais como valores de atraso máximo ou acurácia mínima desejados, e (iv) os limiares a partir dos quais os dados devem ser transmitidos (no caso de modelos de entrega baseados em eventos) são armazenadas no repositório gerenciado por este componente
- **Servidor Web** - componente em que estão implantados os serviços web REST (*Representational State Transfer*) [27] para acessar as informações da rede, estas informações podem ser: (i) dados atuais ou históricos coletados no momento da requisição sobre algum fenômeno físico em alguma área geográfica e (ii) serviços oferecidos pela rede em cada área geográfica coberta pela rede, além disto, este componente é também responsável por enviar os dados coletados pela rede assincronamente para os clientes, funcionando como um *Message Broker,* além de responder a consultas *one-shot* do tipo *request-reply* (comunicação síncrona)
- **Componente de Publicação e Descoberta**- responsável pelo serviço de descoberta do *middleware* e por armazenar os dados coletados pela RSSF, este componente tem acesso a informações que incluem quais serviços estão disponíveis atualmente na

rede e em que área geográfica estes são providos, informações essas que são armazenadas no **Repositório de Serviços,** esse componente possui acesso também aos dados históricos armazenados no **repositório de dados**, e além de acessar estas informações, este componente é responsável por cadastrar novos serviços oferecidos por nós sensores adicionados recentemente na rede e por atualizar a disponibilidade dos serviços providos por nós que podem vir a ficar indisponíveis, além disto, é através deste componente que o **Componente de Decisão** checa a disponibilidade de serviços na rede para verificar se a rede possui capacidades para atender às aplicações clientes

- **Componente de Decisão** - componente que concentra a lógica de decisão do *middleware*, dentre as responsabilidades deste componente podemos destacar: (i) a análise dos serviços (capacidades de sensoriamento oferecidas pelos Nós Sensores da RSSF) para determinar se uma aplicação pode ser executada na rede, e (ii) a análise quanto às restrições impostas pela aplicação clientes para determinar se elas podem ser atendidas pelos nós da rede, por exemplo, consultar informações energéticas e determinar se a aplicação poderá executar pelo período solicitado pelo usuário, além disto, com o auxílio do **Componente de Comunicação** para traduzir mensagens vindas da rede, este componente interpreta todas as mensagens vindas de *cluster heads* e toma as decisões necessárias, tais como: (i) adicionar novo serviço ou atualizar a situação de um serviço existente, em caso de uma mensagem de descoberta de recursos, e (ii) encaminhar um dado coletado para uma determinada aplicação, em caso de uma mensagem de dados vinda da RSSF
- **Componente de Eventos** - componente responsável por gerir o **Repositório de Eventos,** através deste componente é possível consultar os eventos ativos na rede, descobrir o local em que um dado evento está sendo monitorado, descobrir os limiares aplicados a cada evento, a função de agregação utilizada (se solicitada) e através destes dados determinar se existe a necessidade de criar um novo evento para atender a alguma nova aplicação submetida pelo cliente ou se é possível reutilizar dados sendo gerados pela rede, estes eventos são representados no repositório através da descrição do tipo de dado a ser coletado, um operador relacional e um limiar

- **Componente de Comunicação** - responsável por receber as mensagens captadas por um *listener* responsável por escutar mensagens oriundas da RSSF e traduzi-las com o auxílio do **driver de comunicação**, que faz parte deste componente, para então enviá-las para o **Componente de Decisão** onde serão processadas e tratadas, além disto, é responsável por enviar mensagens para dentro da rede, mais uma vez com o auxílio do **driver de comunicação** para estruturar tais mensagens. O **driver de comunicação** é a parte do componente de comunicação responsável por receber as mensagens captadas pelo *listener* e extrair delas os dados coletados pela RSSF. Além disso, o *driver* faz o processo inverso e estrutura as mensagens a serem enviadas para dentro da rede. Mensagens trafegam na rede como uma *string* representando um pacote HTTP e são manipuladas em forma de objeto nos dispositivos. O **driver de comunicação** é responsável por traduzir a *string* que representa um pacote HTTP que é transmitida dentro do pacote padrão 802.15.4 transmitido pelos rádios de comunicação utilizados pelos nós sensores. O *driver* extrai esta *string* e a transforma em um objeto a ser manipulado pelo *Gateway*. O processo inverso também ocorre, ao enviarmos um pacote para dentro da rede, o *driver* monta a string a ser enviada e a coloca dentro do campo de dados do pacote padrão 802.15.4 que é enviado/recebido pelos nós da rede

  Componente de Controle de Topologia - no *Gateway* este componente de *software* é responsável por definir quais clusters da rede devem ser utilizados para uma dada aplicação e quais devem entrar em modo *sleep*, caso estejam ociosos; esta verificação leva em consideração todas as aplicações ativas na rede no momento e os requisitos impostos por uma nova aplicação, como, por exemplo, os serviços necessários, área geográfica alvo e tempo de vida da aplicação
- **Componente Monitor de Contexto** - este componente é responsável por analisar informações de contexto da RSSF, onde no caso da versão atual do PRISMA apenas a energia residual é analisada, outras informações poderiam ser coletadas, como, por exemplo, o *footprint* de memória, unidades de sensoriamento ativas, e etc. Neste subsistema (*Gateway)* este componente verifica o nível energético dos *clusters* disponíveis na RSSF.

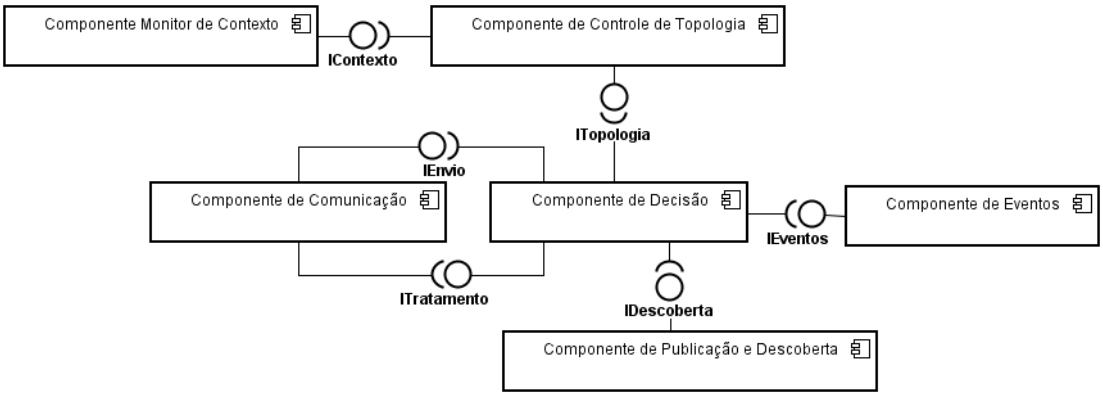

Figura 3.2: Componentes lógicos do subsistema *Cluster Head*

Na Figura 3.2 são apresentados os componentes lógicos do subsistema *Cluster Head.* Os componentes são basicamente os mesmos utilizados no subsistema *Gateway*, com exceção de algumas diferenças no comportamento de alguns componentes que são detalhadas em seguida.

- **Componente de Publicação e Descoberta** - no *Cluster Head* este componente é responsável pelo armazenamento das capacidades de todos os nós de um dado *cluster*, estas capacidades são configuradas em tempo de projeto e são solicitadas pelo **Componente de Decisão** assim que o nó inicializa
- **Componente de Decisão**- no *Cluster Head* o **Componente de Decisão** continua exercendo as funções que foram descritas no subsistema *Gateway* e é adicionada uma nova função, encaminhar as mensagens de anuncio contendo as informações armazenadas no **Componente de Publicação e Descoberta**
- **Componente de Controle de Topologia** - no subsistema *Cluster Head* este componente assume uma visão local do cluster, além disto, este componente também é responsável por selecionar quais nós do cluster são utilizados para atender as aplicações ativas em dado momento, utilizando informações de presença de unidades de sensoriamento relevantes, nível energético atual e número de aplicações já sendo executadas em determinado nó sensor
- **Componente Monitor de Contexto -** este componente é responsável por manter o nível de energia dos nós associados a este *cluster head* na **Base de Nós.**

Os outros componentes assumem funções similares às descritas no subsistema *Gateway*. O **Componente de Comunicação** continua sendo o responsável pela comunicação

e o **Componente de Eventos** continua sendo responsável por manter os eventos sendo monitorados no momento na **Base de Eventos**.

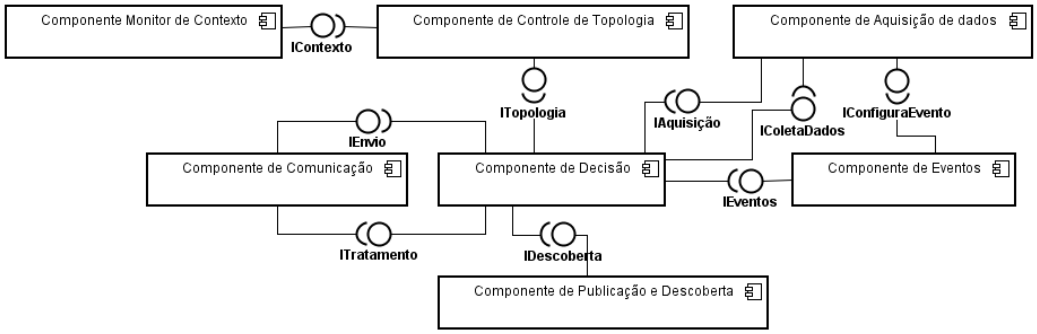

Figura 3.3: Componentes lógicos do subsistema Nó Sensor

E por último, na Figura 3.3, são exibidos os componentes lógicos do subsistema Nó Sensor. Novamente os componentes têm suas funcionalidades semelhantes às descritas anteriormente no subsistema *Gateway* e no subsistema *Cluster Head*, mas possuem algumas funcionalidades específicas deste componente que são citadas em seguida.

- **Componente de Decisão** - no subsistema Nó Sensor o **Componente de Decisão** assume uma nova responsabilidade além das citadas anteriormente, ele é o responsável por receber os dados coletados pelo **Componente de Aquisição de dados** e por enviar comandos de alteração de modo de operação ao **Componente de Controle de Topologia**
- **Componente de Controle de Topologia** - como citado no item anterior, este componente no Nó Sensor é responsável pela alteração do modo de operação do respectivo nó
- **Componente de Aquisição de dados** - este componente é o único que não está presente nos outros dois componentes físicos, já que somente o nó sensor é responsável por coletar dados do ambiente, além disto, é neste componente que são configurados os eventos monitorados no momento para cada nó
- **Componente de Eventos** - no subsistema Nó Sensor o **Componente de Eventos** assume uma nova responsabilidade, além de salvar os eventos que estão sendo monitorados no momento, este componente fica responsável também por configurar os eventos ativos no **Componente de Aquisição de dados** para que somente os dados relevantes sejam encaminhados para a rede.

Conforme visto nesta seção, o subsistema *Gateway* possui os componentes lógicos necessários para a comunicação com as aplicações clientes, através de suas interfaces REST. O subsistema *Cluster Head* possui os componentes necessários para gerir a rede em seu *cluster,* não necessitando se comunicar com as aplicações. Sendo assim, pequenas alterações de comportamento nos seus componentes lógicos devem ser notadas: (i) o **Componente de Controle de Topologia** passa a atuar localmente no *cluster* em que está gerindo, (ii) o **Componente de Decisão** passa também a anunciar as capacidades de seus nós e (iii) o **Componente de Publicação e Descoberta** armazena as capacidades do nó configuradas em tempo de projeto. Por fim, no subsistema Nó Sensor as alterações de comportamento dos componentes lógicos que devem ser notadas são: (i) **Componente de Decisão** recebe os dados do **Componente de Aquisição de dados** para enviar ao nó sorvedouro, (ii) **Componente de Controle de Topologia** fica responsável por alterar o modo de operação, (iii) o **Componente de Aquisição** coleta os dados e (iv) o **Componente de Eventos** configura os eventos no **Componente de Aquisição de dados.**

Através dos componentes lógicos descritos nesta Seção o PRISMA fornece seus três serviços básicos: (i) comunicação; (ii) descoberta de recursos; e (iii) monitoramento de contexto. O serviço de comunicação é responsável pela comunicação entre os componentes do *middleware* e os nós da RSSF, e com entidades externas (aplicações clientes ou a internet). Este serviço é provido pelos seguintes componentes: Componente De Comunicação para trocar mensagens dentro da RSSF; Servidor Web para comunicar com redes externas e com aplicações clientes. Vale ressaltar que mesmo que o componente de Decisão tenha participação no processo de comunicação ele não é citado. Isto ocorre por ser o componente responsável pela intermediação entre todos os componentes do *middleware* e como não faz participação direta na criação/envio das mensagens não foi citado como sendo parte do serviço. O serviço de descoberta de recursos é provido Pelo Componente De Publicação E Descoberta. Este serviço é responsável por identificar novos nós na rede e publicar novos serviços disponíveis para o centro decisório do *middleware*. O serviço de monitoramento de contexto é provido pelo Componente Monitor De Contexto e ele é responsável por monitorar a energia da RSSF/*Cluster*/Nó (dependendo do subsistema) e detectando condições de falta de energia.

3.1.2 Descrição Comportamental dos Componentes do Middleware

Nesta seção é descrito, através de diagramas de sequência UML, o comportamento realizado pelos componentes do PRISMA a fim de atender suas principais funcionalidades. As funcionalidades a são descritas em seguida.

- Entrega de dados segundo um modelo de **Requisição e Resposta** - esta é uma funcionalidade básica que pode ser acionada através de um serviço *web* REST. Através desta funcionalidade o usuário requisita dados sobre um fenômeno físico específico, podendo definir: (i) área alvo de monitoramento, (ii) interesse pelo valor máximo/mínimo/médio dos valores sensoriados na área e (iii) interesse por valores históricos considerando o intervalo de tempo desde a criação da aplicação na rede até o tempo atual ou algum outro intervalo definido
- Serviço de **Descoberta de recursos** do *middleware* - nesta funcionalidade é realizado o processo de descoberta da rede. Cada nó, assim que inicializado, anuncia suas capacidades para o seu respectivo *cluster head* e para o *gateway*. Ao receber esta mensagem de anúncio, tanto o *cluster head* quanto o *gateway* atualizam suas bases de dados com os dados recém-obtidos
- Entrega de dados segundo um modelo **de comunicação assíncrona** - esta funcionalidade é um dos diferenciais do *middleware* proposto neste trabalho. Através dela, aplicações clientes poderão criar canais de comunicação que são alimentados com dados relevantes e podem ser consultados de forma assíncrona, esta funcionalidade é descrita na Seção 3.1.2.1.

A funcionalidade de entrega de dados do tipo Requisição e Resposta foi apresentada em três diagramas de sequência distintos para melhor visualização, os quais estão mostrados nas seguintes figuras: (i) a Figura 3.4 apresenta o comportamento no subsistema *Gateway*; (ii) a Figura 3.5 apresenta o comportamento no subsistema *Cluster Head*; e (iii) a Figura 3.6 apresenta o comportamento no subsistema Nó Sensor. Além destas funcionalidades, existe ainda o serviço de monitoração de contexto, responsável por coletar os valores de energia residual dos nós sensores e *cluster heads*.

O processo se inicia na Figura 3.4, no *Gateway*, com uma requisição HTTP REST sendo enviada através de uma das interfaces REST oferecidas pelo *Gateway*. Inicialmente o *Gateway* trata a mensagem, extraindo os dados necessários da requisição, por exemplo: (i) área geográfica alvo, (ii) fenômeno físico de interesse e (iii) intervalo de tempo de interesse (para dados históricos). Com estes dados uma mensagem com os dados extraídos da requisição é construída e difundida na rede. Os dados resultantes desta requisição são então inseridos no Repositório de Dados e em seguida encaminhados como resposta à aplicação cliente que os solicitou. Vale ressaltar que a resposta é formatada em um arquivo XML que será enviado ao usuário.

A segunda parte do processo é ilustrada na Figura 3.5 onde o *Cluster Head* recebe a mensagem vinda do *Gateway*. A mensagem consiste em uma *string*, diferentemente da mensagem trocada entre aplicações clientes e o *Gateway* que possui o formato XML, que é recebida e traduzida. Logo em seguida, essa mensagem é tratada pelo Componente de Decisão que verifica e seleciona os nós ativos com o auxílio do Componente de Controle de Topologia. Uma mensagem é enviada para todos os nós que sejam relevantes para atender a requisição da aplicação. Quando as respostas (contendo os dados coletados) são recebidas no subsistema *Cluster Head* elas são então encaminhadas para o subsistema *Gateway*.

A última parte do processo é mostrada na Figura 3.6. O subsistema Nó Sensor fica responsável por receber a requisição e coletar os dados necessários e enviar para seu *cluster head*. Isto encerra o processo de Requisição e Resposta.

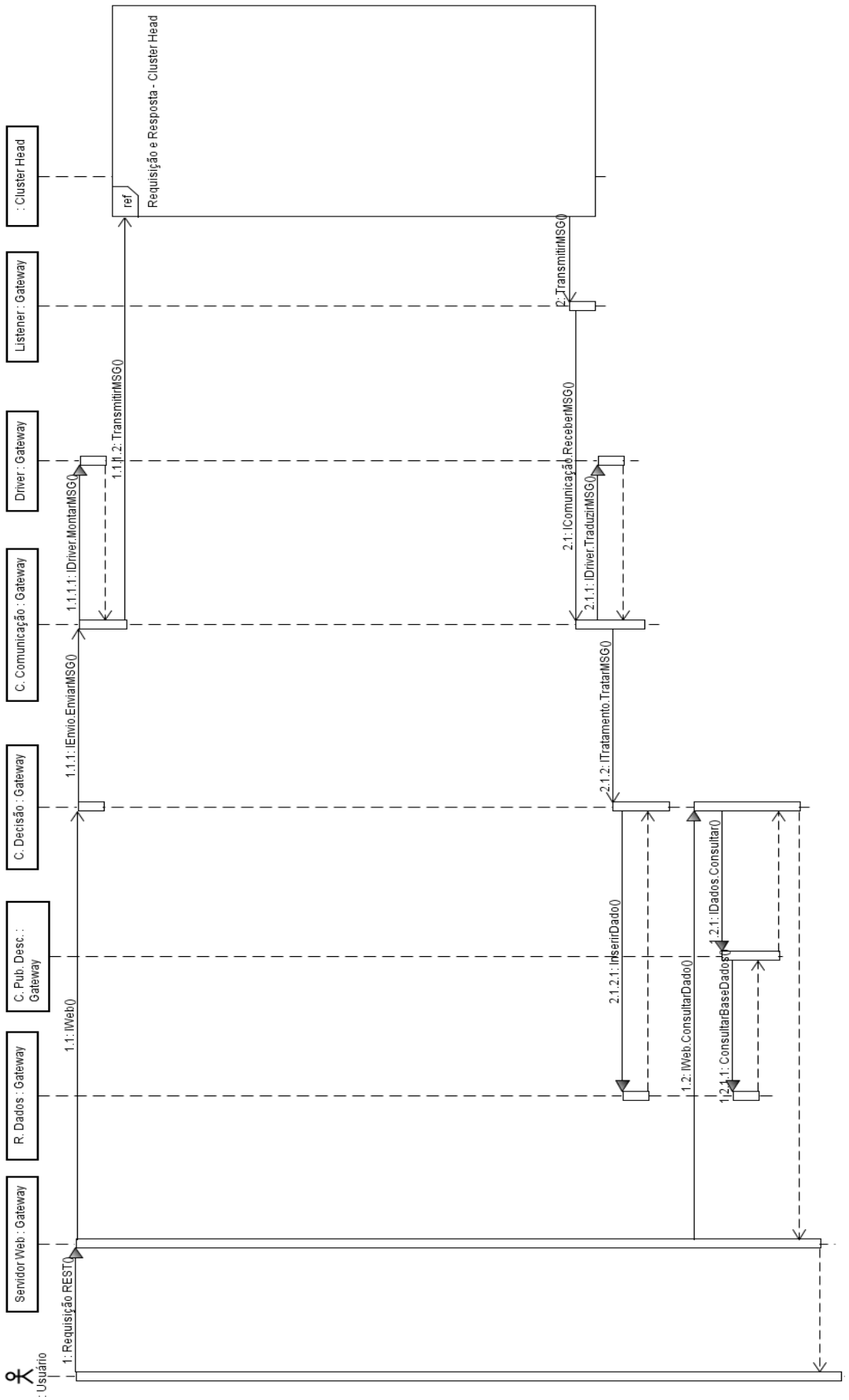

Figura 3.4: Requisição e Resposta - *Gateway*

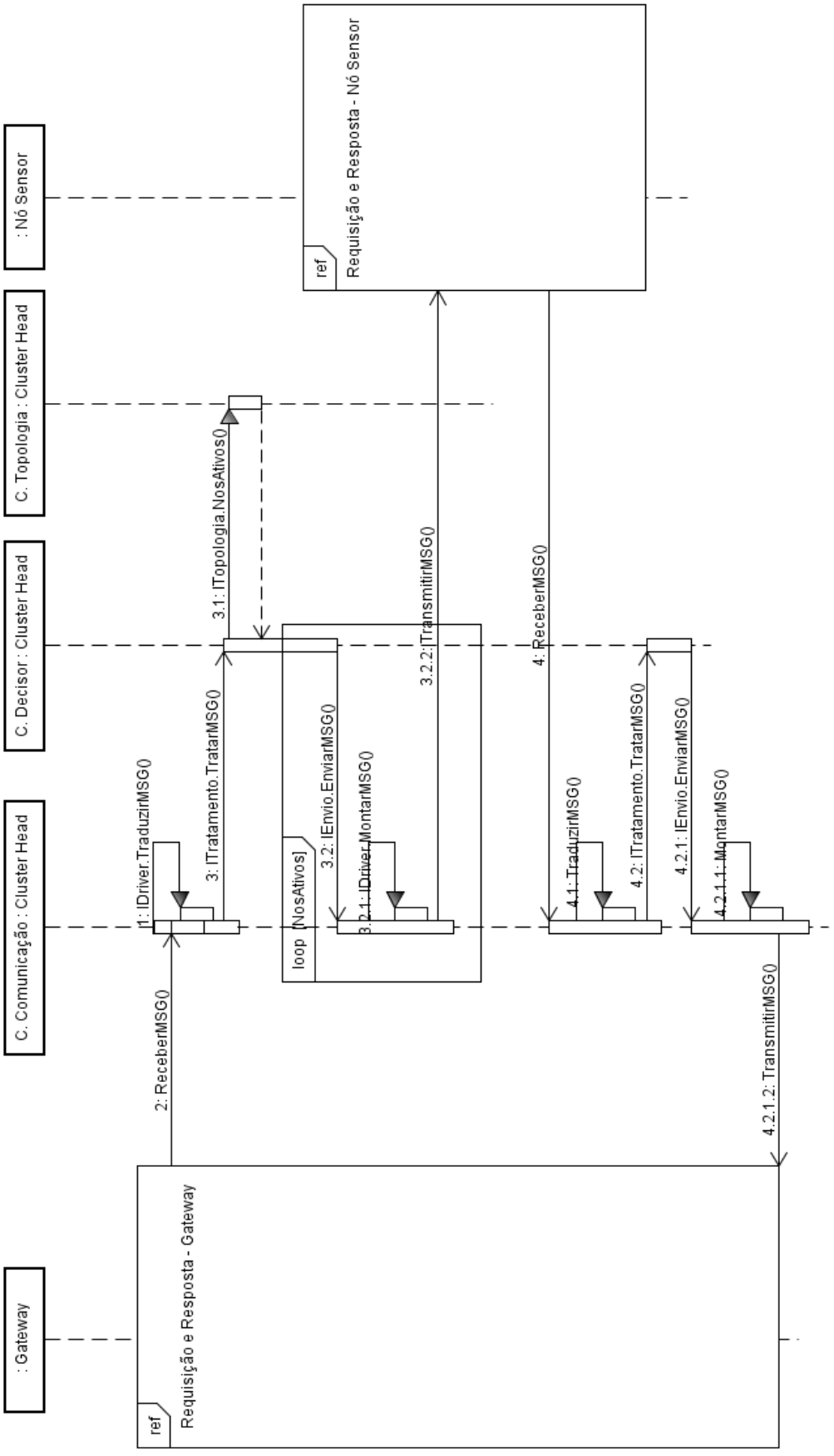

Figura 3.5: Requisição e Resposta – *Cluster Head*

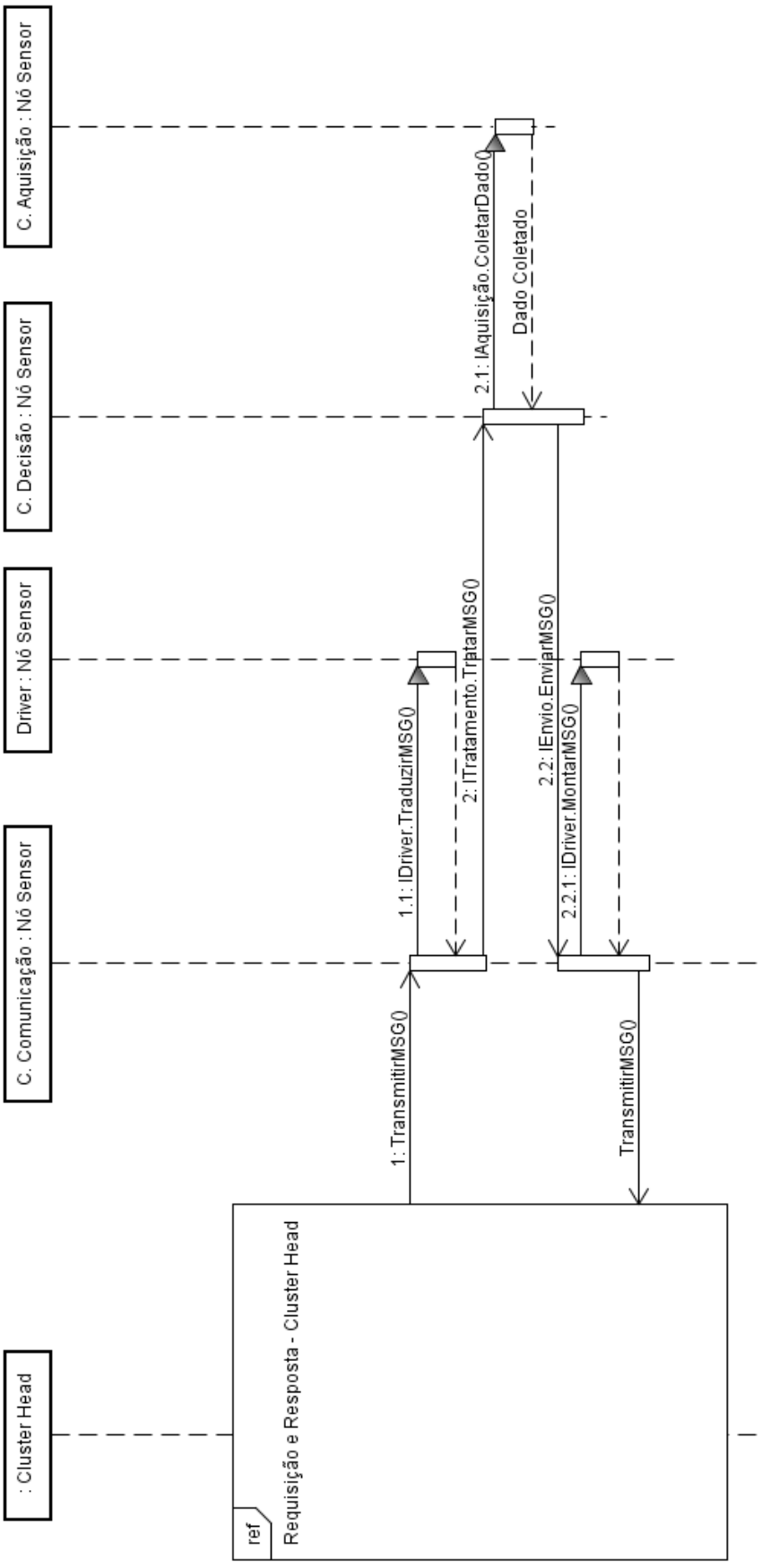

Figura 3.6: Requisição e Reposta – Nó Sensor

A seguir é descrito o comportamento dos três subsistemas do PRISMA para a realização do serviço de descoberta.

O processo de **Descoberta de recursos** se inicia no Nó Sensor e foi divido em três diagramas para melhor visualização, os quais estão ilustrados nas seguintes figuras: (i) a Figura 3.7 mostra o comportamento do subsistema Nó Sensor; (ii) a Figura 3.8 mostra o comportamento do subsistema *Cluster Head*; e (iii) a Figura 3.9 mostra o comportamento do subsistema *Gateway*.

A descoberta de serviços se inicia no subsistema Nó Sensor. Ao inicializar, o **Componente de Decisão** consulta o componente **Descoberta de Serviços** para determinar que capacidades o nó recém-inicializado possui. Em seguida o nó envia uma mensagem anunciando suas capacidades e espera uma resposta. Se não houver resposta o processo se repete até que o nó seja efetivamente inserido na rede, tornando o nó visível para o subsistema *Gateway*.

A segunda parte do processo acontece no *Cluster Head* e seu comportamento pode ser visto na Figura 3.8. O *cluster head,* ao receber um anúncio, adiciona o nó anunciante à sua **Base de Nós** como um nó que compõem o *cluster* que este gerencia. Em seguida, a mensagem é encaminhada ao *gateway*. A mensagem de resposta para o nó sensor tem origem no *gateway* e o nó *cluster head* apenas a encaminha, como ilustrado no diagrama.

A última parte do processo acontece no subsistema *Gateway* (Figura 3.9). Ao receber um anúncio da rede o *gateway* extrai as capacidades anunciadas, o ID do nó e sua área geográfica para armazenamento no **Repositório de Serviços**.

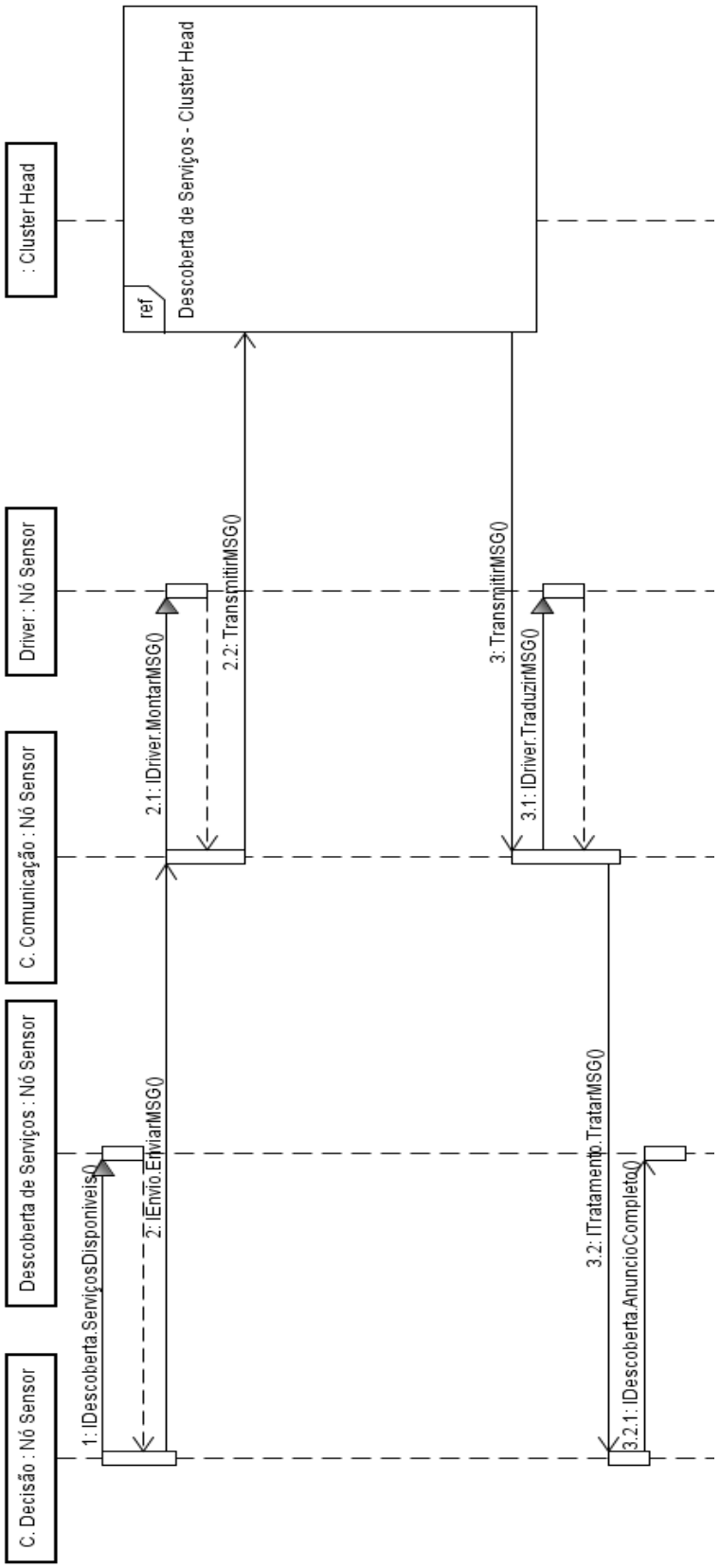

Figura 3.7: Descoberta de recursos – Nó Sensor

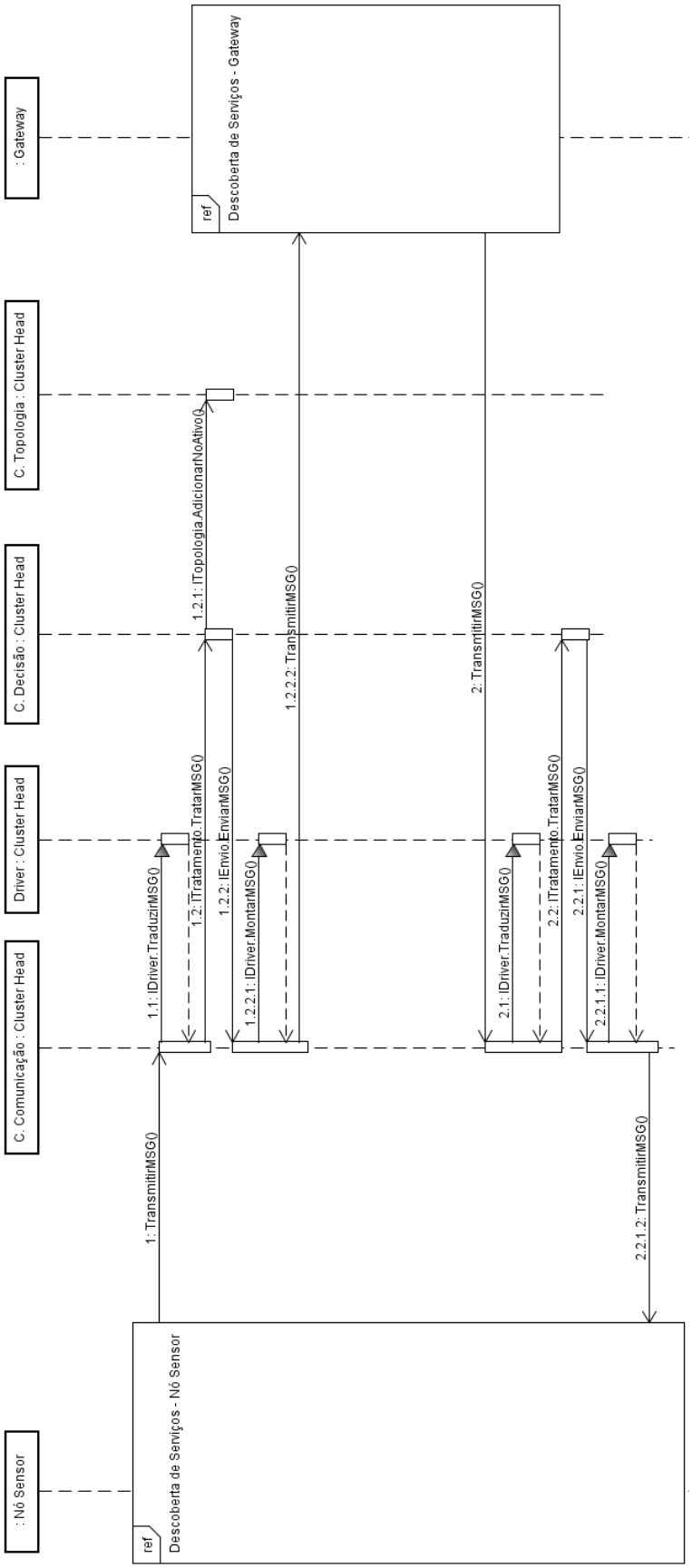

Figura 3.8: Descoberta de recursos – *Cluster Head*

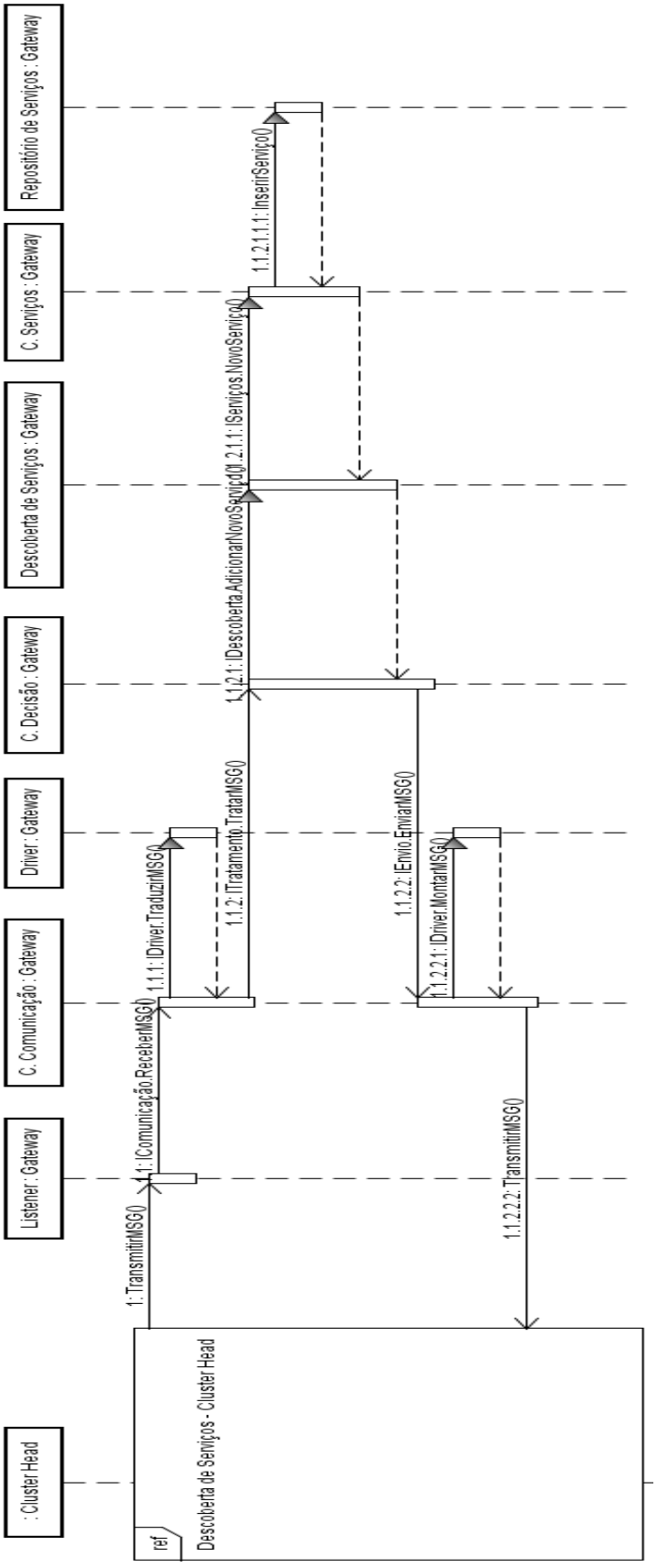

Figura 3.9: Descoberta de recursos - *Gateway*

Na Seção 3.1.2.1, a seguir, será detalhado o serviço comunicação assíncrona utilizado neste trabalho.

3.1.2.1 Serviço de comunicação assíncrona

O serviço de comunicação assíncrona adotado neste trabalho segue o padrão *publish/subscribe* descrito na Seção 2.6. Através deste padrão, o serviço permite a comunicação assíncrona entre o *gateway* e as aplicações clientes.

Diante das classificações exibidas na Seção 2.6, podemos classificar o serviço de comunicação assíncrona do PRISMA como sendo *Content-Based*. A escolha deste modelo se dá pelo fato de, como descrito na Seção 3.1, o cliente poder introduzir requisitos/restrições quanto aos dados a serem recebidos, e os dados a serem recebidos serão somente aqueles que atenderem a estes requisitos. O maior diferencial entre este modelo e o outro modelo mais utilizada na literatura, o *Topic-based*, se dá pela possibilidade de personalização dada ao *subscriber*. Na categoria *Topic-based* o *Publisher* define quais tópicos podem ser assinados, enquanto na categoria *Content-based* o *subscriber* ganha à possibilidade de criar seus próprios canais de comunicação que devem respeitar as restrições impostas por sua aplicação.

No PRISMA existem diversas capacidades de sensoriamento sendo oferecidas pelos dispositivos da rede, por exemplo, umidade, temperatura, detecção de gases inflamáveis, pressão, dentre outros. A gama total de opções de sensoriamento ira depender das unidades de sensoriamento disponíveis para a plataforma de rede de sensores sem fio adotada. A aplicação cliente pode então definir um conjunto de restrições sobre qualquer número destas capacidades de sensoriamento que devem ser atendidas. Neste caso, são criados os eventos. Ao mesmo canal de comunicação estabelecido são enviados os dados de todos os eventos que foram solicitados (exemplo na Seção 3.2). Os eventos são armazenados em um banco de dados relacional que descreve a associação entre um usuário, suas aplicações e seus respectivos eventos. Este banco de dados encontra-se no subsistema *Gateway* e sua modelagem é descrita no anexo B deste documento. As restrições necessárias são armazenadas e assim que um novo dado chega ao *middleware* este passará por filtros gerados por estas restrições e, caso o dado passe por todos devidos filtros de uma aplicação, será enviado à aplicação correspondente ao filtro verificado.

O processo de filtragem dos dados ocorre nos nós sensores e estes dados serão filtrados novamente no *gateway* para confirmação. Já os nós sensores são responsáveis pela menor granularidade no processo de filtragem, ou seja, eles estão em contato direto com cada unidade a ser coletada para que o evento seja detectado e, com isto, são filtrados logo de início dados que não correspondem aos interesses de aplicações clientes. A filtragem ocorre novamente em outras partes do *middleware* já que é possível que, por exemplo, solicitemos a informação de quando a temperatura é maior que 30º e a umidade abaixo de certo limiar em determinada área. Como o sensor não precisa conter sensores capazes de monitorar ambos os fenômenos físicos, podemos ter o caso do sensor ter somente o sensor de temperatura e, neste caso, ele não terá a restrição quanto à umidade. Caso em determinado sensor o dado coletado não seja válido para nenhum dos eventos registrados em seu repositório de eventos, este dado será descartado já na fonte e evitará desperdício energético ao enviar um dado que não será aproveitado por nenhuma aplicação.

Quanto ao sistema de filtragem que correlaciona os eventos gerados com as assinaturas presentes, o serviço de comunicação assíncrona utilizado no PRISMA é classificado como centralizado. Uma vez que todo o processo de comunicação se inicia pelo *gateway*, este funciona como um *Message Broker* que é o responsável por armazenar as mensagens que devem ser enviadas a cada cliente. O cliente então resgata estas mensagens diretamente do único ponto de contato com a rede, o *gateway*. Como o Message Broker reside apenas no *gateway*, podemos observar a questão do ponto de falha como observado nas limitações do escopo do trabalho, Seção 1.1. A seguir uma descrição do processo da comunicação assíncrona no PRISMA.

O processo da comunicação assíncrona se inicia no subsistema *Gateway* e foi dividido em três diagramas para facilitar a visualização, nas figuras a seguir: (ii) na Figura 3.10 é exibido o início do processo no *Gateway*; (ii) a Figura 3.11 mostra os passos do processo realizados no *Cluster Head*; e (iii) a Figura 3.12 mostra a última parte do processo, que ocorre no Nó Sensor.

A comunicação se inicia no subsistema *Gateway* com uma requisição HTTP REST sendo enviada através de uma das interfaces REST oferecidas. Inicialmente o *Gateway* trata a mensagem, extraindo os dados necessários da requisição e os repassando para o **Componente de Controle de Aplicações** que solicita a criação de uma nova aplicação com as

configurações especificadas pelas restrições impostas pelo usuário. Para tal, o subsistema *Gateway* verifica a disponibilidade dos serviços necessários e caso seja possível cria os eventos requisitados pela aplicação. Após a criação dos eventos, uma nova aplicação é adicionada ao **Repositório de Aplicações** e uma confirmação é enviada ao usuário.

A segunda parte do processo ocorre no *Cluster Head* que, ao receber a mensagem de criação de uma nova aplicação executa conjunto de regras para selecionar quais nós devem atender a esta nova aplicação, estas regras são baseadas na energia restante de cada nó e suas capacidades de monitoramento. Além disto, os possíveis eventos requisitados são registrados no *Cluster Head e* a seguir, uma mensagem é enviada para cada nó selecionado para a aplicação.

A última parte do processo ocorre no subsistema Nó Sensor que, ao receber a mensagem e processá-la criará novos eventos a serem monitorados e começará a coleta de dados caso ele já não os esteja coletando. Todo dado coletado que atenda a algum evento configurado no nó será enviado ao nó *cluster head* que encaminha estes dados para o nó *gateway.* O *Gateway* fica responsável por publicar estes dados coletados para a aplicação que os solicitou.

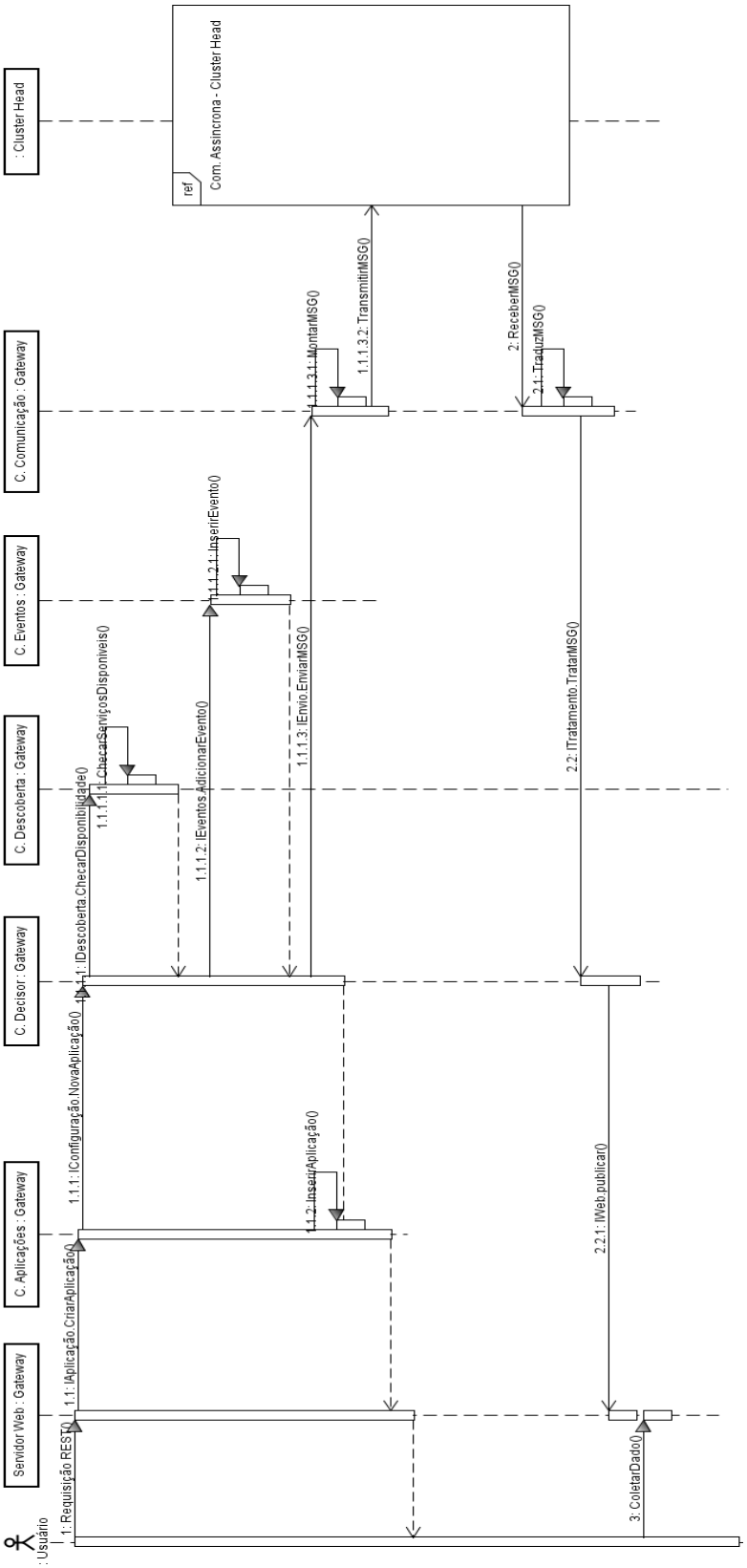

Figura 3.10: Modelo de Comunicação assíncrona – *Gateway*

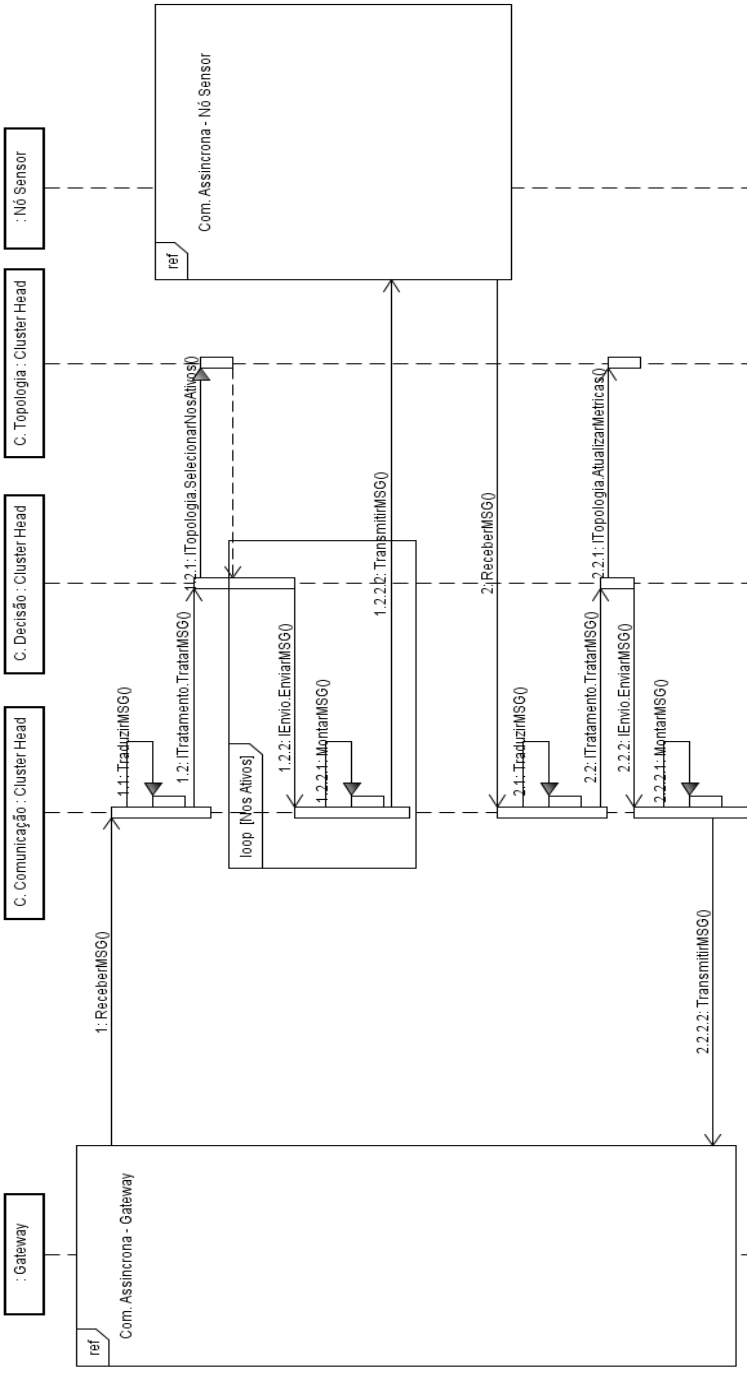

Figura 3.11: Modelo de comunicação assíncrona – *Cluster Head*

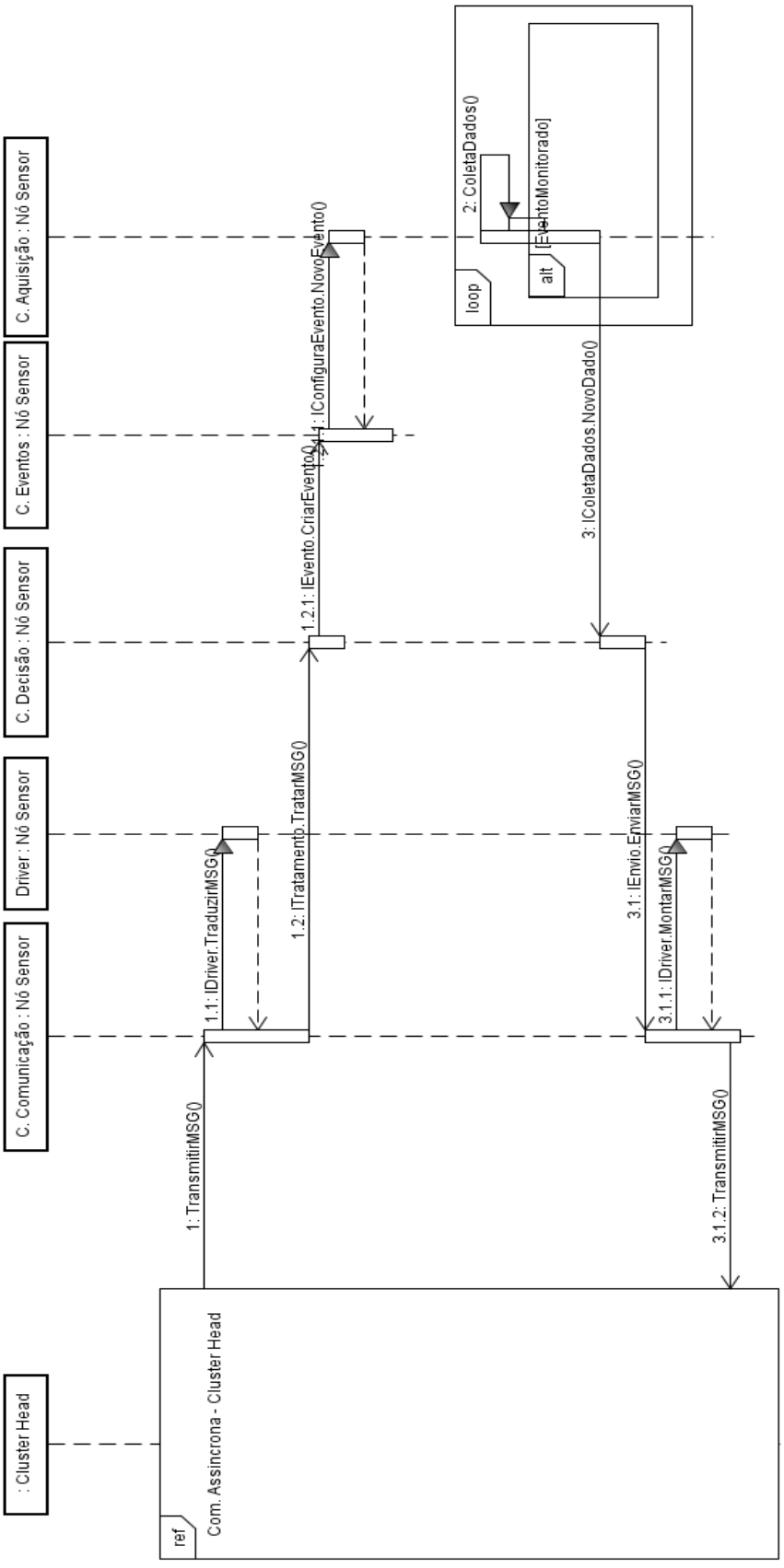

Figura 3.12: Modelo de comunicação assíncrona – Nó Sensor

3.2 Operação do PRISMA

Nesta seção é apresentada a operação de uma rede de sensores sem fio que adota o *middleware* PRISMA, desde o modelo de rede utilizado e a implantação dos nós até a criação e configuração de aplicações.

No modelo de rede adotado assume-se que todos os nós da rede conhecem suas localizações geográficas. Estas coordenadas são pré-configuradas seguindo um plano de instalação. Esta localização é transmitida em mensagens iniciais durante a inicialização da rede onde os nós anunciam suas capacidades. Os rádios de todos os nós possuem o mesmo alcance de transmissão. Além disso, os rádios possuem controle de potência, e a potência de transmissão pode ser alterada em tempo de execução.

Quanto à comunicação dos dados, essa é feita através de múltiplos saltos, desde a origem do dado (nós fontes) até o nó gateway, com nós intermediários (*cluster heads*) realizando agregação dos dados, sempre que solicitado pela aplicação. Cada sensor é capaz de monitorar uma área que pode ser definida como a área circular em torno do sensor com raio igual ao alcance do sensor.

Inicialmente, são implantados nos nós físicos os componentes do *middleware* que correspondem a cada uma das funcionalidades previstas, conforme os subsistemas definidos (*Gateway*, *Cluster Head* ou Nó Sensor), segundo descrito na Subseção 3.1.1. Em seguida, os dispositivos são distribuídos em suas respectivas áreas de maneira estática e assumindo um plano de instalação predefinido. A área de cada sensor é definida durante o planejamento da operação da RSSF e sua distribuição ocorre de acordo com o plano. Após a alocação dos sensores na área, entra em ação o serviço de descoberta. A etapa de formação dos clusters e atribuição de líderes atualmente é feita de forma estática, com a informação sobre qual *Cluster Head* está associado a qual conjunto de nós sendo pré-configurada nos respectivos nós sensores. No entanto, algum protocolo de clusterização poderia ser adotado, como por exemplo, protocolos baseados no LEACH [67], HEED [68], PEGASIS [69], etc.. Durante a fase de descoberta, os nós se identificam com seu *Cluster Head*, o qual em seguida, envia uma mensagem ao *Gateway* para identificar os novos nós ao centro de decisão do *middleware*, sendo esta mensagem chamada mensagem de descoberta. Nesta mensagem de descoberta endereçada ao *Gateway* estarão as seguintes informações: (i) endereço do nó, conforme definido no rádio de comunicação (ID do rádio), (ii) cluster em que está inserido (ID do

cluster), (iii) recursos disponíveis (capacidades de sensoriamento) e (iv) nível de energia (energia residual). O *Gateway* então atualiza sua base de dados com estas informações utilizando o Componente de Publicação e Descoberta. Essa etapa é importante pois, apesar de se seguir um plano de instalação, pode haver nós com mau funcionamento, problemas de falta de cobertura ou conectividade da rede (por exemplo, devido a obstáculos não previstos), de modo que a topologia real implantada pode diferir ligeiramente do plano original de instalação. Ao executar a etapa de publicação e descoberta, garante-se que as informações armazenadas no *Gateway* refletem da melhor forma o estado real da rede. Além da configuração inicial, esta mensagem de descoberta é utilizada para verificar se um nó ainda está ativo na RSSF, já que uma aplicação pode não ser associada a ele por um longo tempo. Outro uso desta mensagem de descoberta é para informar ao *Gateway* que um nó está com a energia quase esgotada e que não deve ser utilizado para uma nova aplicação. O componente Monitor de Contexto é o responsável por identificar quando a energia de um nó está quase esgotada e assim notificar o serviço de Descoberta. Este serviço irá então notificar o respectivo *Cluster Head* e o *Gateway* sobre o baixo nível de energia do nó. Sempre que uma mensagem de descoberta é enviada do nó, este componente verifica o nível energético deste nó para determinar se a energia restante está acima do limite configurado no nó. Deve-se evitar associar nós com energia residual abaixo deste limite a uma nova aplicação. Isto é feito quando uma nova aplicação deve ser criada na rede, quando todo nó com energia residual abaixo do limite ou próximo do limite é deixado em segundo plano e só é utilizado se não houver alternativa.

Uma vez que a rede esteja organizada e todos os seus recursos listados e disponíveis no *Gateway*, aplicações podem ser criadas na RSSF, através da configuração dos parâmetros necessários, como por exemplo, taxa de coleta, atraso máximo e tipos de sensores a serem utilizados. A criação de aplicações, como dito anteriormente, é feita através de uma das interfaces REST disponíveis no PRISMA através de seu servidor *web*. Esta interface espera receber um arquivo XML [70] através de uma mensagem HTTP [71] POST, na url: http://EndereçoDoServidor:8080/prisma/rest/aplicacao/criar. Um exemplo do arquivo é encontrado na Figura 3.13.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Aplicacao>
3      <username>username</username>
4      <taxaColeta>1000</taxaColeta>
5      <atrasoMax>1000</atrasoMax>
6      <tempoDeVida>15d</tempoDeVida>
7      <servicos>
8          <fenomenoMonitorado>Umidade</fenomenoMonitorado>
9          <areaGeografica>Lab. 1</areaGeografica>
10     </servicos>
11     <eventos>
12         <servico>
13             <fenomenoMonitorado>Temperatura</fenomenoMonitorado>
14             <areaGeografica>Lab. 2</areaGeografica>
15         </servico>
16         <limiteSuperior>35</limiteSuperior>
17         <limiteInferior>35</limiteInferior>
18         <operadores>">"</operadores>
19         <areaGeografica>Lab. 2</areaGeografica>
20     </eventos>
21 </Aplicacao>
```

Figura 3.13: Arquivo de configuração de nova aplicação

No arquivo está descrita uma aplicação a ser inserida na rede. Nele é possível definir os requisitos desta nova aplicação, que são: (i) taxa de coleta de dados, (ii) atraso máximo desejado, ambos em milissegundos, e (iii) tempo de vida da aplicação que pode ser definida em dias ou horas. Além destes requisitos, definem-se também quais fenômenos físicos serão monitorados e em que área geográfica devem estar localizados. No exemplo deseja-se coletar a cada segundo dados de **Umidade** da área geográfica **Laboratório 1**, a definição desta requisição se encontra entre as linhas 7 e 10 da Figura 3.13.

Podem-se também definir eventos que serão monitorados, especificando o fenômeno físico a ser monitorado, área geográfica alvo, limite superior/inferior e o operador a ser utilizado. No exemplo desejamos coletar dados de Temperatura que sejam maiores que 35º na área geográfica **Laboratório 2**, este evento foi definido entre as linhas 11 e 20, como pode ser visto na Figura 3.13. Após o envio deste arquivo de configuração, o middleware enviará os dados de forma assíncrona através de sua interface web. Ambas as tarefas de monitoramento devem permanecer ativas por 15 dias, como definido na linha 6.

O funcionamento do PRISMA está ilustrado no diagrama UML de sequência da Figura 3.14:

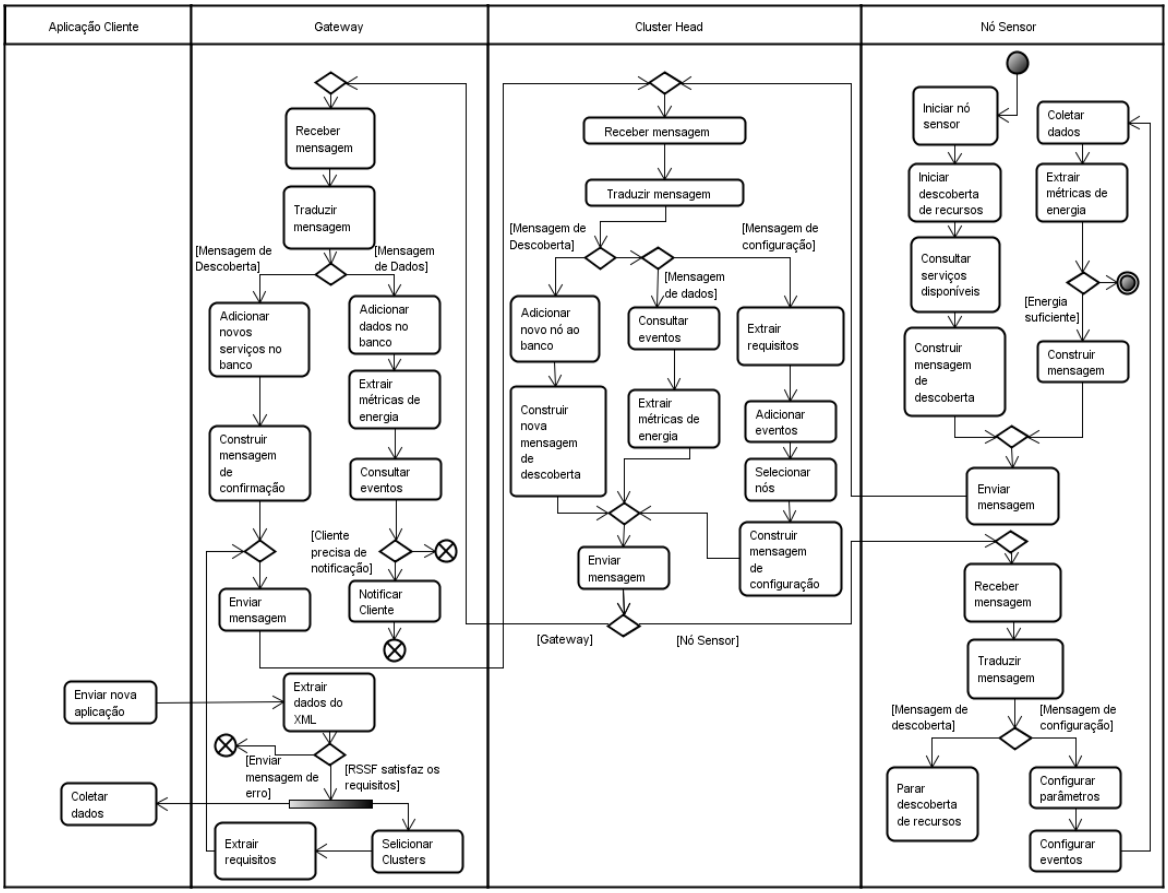

Figura 3.14: Diagrama de atividades descrevendo a operação do sistema

3.3 Controle de Topologia

Nesta seção é descrito o algoritmo de controle de topologia previsto para ser adotado pelo PRISMA. Inicialmente é mostrada uma visão geral do procedimento de controle de topologia em RSSF e em seguida detalhes do algoritmo adotado são apresentados.

Um dos principais objetivos no gerenciamento de nós em RSSFs é racionalizar o uso dos recursos de energia da rede, a fim de prolongar seu tempo de vida e consumir a energia homogeneamente entre os nós. Uma das formas de se atingir esse objetivo é adotar um esquema de rodízio do trabalho realizado pelos nós da rede, alterando o seu modo de operação (ativos ou em modo *sleep*). O subconjunto selecionado para permanecer ativo deve ser capaz de atender aos requisitos solicitados pela aplicação (ou aplicações) atualmente em execução na rede, enquanto garante a conectividade da rede.

Dada uma aplicação que submete uma tarefa de sensoriamento para a RSSF, o problema de seleção de nós ativos pode ser expresso como o algoritmo que decide quais

sensores devem permanecer ativos para a execução da tarefa. No PRISMA, o algoritmo de controle de topologia é provido como um dos serviços do middleware e utilizado como seu mecanismo de QoS, uma das quatro funcionalidades necessárias de um *middleware*.

O algoritmo de controle de topologia é executado pela primeira vez quando os requisitos de uma aplicação são submetidos para a rede. Requisitos da aplicação contêm a descrição da tarefa de sensoriamento (coordenadas da área alvo; tipo de sensor desejado; duração do monitoramento; funções de agregação de dados, se desejado) e os requisitos de QoS desejados. O algoritmo pode ser executado novamente nos seguintes casos: (i) sob demanda pela aplicação, quando deseja alterar algum parâmetro de QoS; (ii) proativamente pela rede, como medida de conservação de energia; e (iii) reativamente pela rede, quando detecta que algum requisito de QoS foi violado. Na adaptação feita do algoritmo original feita para o PRISMA, uma nova execução só acontece ocorre nos casos (ii) e (iii).

No PRISMA, o mecanismo de controle de topologia é executado em dois componentes físicos: (i) ***Gateway*** e (ii) ***Cluster Head***, onde cada um dos componentes executa o algoritmo em um nível diferente. No primeiro nível, executado pelo Gateway, ocorre uma pré-seleção baseada na área geográfica de interesse da aplicação cliente. Já no segundo nível, executado pelos *Clusters Heads*, ocorre à seleção dos nós ativos na área alvo desejada pela aplicação cliente.

O primeiro nível corresponde à visão global da rede (**Componente de Controle de Topologia** do ***Gateway***) onde o ***Gateway*** é responsável por determinar os *clusters* a serem utilizados para uma dada aplicação: *clusters* que não possuírem recursos ou capacidades para atender uma dada aplicação já serão excluídos do processo de seleção de nós ativos, isto sendo feito a partir de um conjunto de regras para eliminação de *clusters*. O segundo nível, local dentro de cada *cluster*, executará nos coordenadores de *cluster* (***Cluster Heads***) por estes possuírem uma maior capacidade de processamento que os nós sensores e para distribuir o processo de seleção de nós ativos. Os nós trabalham com apenas um rádio e o processo de alterar o modo de operação do nó será realizado através de mensagens de controle/configuração do rádio para configurar o *duty cycle* de cada nó.

Um pseudocódigo foi desenvolvido para o mecanismo de controle de topologia e deverá ser implementado como trabalho futuro. Temos abaixo a primeira parte do pseudocódigo, correspondente ao primeiro nível do mecanismo de controle de topologia, a

ser executada no ***Gateway*** para seleção de *clusters* presentes na área geográfica de interesse da aplicação cliente. Como entrada o algoritmo recebe a aplicação (um ID e um conjunto de serviços necessários, retirados do arquivo de configuração) e seus requisitos, como por exemplo, área alvo, taxa de coleta, taxa de envio e duração da aplicação. Além disso, o algoritmo também recebe como entrada o conjunto de capacidades de monitoramento disponíveis na rede. O algoritmo tem como saída o conjunto de *clusters* a ser utilizado pela aplicação. Este conjunto consiste de uma lista composta pelos *Ids* dos *clusters* a serem utilizados pela aplicação.

O algoritmo inicia com a divisão da aplicação em um conjunto de capacidades necessárias $S = (s_1, s_2, .., s_n)$ e para cada capacidade *i* da lista é criada uma lista de nós candidatos LC_i. Na segunda parte é verificado o menor número de clusters que conseguem cobrir todas as capacidades solicitadas, verificando os *clusters* que estão presentes no maior numero de listas LC_i.

No passo final do algoritmo no *Gateway*, mensagens são encaminhadas aos *Clusters Heads* que são líderes dos clusters escolhidos para participar na aplicação. Nesta mensagem estarão contidas as informações da aplicação necessárias para a configuração dos nós sensores e o registro desta aplicação no *Cluster Head*. Após o envio da mensagem, os eventos são armazenados no *Gateway* e, caso necessário, uma confirmação é enviada ao usuário. Com isto está finalizado o algoritmo no *Gateway*, mostrado no pseudocódigo a seguir, Figura 3.15.

Entrada: Requisitos da aplicação, Conjunto de capacidades disponíveis na RSSF Saída: Conjunto de *clusters* a serem utilizados

//Checagem da disponibilidade das capacidades requisitadas
Ao chegar uma nova aplicação no Gateway, a aplicação é então decomposta em uma lista de capacidades $S = (s_1, s_2, .., s_n)$
While size(S) > 0 **do**
Para uma determinada capacidade $s_i \in S$, checar a disponibilidade da capacidade consultando o componente de software **Descoberta de Serviços** (guarda informações das capacidades disponíveis na rede)
Cria uma lista de nós candidatos LC_i e adiciona os nós sensores com a capacidade s_i disponível e que estejam na área geográfica desejada
Remove a capacidade s_i da lista S
End While
//Seleção dos *clusters* necessários para a aplicação
For each nó candidato na lista LC_i
For each nó $n_j \in LC_i$ **do**
Adicionar *cluster* ao qual pertence o nó n_j na lista C_i

End For

For each lista C_i

Verificar ocorrência de cada *cluster* em cada capacidade de monitoramento

End For

Retornar lista de *clusters* que possuem as maiores ocorrências, de forma a atender todas as capacidades solicitadas

End For

Enviar mensagem aos clusters selecionados para a criação da aplicação, informando os requisitos, eventos, e capacidades de monitoramento a serem utilizadas

Registrar novos eventos, definidos através do arquivo de configuração, a serem monitorados

Confirmar criação da aplicação ao usuário

Figura 3.15: Algoritmo de pré-seleção de clusters

Com isto está concluído a pré-seleção das áreas nas quais acontecerá a seleção de nós ativos na rede, correspondente ao segundo nível.

No PRISMA a seleção dos nós que ficarão ativos para uma aplicação ocorre principalmente dentro do subsistema ***Cluster Head***, que é responsável por gerenciar os nós localmente. Este mecanismo se inicia quando é solicitada a criação de uma nova aplicação, e apenas nos *clusters* que foram pré-selecionados pelo primeiro nível, ou quando o nível de energia de algum nó ou de alguma área está abaixo de um limite mínimo para a execução de suas tarefas.

O processo se inicia com a consulta aos níveis de energia dos nós do cluster, mantidos pelo ***Cluster Head*** da área. Após a execução do algoritmo de seleção de nós ativos, uma mensagem é enviada a cada nó do *cluster* definindo se este estará ativo ou em modo *sleep*, caso este esteja em modo *sleep*, na próxima vez que acordar irá consultar ao seu *Cluster Head* para verificar mensagens pendentes e então receberá sua mensagem de controle. Métricas de energia são coletadas e enviadas juntamente com os dados coletados por estes sensores para abastecer seu ***Cluster Head*** com os níveis de energia atuais da rede.

Na Figura 3.16 é apresentada a segunda parte do pseudocódigo, correspondente ao segundo nível, que fará parte do mecanismo de controle de topologia e será executado nos *cluster heads* selecionados no processo anterior.

Entrada: Requisitos da aplicação, Conjunto de capacidades disponíveis no *cluster*

Saída: Conjunto de nós a ficarem ativos

//Listar possíveis nós para cada capacidade

Ao chegar uma mensagem indicando à criação de uma nova aplicação no Cluster Head, a aplicação é então decomposta em uma lista de capacidades de monitoramento $S = (s_1, s_2, .., s_n)$

While size(S) > 0 **do**

Para uma determinada capacidade $s_i \in S$, checar quais nós do *cluster* podem oferecer o recurso necessário consultando o registro de nós do *cluster*

Criar uma lista de nós candidatos LC_i e adiciona os nós sensores com a capacidade de monitoramento s_i disponível

Remover a capacidade s_i da lista S

End While

//Seleção dos nós a permanecerem ativos de forma a maximizar a energia residual dos nós utilizados

For each nó candidato na lista LC_i

For each nó $n_j \in LC_i$ **do**

Selecionar um conjunto de nós com maior energia residual

Adicionar nós na lista de candidatos à capacidade i (CS_i)

End For

For each lista CS_i

Selecionar menor conjunto de nós para atender à aplicação, maximizando a energia residual dos nós ativos

End For

Retornar lista de nós a serem utilizados

End For

Enviar mensagem aos nós selecionados para a criação da aplicação e configuração dos Eventos

Enviar mensagem aos outros nós que não estejam sendo utilizados pela rede para entrarem em modo *sleep*, através da configuração de parâmetros do rádio, especificamente os parâmetros SM e SP.

Registrar novos eventos a serem monitorados no ***Cluster Head***

Figura 3.16: Algoritmo de seleção de nós ativos que executa no subsistema *Cluster Head*

4 Trabalhos Relacionados

Nessa Seção serão discutidos trabalhos relacionados ao PRISMA na vertente de *middleware publish-subscribe* e *middlewares* orientados a recursos para redes de sensores sem fio. Em seguida, será feita uma breve comparação entre as funcionalidades providas pelos trabalhos selecionados e as funcionalidades previstas no PRISMA, dentro das três funcionalidades que são abordadas nesta dissertação: (i) abstração de programação, (ii) serviços e (iii) suporte em tempo de execução.

4.1 TinyDDS

Em [72] é descrito o TinyDDS (*Toward Interoperable Publish/Subscribe Communication between Wireless Sensor Networks and Access Networks*), *middleware* com as características de ser (re)configurável, de código aberto, baseado no padrão de projeto em camadas, e tratar da interoperabilidade de comunicação entre aplicações baseada em *publish-subscribe*. O TinyDDS é dividido em cinco camadas, brevemente descritas a seguir. A camada de APIs fornece um subconjunto de interfaces DDS que podem ser utilizadas pelas aplicações para criar tópicos, subscrever eventos dos tópicos e publicar eventos. Esta camada utiliza uma implementação leve, adaptada para RSSFs, do serviço de distribuição de dados (*Data Distribution Service* – DDS) que o grupo OMG (*Object Management Group*) padroniza para *middlewares publish-subscribe* baseados em tópicos. A camada OERP (*Overlay Event Routing Protocols*) é utilizada para transportar eventos publicados (por exemplo, dados sensoriados) através de uma rede *overlay*. A camada 5 (*Layer L5*) é responsável pelo protocolo TinyGIOP que consiste em um subconjunto do GIOP (*General Inter-ORB* Procotol), padronizado também pela OMG. Este protocolo é independente dos protocolos das camadas de enlace, rede e transporte do modelo OSI e encapsula/transmite os dados formatados com o padrão TinyCDR. O TinyCDR consiste em um subconjunto do CDR (*Common Data* Representation), também padronizado pela OMG. Ao utilizar uma versão simplificada destes dois padrões para sistemas embarcados, o TinyDDS pretende fornecer interoperabilidade de comunicação *publish-subscribe* entre a RSSF e outras redes de acesso. Na camada 4 (*Layer L4*) reside a TinyDDS *Adaptation Layer* (L4AL) que é uma

camada abstrata de rede utilizada para acessar a rede física, separando a implementação da rede *overlay* da implementação da rede física. E por último a camada 3 (*Layer L3*) é responsável pelos protocolos de roteamento do *middleware*. O *middleware* TinyDDS é composto de duas partes, a definição das interfaces DDS e a implementação TinyDDS dessas interfaces.

Dentre as funcionalidades que um *middleware* para RSSF deve fornecer, no trabalho [72] o mecanismo de abstração utilizado baseia-se no padrão DDS, usado para abstrair os dispositivos da rede como um conjunto de recursos disponíveis para o servidor central. Tal abordagem difere do PRISMA, que se utiliza de interfaces REST para prover a abstração e visa abstrair as capacidades da RSSF como recursos para a *web*. A abstração provida pelo PRISMA facilita ao usuário final o aproveitamento das capacidades da RSSF já que os recursos são disponibilizados na *web*. Além de possuir menor *overhead* e ser mais simples quando comparado ao padrão DDS, a principal diferença é que no TinyDDS os recursos são abstraídos para um servidor central, ou seja, para o próprio *middleware* enquanto no PRISMA os recursos são abstraídos para a *web*. Outra diferença está no fato da interface REST ser uma interface nativa com a web, sem necessidade de outras camadas para possibilitar a comunicação com o exterior ou a adição de complexidade na transmissão dos dados.

Quanto aos serviços providos, podemos destacar o serviço de comunicação assíncrona interno à rede obtido com o uso do padrão DDS, onde no TinyDDS as estações base (*sinks*) são os *subscribers* e os nós da RSSF são os *publishers.* A mesma abordagem é usada no PRISMA, com o diferencial de que os *subscribers* são aplicações clientes e, com isto, a comunicação assíncrona se dá tanto dentro da RSSF quanto com clientes externos ao *middleware*.

Já no tocante ao suporte em tempo de execução, podemos destacar o mecanismo de autoconfiguração desempenhado pelo algoritmo evolucionário MONSOON que é responsável por atualizar os parâmetros configuráveis de QoS da implementação DDS sobre a plataforma TinyOS especificada em [72].O MONSOON extrai métricas no servidor central do TinyDDS e avalia cada *Publisher* para estabelecer se as métricas estão aceitáveis de acordo com as restrições impostas pela aplicação. No trabalho são destacados dois parâmetros: (i) atraso e (ii) confiabilidade. No PRISMA, suporte em tempo de execução é

provido pela criação de aplicações dinamicamente conforme descrito na Seção 3.2. Outro diferencial do PRISMA com relação a este trabalho é o fato do PRISMA ser orientado a recursos. Suas capacidades são abstraídas como recursos para a *web* e por utilizar o padrão RESTful permite uma maior interoperabilidade. Apesar do padrão DDS também prover interoperabilidade, é necessária outra camada de abstração para possibilitar que os dados sejam acessíveis via internet e traduzidos pela API DDS para só então serem processados. Diferente do que acontece ao utilizarmos interfaces REST, por serem nativamente compatíveis com a web (protocolo HTTP) mesmo dispositivos simples podem resgatar estes dados da web, o que é interessante no conceito de IoT onde qualquer dispositivo poderá se integrar com o *middleware* consumindo e processando os dados gerados sem necessidade de complexidade adicional imposta pelo uso de uma API para tradução dos dados.

4.2 MiSense

MiSense[73] é um middleware orientado a serviço e baseado em componentes projetado para permitir que sensores suportem aplicações distribuídas com diferentes requisitos de desempenho. Ele reduz a complexidade de programação impondo uma estrutura na parte superior do modelo de componentes adotado pelo middleware e oferecendo interfaces de serviço bem definidas para o resto do sistema, abstraindo das aplicações e usuários as peculiaridades de *hardware* e *software* da plataforma de sensor utilizada. MiSense fornece um serviço de comunicação *publish-subscribe* baseado no conteúdo bem definido disponibilizado pelos dispositivos da rede (por exemplo, temperatura, umidade, etc.), mas também permite que o desenvolvedor da aplicação possa adaptar o serviço, fornecendo novos tópicos, além dos bem definidos citados anteriormente. O middleware é dividido em três camadas: a camada de Comunicação, que fornece um serviço de *publish-subscribe* chamado de MiPSCom, que permite que o desenvolvedor da aplicação possa adaptar o serviço criando novos tópicos. A camada de Serviços Comuns, que fornece serviços comuns a diferentes tipos de aplicações, tais como a agregação de dados, detecção de eventos, gerenciamento de topologia e roteamento. A camada de Domínio, que visa apoiar as necessidades específicas de domínio, tais como fusão de dados e suporte a semântica dos dados para permitir a produção de informação relevante à aplicação relativa ao processamento dos dados brutos. A interface de programação fornece um conjunto de

funções que permitem ao usuário programar a rede de sensores como um todo, sem se preocupar com a alocação de tarefas de processamento de dados e comunicação.

Voltando para as funcionalidades que um *middleware* de RSSF deve oferecer, no quesito de abstração de programação podemos destacar as abstrações oferecidas pelos serviços do MiSense. Os autores do trabalho deixam claro que a abstração provida por eles visa maximizar os ganhos de produtividade abstraindo detalhes de *hardware* e *software* em forma de um conjunto de serviços; já no PRISMA as capacidades da RSSF são abstraídas como recursos para a *web* e acessadas via REST.

Na funcionalidade de serviços o MiSense fornece um conjunto de serviços básicos de uso geral de aplicações para RSSF que, em alguns casos, podem até ser aproveitados para uso como serviços específicos de domínio. Apesar dos autores deixarem claro que novos serviços podem ser adicionados/configurados no *middleware* através do componente de extensão de serviços, os serviços oferecidos pelo MiSense são genéricos. O modelo de comunicação proposto no MiSense, onde cada nó possui seu próprio *Broker*, gerencia seus tópicos e assinaturas, pode vir a sobrecarregar os nós em condições de altas cargas de trabalho, onde um novo dado coletado terá que ser transmitido para todas as aplicações/nós subscritos e, como visto anteriormente, o rádio é o maior responsável pelo gasto de energia de um nó. No PRISMA o gateway toma estas responsabilidades e trabalha como intermediário entre as aplicações e a RSSF, evitando troca excessiva de mensagens inerente ao uso do modelo de comunicação assíncrona. No PRISMA são disponibilizados menos serviços, mas que são mais robustos quando comparados aos serviços oferecidos pelo MiSense.

O suporte em tempo de execução é tratado de forma diferente em ambos os trabalhos. No MiSense ele se dá pela adaptação dos *clusters* ao se elegerem novos *cluster heads* quando da ocorrência de falhas em dispositivos ou por insuficiência energética. Já no PRISMA esta funcionalidade é atingida ao criarmos novas aplicações na RSSF em tempo de execução. Além da criação são escolhidos, também em tempo de execução, quais nós devem suprir as necessidades de uma nova aplicação através de um conjunto de regras. Filtrados os nós, os requisitos da aplicação são enviados aos nós e configurados on-the-fly. Apesar de ambos os *middlewares* proverem esta funcionalidade não há como compararmos as soluções encontradas.

Novamente atentamos para o fato de o PRISMA ser orientado a recursos. Suas capacidades são abstraídas como recursos para a *web* e por utilizar o padrão RESTful permite uma maior interoperabilidade entre os recursos do *middleware* e dispositivos que tenham acesso à web. Diferentemente do proposto neste trabalho.

4.3 MufFIN

MufFIN[18](*Middleware For the Internet of thiNgs*) é um *middleware* orientado a serviços que utiliza os princípios de SOA e o padrão Sensor Web Enablement (SWE) para prover uma camada de abstração utilizada para implantação de código e comunicação entre dispositivos heterogêneos, possibilitando aplicações clientes programar tais dispositivos e gerenciar/distribuir os dados coletados através de serviços *web*. O padrão SWE visa permitir que todos os dispositivos possam ser descobertos, acessíveis e utilizáveis via Web. Os autores escolheram acomodar as diferenças entre os dispositivos no nível de *middleware*, e na perspectiva de uma aplicação todos os dispositivos são reprogramáveis e se comunicam. Para o MufFIN todos os dispositivos são reprogramáveis porque mesmo que a plataforma não possua tal característica, é criado um filtro no *middleware* para processar as informações recebidas pela rede. Portanto, para a aplicação será como se o dispositivo estivesse executando o código, mas na realidade o dado coletado pelo dispositivo passará por um filtro no *middleware* e somente após tal passagem ele será entregue ao cliente. A arquitetura do MufFIN é composta por um conjunto de 7 módulos fracamente acoplados. Estes módulos consistem em: (i) *thingsGateway* , (ii) *WS-Gateway* , (iii) *Core*, (iv) *DFN-Engine,* (v) *Subscriptions* , (vi) SOS (*Sensor Observation Service*) e (vii) *DataAcess.* Dois módulos fazem papel de *Gateways*, sendo o *thingsGateway* para comunicação com os dispositivos inteligentes e o *WS-Gateway* responsável pela comunicação com as aplicações através de serviços *web*. Com isto, o MufFIN fornece suporte à comunicação síncrona e assíncrona. O módulo *Core* é responsável pela implementação da interface *web* e tratamento das invocações de serviços. O módulo *DFN-Engine* gerencia as aplicações implantadas pelo usuário e suas dependências; este módulo instancia as aplicações e cria as conexões *publish-subscribe* para sua comunicação. O módulo *Subscriptions* recebe as assinaturas dos clientes, as processa, publica as notificações e salva informações sobre os clientes. O processamento dos documentos XML é feito pelo módulo SOS (*Sensor Observation Service*). E, por último, o

módulo *DataAcess* provê um conjunto de operações para permitir acesso aos dados coletados. O MufFIN foi implementado em JAVA e executa na plataforma *Fuse Enterprise Service Bus* (*Fuse ESB*), baseada no *Apache ServiceMix*. *Fuse ESB* é uma arquitetura que facilita a integração entre serviços e a criação de fluxo dinâmico entre os componentes do sistema. Como trabalho futuro os autores se comprometeram a testar e validar o *middleware* em um ambiente real.

Voltando às funcionalidades que um *middleware* deve oferecer, no MufFIN os autores focaram seus esforços em prover abstração na interação entre as aplicações e os objetos inteligentes, tanto em questão de comunicação e capacidades de programação (*hardware* e *software*) através de filtros criados dentro do *middleware* quanto através de comunicação entre serviços heterogêneos através de um formato comum, neste caso o XML. No PRISMA temos esta mesma abstração de *hardware* e *software*, mas provida através de suas interfaces REST que abstraem as capacidades da rede como recursos para a *web* também fazendo uso do formato XML.

No tocante a funcionalidade de serviços providos, o MufFIN fornece um conjunto de serviços básicos de comunicação (assíncrono e síncrono) e podemos destacar seu serviço de criação de eventos, o que corresponderia à criação de aplicações no PRISMA, e seu serviço de descoberta que propaga informação sobre os módulos (capacidades de sensoriamento da rede) para o *middleware*. No PRISMA, além destes serviços básicos de comunicação e descoberta, temos também o serviço de monitoração de contexto para auxiliar no monitoramento energético, uma área que não é citada pelos autores no trabalho [18].

Quanto à terceira funcionalidade, suporte em tempo de execução, o MufFIN dá suporte à criação de eventos da mesma maneira que podem ser criadas aplicações no PRISMA, ou seja, através do envio de arquivos XML contendo as descrições das capacidades necessárias e requisitos que devem ser atendidos. O *middleware* então é responsável por criar o evento. Porém, diferentemente do PRISMA, esta criação de eventos não é feita sempre no nó sensor da RSSF. No MufFIN, quando não é possível a instanciação do evento no sensor, seja por obstáculos de programação ou *hardware*, é criado um filtro no próprio *middleware*. As mensagens que estão sendo recebidas passam por este filtro que extrai os dados necessários para disparar o evento solicitado. No PRISMA não existe essa

possibilidade, aplicações são criadas e instanciadas em um ou em diversos nós da RSSF para responder às necessidades do cliente.

Este trabalho também não adota o estilo arquitetural orientado a recursos. Suas capacidades são abstraídas como recursos para a *web* e por utilizar o padrão RESTful permite uma maior interoperabilidade entre os recursos do *middleware* e dispositivos que tenham acesso à web. Outro diferencial a favor do PRISMA.

4.4 Mires

Mires [74] é um *middleware* baseado em serviços e fundamentado no paradigma *publish-subscribe*. O Mires opera acima da camada do sistema operacional TinyOS encapsulando suas interfaces e provendo serviços de alto nível para as aplicações. Internamente o Mires é composto por um serviço *publish-subscribe*, um componente de roteamento e serviços adicionais. Segundo aos autores, serviços adicionais podem ser facilmente agregados ao middleware e integrados ao serviço de *publish-subscribe* caso implementem as interfaces necessárias. No Mires a comunicação entre os nós se dá em três fases. Inicialmente os nós na rede anunciam suas capacidades (no contexto de *publish-subscribe*, tópicos). Em seguida, as mensagens de anúncio são roteadas até o sorvedouro utilizando um algoritmo de roteamento *multi-hop*. Uma aplicação cliente pode se conectar ao sorvedouro para selecionar um dos tópicos anunciados para ser monitorado (dessa forma realizando sua subscrição naquele tópico). Por fim, mensagens de subscrição são difundidas para os nós da rede indicando quais tópicos estão sendo requisitados. O principal componente do Mires é o serviço *publish-subscribe*. Este serviço é responsável por intermediar a comunicação entre os serviços de *middleware* e também é responsável por anunciar os tópicos providos, manter uma lista de tópicos subscritos por aplicações clientes e publicar mensagens contendo dados relacionados aos tópicos anunciados. A principal característica deste serviço é que somente mensagens relacionadas a tópicos subscritos são encaminhadas, visando à economia de energia. Estas mensagens são encaminhadas utilizando o componente de roteamento. Este componente de roteamento, por ser independente, possibilita o uso de qualquer algoritmo de roteamento *multi-hop*, desde que implementem as interfaces definidas pelo Mires. O componente *Publish-Subscribe* provê 4 interfaces: *Advertise* e *Publish* para os nós sensores e *PublishState* e *Notifier* para os serviços

a serem implementados. As interfaces *Advertise* e *Publish* permitem ao nó sensor anunciar suas capacidades e enviar mensagens relacionadas a algum tópico de interesse. Já as interfaces *PublishState* e *Notifier* permitem a interação entre o componente *publish-subscribe* e serviços adicionados ao *middleware*, a interface *PublishState* permite ao serviço enviar seus resultados para o tópico de interesse e a interface *Notifier* define os eventos que necessitam ser manipulados pelo serviço para ser notificado da chegada de novos dados coletados

Quanto às funcionalidades de um *middleware* para RSSF, o Mires aborda abstração apenas no contexto de esconder as peculiaridades de manipulação específica do *hardware,* provendo interfaces de mais alto nível para acesso das funcionalidades disponíveis. No PRISMA, como visto, esta abstração se dá ao utilizarmos interfaces REST para disponibilização das capacidades da rede como recursos para a *web*.

A segunda funcionalidade, provisão de serviços, é fornecida pelo Mires através de seu conjunto de serviços básicos gerais (agregação de dados, roteamento e descoberta de recursos) e serviços de comunicação síncrona e assíncrona. Os autores dão ênfase ao serviço de comunicação assíncrona implementado pelo *middleware*. Toda a comunicação do *middleware* é feita assincronamente. Já o PRISMA possui um serviço que não é abordado pelo Mires, o serviço de monitoramento de contexto. Com este serviço o PRISMA consegue gerenciar a RSSF de forma mais eficiente ao obter valores de nível energético de cada nó para poder, através destes valores, determinar que nós estão disponíveis para atenderem novas aplicações.

Quanto à terceira funcionalidade, suporte em tempo de execução, os autores descrevem um cenário em que tópicos são enviados aos nós sensores para que estes comecem a coletar dados de interesse, da mesma forma como se dá a criação de uma aplicação no PRISMA. Porém não são fornecidos detalhes de como esta requisição é feita, se é através da *web* ou se é local do próprio *middleware*. Neste caso podemos considerar o caso de que aplicações são criadas em tempo de execução, mas diferentemente do PRISMA não podemos afirmar que estes dados são disponibilizados para a *web* de maneira assíncrona.

Outro diferencial do PRISMA com relação a este trabalho é a adoção do estilo arquitetural orientado a recursos, que juntamente com seus serviços RESTful possibilitam uma maior interoperabilidade com outros dispositivos e até mesmo outros *middlewares*.

4.5 MARINE

MARINE [75] é um *middleware* baseado em componentes, especificamente projetado para RSSFs, que foi implementado pelo nosso grupo de pesquisa. O MARINE adota os padrões arquiteturais REST e *microkernel*. MARINE provê um serviço de comunicação baseado no REST e, para tratar do dinamismo do ambiente de RSSF e a necessidade de otimização dos recursos, o MARINE provê serviços de inspeção, adaptação e configuração. Novos serviços podem ser criados por terceiros e incorporados ao *middleware* ao utilizar as interfaces de programação e o modelo de componentes providos. A comunicação assíncrona é provida pelo protocolo *PubSubHubbub*. O MARINE cria um *Hub* em cada nó sensor para que cada requisição ao nó seja encaminhada diretamente a ele, criando um grande consumo energético já que os nós são acessados diretamente. MARINE provê todas as funcionalidades requeridas por um *middleware.*

Detalhando as funcionalidades necessárias de um *middleware*, destacamos a abstração de programação. O MARINE prove uma abstração baseada em REST que esconde peculiaridades dos dispositivos da RSSF das aplicações clientes ou usuários, além de possibilitar a integração dos dados com outros recursos da *web*. A mesma forma de abstração é provida no PRISMA. Tanto o PRISMA quanto o MARINE utilizam o estilo arquitetural orientado a recursos juntamente com a adoção de serviços RESTful para abstrair suas capacidades para a *web* e assim possibilitarem uma maior interoperabilidade.

Os serviços providos pelo MARINE incluem o serviço de comunicação baseado em REST, o serviço de inspeção/adaptação e configuração. Os autores deixam claro que novos serviços podem ser adicionados ao *middleware* se forem implementados seguindo o modelo de componentes e a API proposta. O serviço de inspeção/adaptação prove capacidades reflexivas ao *middleware*, assim como o serviço de monitoramento de contexto do PRISMA.

O suporte em tempo de execução no tempo de execução é tratado no MARINE através das propriedades reflexivas do *middleware* que analisa o próprio contexto para alterar suas configurações em tempo de execução, como o componente monitor de

contexto deste trabalho. Em ambos os *middlewares* é possível à criação de novas aplicações em tempo de execução.

5 Implementação

Foi desenvolvida uma implementação da arquitetura de middleware proposta a fim de validá-la, bem como para executar testes de desempenho do sistema. Para melhor organização esta seção foi subdividida em quatro partes, a saber: ambiente de implementação (Seção 5.1), implantação (Seção 5.2), interfaces gráficas (Seção 5.3) e lições aprendidas (Seção 5.4).

5.1 Ambiente de implementação

Na implementação desenvolvida, o subsistema Gateway funciona como um servidor de aplicações e foi construído com o Apache Tomcat 8[76], sobre a plataforma J2EE 1.4, utilizando os *frameworks*: *Jersey* para a criação das interfaces REST; Log4J para manipulação de logs; e *Hibernate* para a comunicação entre o subsistema *Gateway* e o *SGBD* relacional MySQL[77], utilizado para construir os bancos de dados implementando os repositórios do PRISMA. O sistema foi desenvolvido seguindo o padrão de projeto *Data Access Object* (*DAO*[78]). A comunicação assíncrona foi desenvolvida utilizando o *framework Jersey* para a comunicação não bloqueante entre o cliente e o *middleware*.

Como descrito no Capítulo 1, a atual plataforma de sensores alvo da implementação do PRISMA é o Arduino. A principal motivação para esta escolha é que esta plataforma é recente (lançada em 2005), *open hardware*, não explorada extensivamente pela comunidade acadêmica de rede de sensores, especialmente na área de *middleware*. Tanto quanto é do nosso conhecimento, atualmente não existe implementação de *middleware* de RSSF para esta plataforma. Finalmente, a plataforma utiliza uma linguagem de alto nível que facilita o desenvolvimento de aplicações.

Tanto o subsistema *Cluster Head* quanto o subsistema Nó Sensor foram desenvolvidos parcialmente utilizando a IDE de desenvolvimento do Arduino, chamada Arduino IDE e disponibilizada em seu site oficial [4]. Ao verificar que esta IDE não era precisa o suficiente em suas informações de *debug* como, por exemplo, não era apresentada a causa de um erro de compilação, a linha em que ocorre o problema, qual problema ocorreu (comunicação não pode ser estabelecida com o dispositivo via USB, comunicação não

sincronizada – *baud rate* – incorreto, etc.), decidiu-se por utilizar uma nova interface para prosseguir com o desenvolvimento. Tal interface consistiu no programa Sublime Text [79], utilizado juntamente com um plugin denominado Stino [80], o qual transforma este editor de texto em uma IDE para desenvolvimento para a plataforma Arduino. Os nós foram programados na linguagem *Arduino Programming Language* [81], a qual é baseada na linguagem de programação *Wiring* [82]. Os blocos de construção da linguagem *Wiring* são divididos em 3 categorias principais: estruturas, valores (variáveis e constantes) e funções. Por ser baseada em C/C++ [83], funções de ambas as linguagens podem ser desenvolvidas ou utilizadas. Como dito anteriormente, o PRISMA considera uma rede de sensores heterogênea, onde o *Cluster Head* necessita de maior poder computacional. Portanto, o Arduino MEGA foi escolhido para a implementação do subsistema *Cluster Head*, por ter maior poder computacional. Já o subsistema Nós Sensores utiliza o modelo Arduino UNO, ilustrados na Figura 5.1.

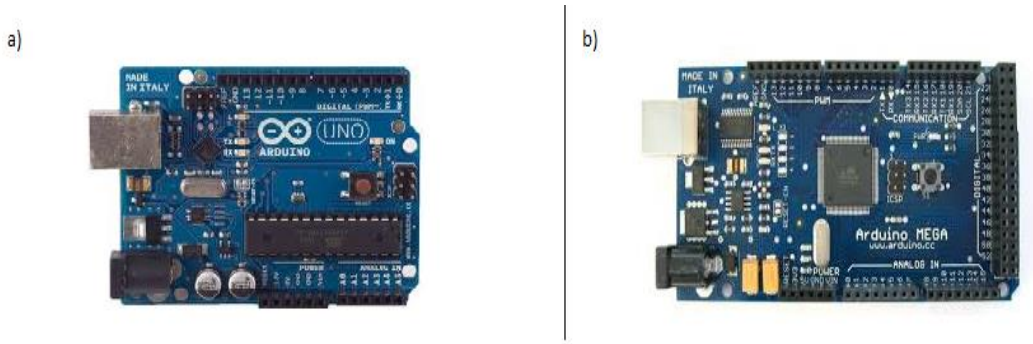

Figura 5.1: A) Arduino UNO; B) Arduino MEGA

Mais detalhes sobre cada um destes modelos podem ser encontrados na Tabela 2, a seguir:

Tabela 2: Comparação dos modelos de Arduino

	Arduino UNO	Arduino MEGA
Microcontrolador	ATmega328	ATmega1280
Voltagem de operação	5V	5V
Voltagem de entrada recomendada	7-12V	7-12V
Limite de voltagem de Entrada	6-20V	6-20V
Pinos digitais de entrada e	14 (6 podem	54 (15 podem

saída	prover energia)	prover energia)
Pinos de entrada analógicos	6	16
Saída de corrente para pinos de E/S	40mA	40mA
Corrente fornecida pelo Pino de 3.3V	50mA	50mA
Memória Flash	32KB (0,5KB é utilizado pelo bootloader)	128KB (4KB é utilizado pelo bootloader)
SRAM	2KB	8KB
EEPROM	1KB	4KB
Velocidade do clock	16MHz	16MHz

Os nós da plataforma Arduino podem operar em diferentes modos, e cada modo possui seu próprio consumo de energia. O consumo total de energia em um nó pode ser definido como a soma dos valores de energia gastos com sensoriamento, transmissão, recepção e processamento de dados. Os modos de operação dos nós podem ser: (i) inativos ou em modo *sleep*; e (ii) ativos, quando estão exercendo alguma tarefa solicitada por uma aplicação cliente.

A plataforma Arduino oferece o conceito de *shields*, que são placas que podem ser adicionadas ao nó para aumentar sua funcionalidade. Existem *shields* para conexão do Arduino com módulos *Bluetooth*, *Ethernet*, entre outros. A placa utilizada neste trabalho, denominada *XBee Shield*, permite a interligação do Arduino ao rádio *XBee*[84]. O Arduino com o *XBee Shield* é ilustrado na Figura 5.2.

Figura 5.2: Arduino UNO com a placa *XBee Shield*

O PRISMA possui um conjunto de bibliotecas desenvolvidas, representando duas funcionalidades do middleware: descoberta de serviços e uma biblioteca para tratamento de mensagens. Fora as bibliotecas desenvolvidas, as seguintes bibliotecas são utilizadas: *Xbee-Arduino* [85], responsável por se comunicar com o rádio XBEE e *PString* [86] para facilitar o uso das funções do modo API e reduzir a complexidade do tratamento das mensagens e *Narcoleptic* [87] que é utilizada para alterar o modo de operação do processador do nó de ativo para *sleep* e desta forma poupar energia.

5.1.1 Rádio XBee

O rádio *XBee* trabalha com dois modos de operação de transmissão e recepção de dados. No primeiro modo, *Transparent Operation* ou *AT*, os dados são enviados e recebidos diretamente pela porta serial, tendo uma interface simples e sendo mais fácil o desenvolvimento de aplicações, bastando à aplicação se conectar a porta serial do módulo e enviar os dados utilizando comandos *AT*. Este modo, apesar de simples, não é escalável para enviar dados para múltiplos destinatários e também não permite o envio de configurações remotas de módulos.

O segundo modo, que é utilizado neste trabalho, é o modo *Application Programming Interface* (*API*), o qual utiliza o padrão 802.15.4 para o envio e recepção de quadro de dados. Através destes quadros o modo API especifica: (i) o quadro para envio de comandos de configuração de rádio; (ii) o quadro com as respostas destes comandos de configuração; e (iii) o quadro contendo informações sobre o estado de funcionamento do rádio. Os quadros utilizados pelo modo API podem ser observados nas **Erro! Fonte de referência não encontrada.**, **Erro! Fonte de referência não encontrada.** e **Erro! Fonte de referência não encontrada.**. Tal modo de operação propicia um aumento na escalabilidade da rede já que através deste modo podemos interagir com cada nó da rede ou apenas parte da rede separadamente. Comandos AT também podem ser enviados e recebidos através do modo API, permitindo a coexistência dos dois modos em uma rede. A estrutura do quadro de dados é descrita a seguir e as aplicações que utilizem o modo API devem estar em conformidade com esta estrutura.

Assim como os nós, os rádios também podem operar em diferentes modos de operação. No caso do PRISMA que utiliza o rádio *XBee* do Arduino existem cinco possíveis modos de operação: (i) *Idle*, (ii) *Transmit,* (iii) *Receive,* (iv) *Command* e (v) *Sleep.* Cada um

destes modos de operação consome uma quantidade diferente de energia. Em nosso exemplo do *XBee*, ele permanece, por padrão, no modo *Idle* quando não está em operação de envio e recebimento de dados. A partir desse modo, ele poderá mudar para os modos *Transmit*, quando necessita transmitir dados; *Receive*, quando um pacote de dados válido é recebido pela antena; *Command*, quando está em modo para receber instruções de controle e configuração do rádio e *Sleep*, em que o rádio fica em estado de dormência. Porém o modo de operação a ser observado, quando tratando de economia de energia, é o modo *Sleep*. Este modo pode ser configurado no rádio *XBee* alterando o parâmetro SM (parâmetro que especifica o modo de operação *Sleep Mode)*. No anexo A estão os detalhes de cada modo do parâmetro SM disponível no rádio XBee.

5.2 Implantação

A implantação dos componentes de software que compõem a arquitetura do PRISMA nos dispositivos físicos que compõe a RSSF deve ser realizada conforme descrito nos diagramas UML de implantação: Figura 5.3 (*Gateway*), Figura 5.4 (*Cluster Head*) e Figura 5.5 (Nó Sensor). O subsistema *Gateway* contém os componentes: Servidor Web, Componente de Controle de Aplicações, Componente de Decisão, Componente de Publicação e Descoberta, Componente de Eventos, Componente de Comunicação, Componente de Controle de Topologia e Componente Monitor de Contexto. O subsistema *Cluster Head* contém os mesmos componentes, com exceção dos componentes Servidor *Web* e Componente de Controle de Aplicações. Já o subsistema Nó Sensor possui os mesmos componentes do subsistema *Cluster Head* com a adição do Componente de Aquisição de Dados. Cabe lembrar que, embora prevista na arquitetura e descrita nos diagramas de implantação, a funcionalidade de controle de topologia não foi implementada na versão atual do protótipo do PRISMA. Portanto, os módulos de software que implementam tal funcionalidade não foram construídos. A implementação do PRISMA para a plataforma Arduino está disponível em: http://146.164.247.214/wordpress/projects/prisma/.

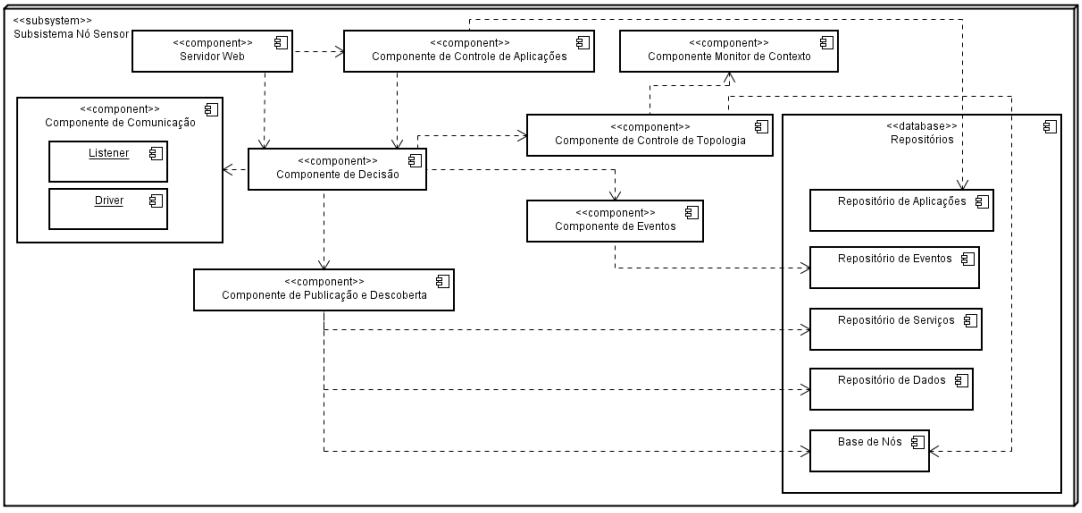

Figura 5.3: Diagrama de implantação do subsistema *Gateway*

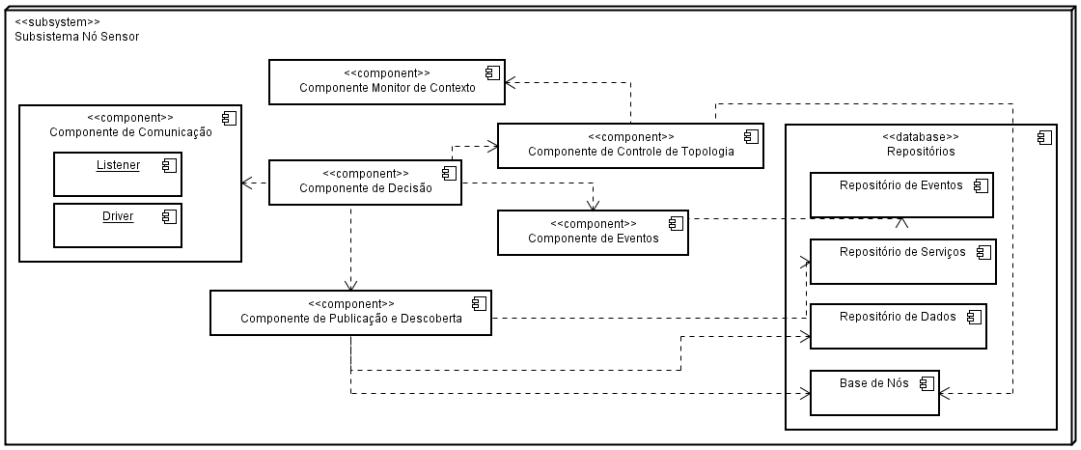

Figura 5.4: Diagrama de implantação do subsistema *Cluster Head*

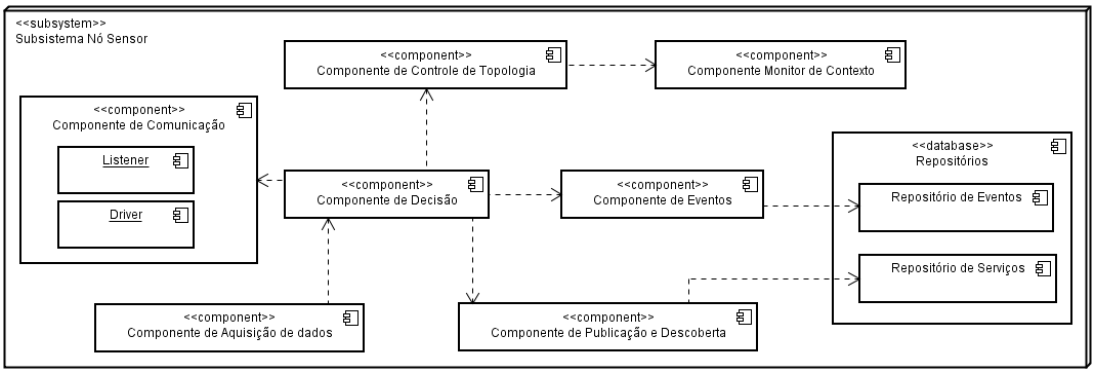

Figura 5.5: Digrama de implantação do subsistema Nó Sensor

As principais tecnologias utilizadas na implementação do PRISMA podem ser vistas na Figura 5.6:

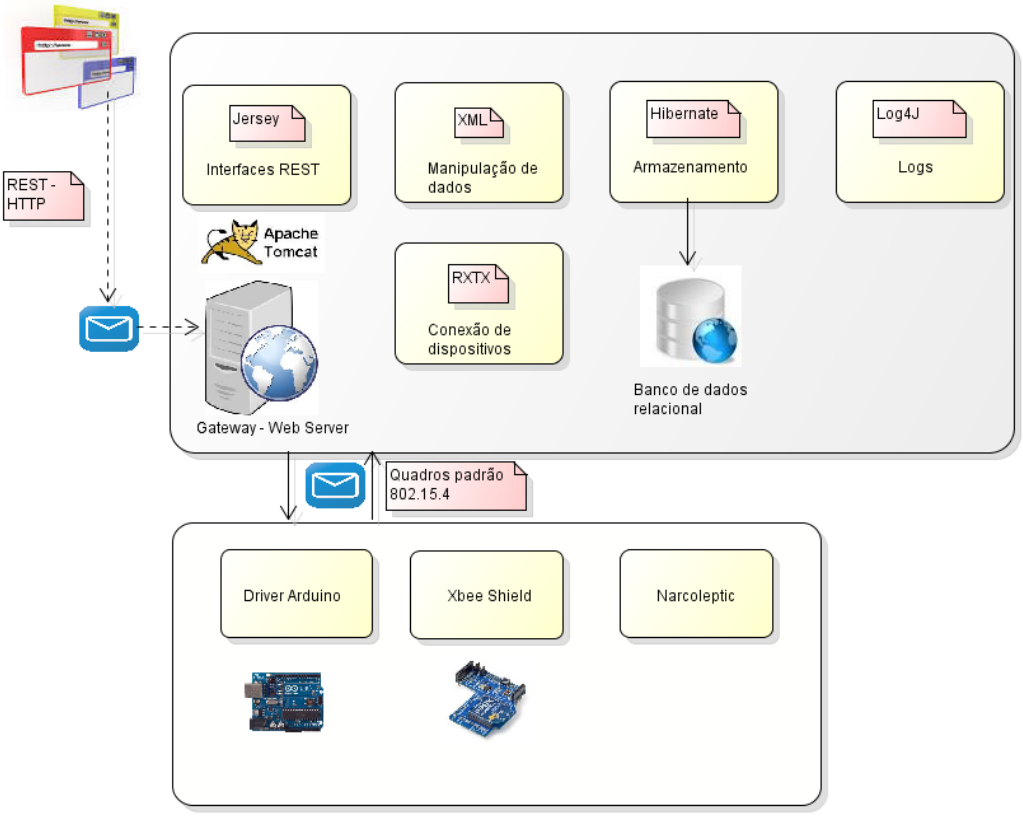

Figura 5.6: Principais tecnologias utilizadas no desenvolvimento do PRISMA

5.3 Interfaces gráficas

Para prover um maior grau de abstração e facilidade aos clientes, também foi desenvolvida uma página Web para a criação de novas aplicações e submissão a RSSF, para verificar estatísticas sobre aplicações em execução e para exportar os resultados obtidos para outros formatos, como por exemplo, XML ou CSV.

Ao entrar na página o usuário encontra uma página inicial para efetuar seu *login* ou criar uma nova conta, a qual está ilustrada na Figura 5.7.

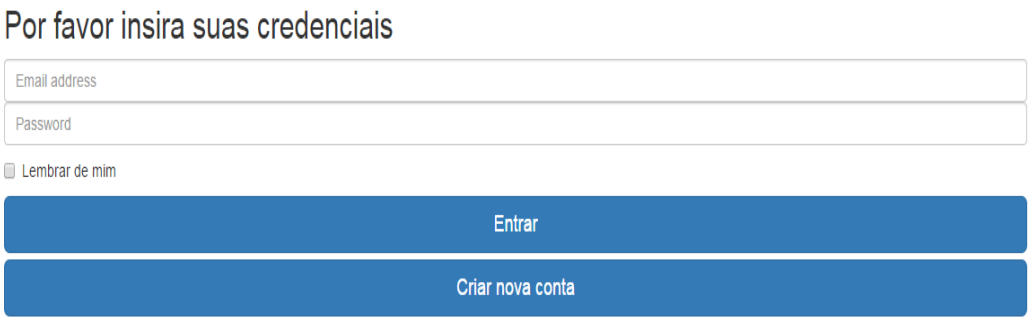

Figura 5.7: Página de login

Ao entrar com suas credenciais o usuário passa a ter acesso ao *middleware* e pode solicitar a criação de uma nova aplicação. Os dados solicitados são pela interface web são divididos em três categorias: (i) dados gerais da aplicação, (ii) fenômenos a serem monitorados periodicamente e (iii) eventos a serem monitorados. Na primeira categoria é solicitada a taxa de coleta e tempo de vida da aplicação. Na segunda categoria são solicitados os dados sobre os fenômenos a serem monitorados periodicamente: (i) fenômeno físico a ser monitorado e (ii) área geográfica que deve ser monitorada. E a terceira categoria solicita dados sobre os eventos a serem monitorados na rede, são eles: (i) fenômeno físico a ser monitorado, (ii) limite inferior, (iii) operador utilizado sobre os limites, (iv) limite superior e (v) área geográfica a ser monitorada. A página de criação pode ser vista na Figura 5.8.

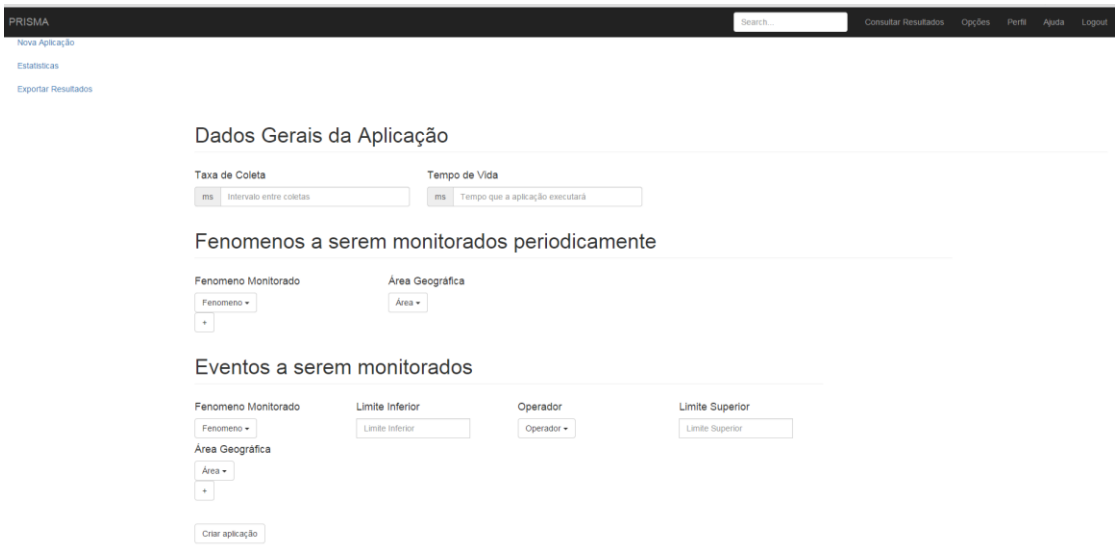

Figura 5.8: Página para criação de novas aplicações

O objetivo desta outra interface com os usuários é permitir que usuários que não tenham conhecimento de desenvolvimento possam coletar dados de redes de sensores sem fio. Isto é possível aumentando o nível de abstração na interação com a rede, pois como vemos na Figura 5.8, o usuário deve apenas preencher os campos sobre os quais ele pode exercer algum controle. Como os fenômenos possíveis de serem monitorados e as áreas geográficas em que o PRISMA está agindo não dependem do usuário específico utilizando a rede, apenas um menu *dropdown* é fornecido. Com isto o usuário fica limitado às opções que são possíveis de serem atendidas na RSSF. Estas opções são preenchidas de acordo com as entradas do banco de dados que armazena os nós presentes na rede.

Apesar de a interface ser mais simples que o manuseio do arquivo XML para criar uma nova aplicação, através desta interface não é possível receber os dados assincronamente. A interface web fornece uma visão dos dados em forma de tabela. Além disto, os dados podem ser exportados em formato CSV ou XML, conforme mostrado na Figura 5.9 (menu lateral).

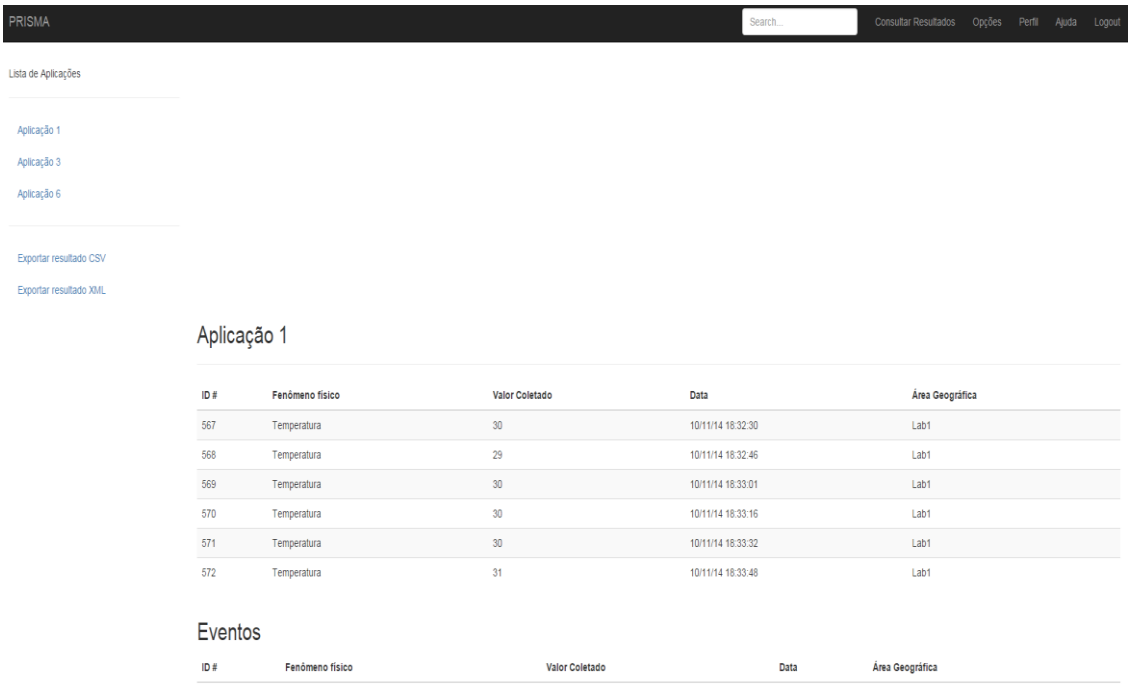

Figura 5.9: Resultados apresentados na interface web

5.4 Lições aprendidas na Implementação do PRISMA

A primeira dificuldade encontrada ao trabalhar com a plataforma Arduino foi a carência de documentação disponível para consulta. Tal dificuldade ocorre particularmente no contexto de desenvolvimento de trabalhos na área de redes de sensores sem fio, já que o uso mais comum da plataforma Arduino é em trabalhos de robótica, automação, instrumentação de ambientes, emulação ou implementação de sistemas interativos e até mesmo em sistemas musicais. O site oficial da plataforma apresenta apenas conceitos básicos sobre a linguagem utilizada e pequenos exemplos de uso bastante simples. A parte referente à criação de bibliotecas e conhecimentos mais avançados das possibilidades da linguagem foram adquiridos consultando fóruns da própria comunidade do site oficial do Arduino. Bons tutorias e vídeos explicativos foram compartilhados pelos usuários e puderam ser consultados. Juntamente com esta descoberta de novas fontes para consulta, também foi recomendado abandonar a IDE de desenvolvimento padrão e que fosse utilizado o editor de texto *Sublime* [79] com um *plugin* que possibilitava adicionar as funcionalidades da IDE normal, mas que fornecia informações de debug mais precisas e que são de fato úteis para o programador/desenvolvedor.

Depois dos desafios iniciais de aprendizado básico da plataforma, apareceram dificuldades em manipular os dispositivos, inicialmente no tocante a capacidade de monitorar o nível energético de cada nó. Neste caso, somente foi possível coletar o nível energético dos nós após encontrar uma lista de discussão que foi respondida por um dos colaboradores diretos da plataforma. Ele explicou como coletar a voltagem sendo suprida para o dispositivo em dado momento. A operação é bem complicada e envolve manipulação dos registradores internos do dispositivo para obtermos este valor. A função pode ser vista na Figura 5.10:

```
//Função responsável por coletar a voltagens sendo suprida ao arduino
long readVcc() {
  // Read 1.1V reference against AVcc
  // set the reference to Vcc and the measurement to the internal 1.1V reference
#if defined(__AVR_ATmega32U4__) || defined(__AVR_ATmega1280__) || defined(__AVR_ATmega2560__)
  ADMUX = _BV(REFS0) | _BV(MUX4) | _BV(MUX3) | _BV(MUX2) | _BV(MUX1);
#elif defined (__AVR_ATtiny24__) || defined(__AVR_ATtiny44__) || defined(__AVR_ATtiny84__)
  ADMUX = _BV(MUX5) | _BV(MUX0);
#elif defined (__AVR_ATtiny25__) || defined(__AVR_ATtiny45__) || defined(__AVR_ATtiny85__)
  ADMUX = _BV(MUX3) | _BV(MUX2);
#else
  ADMUX = _BV(REFS0) | _BV(MUX3) | _BV(MUX2) | _BV(MUX1);
#endif

  delay(2); // Wait for Vref to settle
  ADCSRA |= _BV(ADSC); // Start conversion
  while (bit_is_set(ADCSRA,ADSC)); // measuring

  uint8_t low  = ADCL; // must read ADCL first - it then locks ADCH
  uint8_t high = ADCH; // unlocks both

  long result = (high<<8) | low;

  result = 1125300L / result; // Calculate Vcc (in mV); 1125300 = 1.1*1023*1000
  return result; // Vcc in millivolts
}
```

Figura 5.10: Função para coleta de voltagem fornecida para o Arduino.

Após esta dificuldade ser superada, surgiu outra dificuldade relacionada à manipulação do rádio dos nós sensores. Como a plataforma utiliza o conceito de *Shields*, utilizamos o *XBee Shield*. Todo rádio XBee vem de fábrica configurado com o modo AT (descrito no início deste Capítulo) e não é provido nenhum guia sobre como alterar o modo de operação. Os vários modos de operação existentes no *XBee Shield* foram descobertos ao realizarmos uma extensa pesquisa sobre esse assunto na Internet. Todas as informações relacionadas desde ao programa necessário para manipulação dos parâmetros do rádio (X-CTU [88]) até a descrição destes parâmetros foram encontradas em fóruns e comunidades da plataforma e em pequenos tutoriais espalhados pela Internet. Mais detalhes foram conseguidos ao procurar no *datasheet* do rádio XBee que explicava cada parâmetro que poderia ser configurado nos rádios. Ao adicionarmos este conhecimento ao conhecimento adquirido nos tutoriais encontrados, foi possível a adaptação do ciclo de trabalho dos nós sensores, bem como a alteração do modo de operação em tempo de execução utilizando em conjunto o modo de transmissão AT e o modo API. Mesmo obtendo estas informações, o processo de alteração dos parâmetros não é 100% a prova de falhas, e ao alterarmos tais parâmetros em testes iniciais realizados nos dispositivos reais, dois rádios de comunicação foram perdidos. Não foi possível conseguir evidências suficientes para determinar se o problema foi causado pela aplicação de manipulação (X-CTU) ou por defeito de fábrica nos rádios de comunicação. Pelo que foi deduzido e encontrado em pesquisas na Internet,

ocorreu uma falha na atualização de *firmware,* o que não permitia que o X-CTU se comunicasse mais com o rádio. Ao pesquisar sobre detalhes do problema ocorrido e pelos relatos encontrados nos fóruns, constatou-se que este é um problema que ocorre com certa frequência e estes dispositivos em falha até possuem um termo associado a eles, dizem que o rádio se tornou "*bricked*".

A principal lição aprendida está relacionada diretamente à redução do consumo de energia pelos Arduinos. Por ser uma plataforma de prototipagem que é mais frequentemente utilizada em áreas de robótica e automação, o consumo de energia apesar de ser importante, não é vital como no caso de RSSFs. Abaixo está o resultado de alguns testes com o consumo de energia do Arduino, na Figura 5.11:

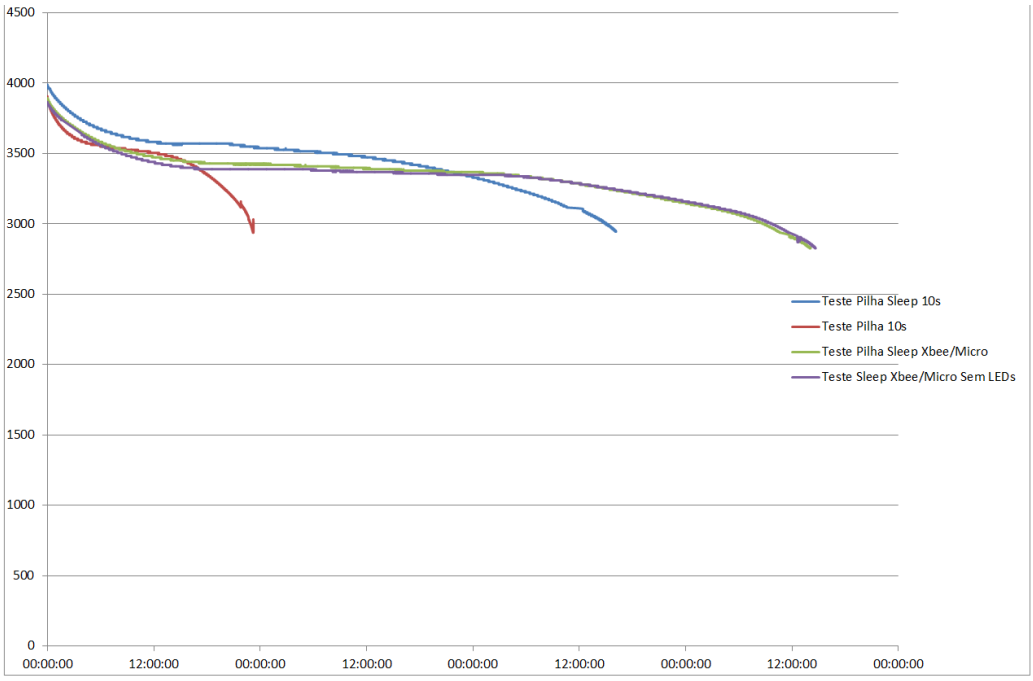

Figura 5.11: Consumo de energia do Arduino em diversos testes

Nos testes, podemos ver que ao configurarmos o Arduino para enviar mensagens de 10 em 10 segundos sem entrar em modo *sleep* a bateria durou apenas 1 dia. Tal duração pode ser considerada extremamente baixa pelos requisitos de monitoramento contínuo de aplicações de RSSF como, por exemplo, monitoramento ambiental, que podem requerer duração de meses ou até anos. Ao introduzirmos o modo *sleep* neste mesmo cenário notamos que a bateria durou aproximadamente 36 horas. Depois, ao utilizarmos juntamente a biblioteca *Narcoleptic* [87], podemos notar que a bateria ganhou uma sobrevida de mais aproximadamente 24 horas. Com esta biblioteca podemos alterar o modo de operação do microcontrolador do Arduino, do mesmo modo como feito com o rádio de comunicação,

alteramos o modo de operação para "*sleep*" e, com isto, geramos uma maior economia de energia. Normalmente o processador entraria em modo *Idle* que apesar de consumir menos energia ainda é um grande desperdício de energia quando comparamos com a energia consumida em modo "*sleep"*. Isso se deve ao fato de que mesmo que o rádio "durma" e reduza o consumo de energia o processador do Arduino continuava em operação consumindo a bateria. Com a adoção desta biblioteca foi possível ajustar o *timer* e deixar tanto o rádio de comunicação quanto o processador em modo *sleep* juntos. Os melhores resultados obtidos estão diretamente ligados a esta biblioteca, já que através de seu uso, um grande consumo de energia foi reduzido. Mesmo assim podemos notar que o tempo de vida da bateria está muito longe do esperado para RSSFs.

Ao pesquisar sobre o assunto no site oficial e em outros sites onde já havíamos encontrado boas informações sobre Arduinos, encontramos um grupo de pesquisadores discutindo este tema e eles alertaram que os LEDs da placa consumiam uma grande quantidade de energia. Então novos testes foram realizados, desta vez desligando LEDs de sinalização. Foi possível notar um pequeno ganho, entre 1 e 2 horas a mais de tempo de vida, mas não foram todos os LEDs da placa que foram desligados, então este resultado pode ser melhorado.

Além disto, foi alertado que existe um grande desperdício de energia por parte do controlador de voltagem *onboard* da plataforma. No gráfico de calor a seguir podemos ver que o regulador de voltagem (mais a direita) é responsável por uma grande dissipação de calor, o que pode ser traduzido em energia desperdiçada, Figura 5.12:

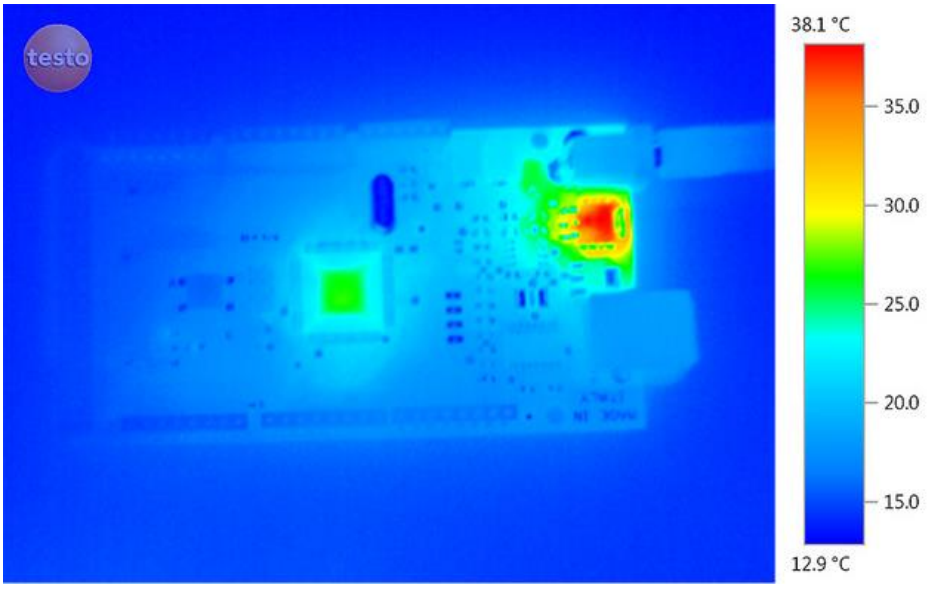

Figura 5.12: Gráfico de calor do Arduino MEGA

Mesmo encontrando todas estas dificuldades, a plataforma se mostrou promissora para a área de redes de sensores sem fio, já que a maioria das dificuldades se originou pela documentação pobre fornecida no site oficial da plataforma. Acreditamos ser natural que, quanto mais a plataforma for empregada em projetos fora do conceito de DYI (do inglês, *Do it yourself*), mais a comunidade criará novas documentações e bibliotecas para facilitar o uso da plataforma.

Variações da plataforma já estão sendo criadas, até mesmo visando domínios específicos de atuação, como o Arduino Lilypad que tem como alvo principal ser empregado em vestimentas. Com isto, podemos ver que a plataforma está expandindo e atingindo novos domínios. Como até a data da pesquisa deste trabalho não encontramos nenhuma plataforma de *middleware* que utilizasse Arduino no domínio de RSSF, se a plataforma continuar a ser empregada talvez possamos presenciar a criação de uma variação que se proponha a superar suas limitações.

Dentre as limitações encontradas quanto à utilização da plataforma em RSSFs, uma das principais, e que a meu ver necessita ser resolvida para que a plataforma possa ser amplamente utilizada, é a limitação energética. Na Figura 5.12 podemos observar que um componente de *hardware* da plataforma é responsável por um gasto energético inviável para o contexto de RSSF. Os nós destas redes devem permanecer ativos por meses, e até mesmo por anos, porém pelos resultados obtidos nos testes exibidos na Figura 5.11, o máximo de tempo de vida obtido utilizando estes modelos foi de cerca de 4 dias.

Outra limitação, que levou a redução do escopo da solução proposta devido à decisão de realizar sua implantação no Arduino, consiste na inexistência de bibliotecas que implementem algoritmos de roteamento, mecanismos mais sofisticados de controle de acesso ao meio, ou até mesmo bibliotecas que forneçam funcionalidades gerais de uma rede de sensores sem fio (como agregação de dados, monitoramento de contexto do nó, por exemplo). Neste trabalho, foi dado início ao desenvolvimento de tais bibliotecas através da implementação de funcionalidades de monitoramento de contexto, atualmente somente o nível energético do nó. Bibliotecas que facilitem esta manipulação de elementos do *hardware* seriam essenciais para o início de uma maior exploração da plataforma no domínio de RSSF.

6 Avaliação

Este Capítulo descreve a avaliação realizada com o principal intuito de validar a arquitetura do PRISMA e para determinar se ele atinge os objetivos descritos na Seção 6.2. Além disso, a avaliação procurou mensurar a complexidade e os benefícios ao se utilizar o PRISMA. A seguir será dada uma breve descrição da metodologia GQM, usada para nortear a avalição conduzida e, em seguida, serão definidos os objetivos, questões e métricas a serem utilizados na avaliação, segundo a metodologia GQM.

Em todos os experimentos realizados com o PRISMA, a RSSF foi composta de Arduinos UNO para os nós sensores e Arduinos MEGA para os *cluster heads* (conforme descrito no Capítulo 5 e ilustrado na Figura 6.1). Estes nós foram alimentados com quatro pilhas do tipo AA (1,5V, 1500 mAh) que fornecem aproximadamente 32kJ de energia. Os experimentos foram realizados em nós reais no laboratório de computação Ubíqua do PPGI-UFRJ.

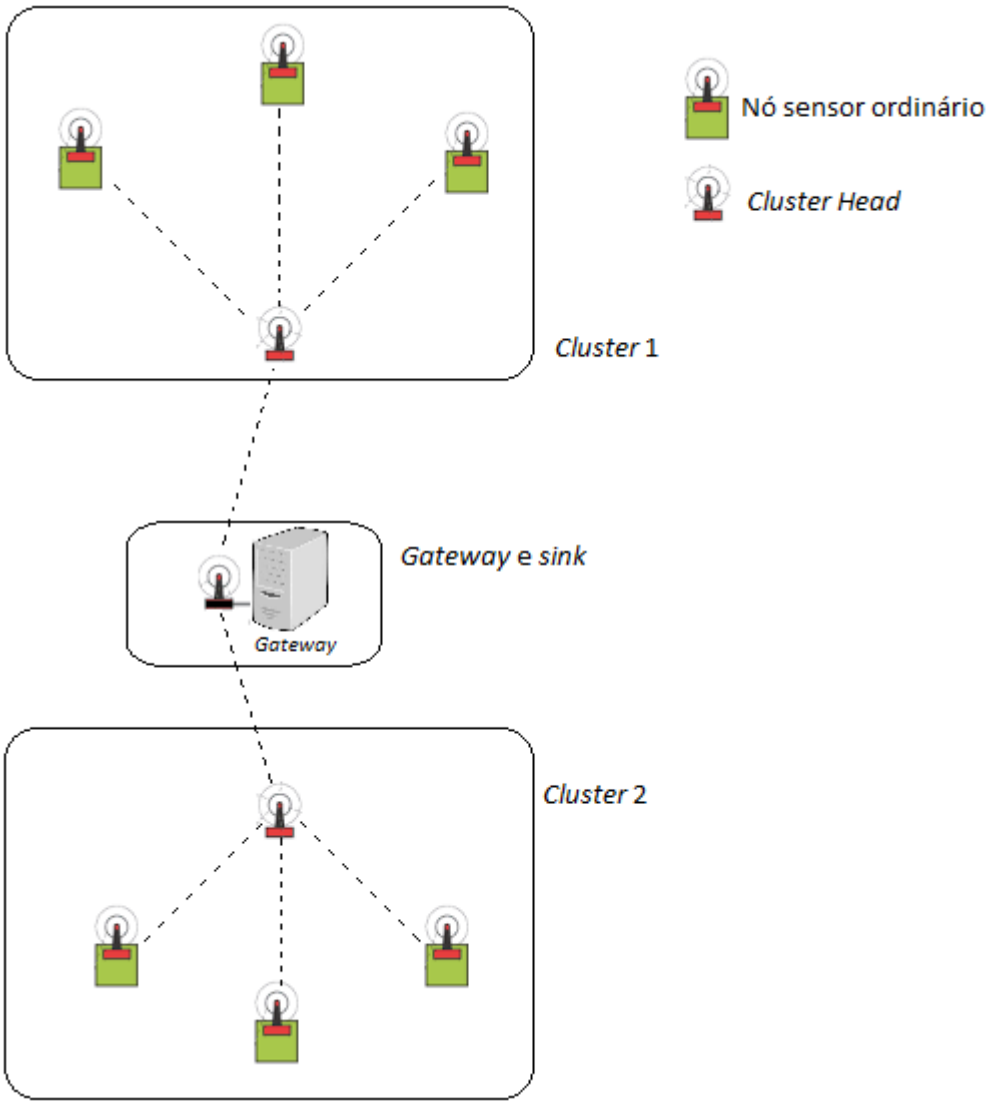

Figura 6.1: Diagrama esquemático da topologia de rede utilizado no experimento

6.1 Abordagem Goal/Question/Metric (GQM)

Proposta em 1988, a metodologia GQM [89] é uma abordagem orientada a metas para a medição de produtos e processos de *software*. É executada em duas partes: a primeira contempla a definição *top-down* de um programa de mensuração e a segunda parte contempla a análise e interpretação *bottom-up* dos dados coletados. A metodologia GQM se baseia no princípio de que a medição deve ser orientada a metas, de forma que a coleta dos dados deve ser baseada nos fatores de qualidade que influenciam direta ou indiretamente as metas definidas.

Os programas de mensuração baseados em GQM devem ser planejados e executados de acordo com os seguintes princípios: (i) a tarefa de análise a ser executada precisa ser

especificada precisamente e explicitamente através de uma meta; (ii) as medidas precisam ser derivadas de forma *top-down* baseadas nas metas e questões definidas; uma estrutura de metas e questões não pode ser adaptada de forma retroativa a um conjunto de medidas existente; (iii) cada medida precisa ter um fundamento lógico subjacente que é documentado explicitamente; tal fundamento lógico é usado para justificar a coleta dos dados e para guiar a análise e interpretação; (iv) os dados que são coletados com respeito às medidas definidas precisam ser interpretados de uma forma *bottom-up* no contexto das metas e das questões; e (v) as pessoas que vão utilizar os resultados do programa de mensuração precisam ser envolvidas profundamente na definição e interpretação desse programa. Elas são os reais peritos com respeito ao objeto e ao enfoque de qualidade investigado no programa de mensuração, portanto proveem interpretações válidas no ambiente especifico.

6.2 Avaliações do PRISMA baseadas na abordagem GQM

No Capítulo 3.3 já foi feita uma breve avaliação sobre como o PRISMA oferece as funcionalidades quando comparado com outro *middleware*. Neste capítulo é dada uma ênfase na análise quantitativa destas mesmas funcionalidades.

Com o objetivo de verificar se o PRISMA atende à funcionalidade de prover uma abstração de programação para RSSF, e se ele é escalável em termos de requisições de clientes, foram elaboradas **metas** primárias que são a seguir refinadas em um conjunto de **questões** as quais são respondidas através de **métricas**. Estas fornecem embasamento para as respostas obtidas.

O seguinte conjunto de metas (*goals*) foi definido:

G1: Analisar o PRISMA com o propósito de avaliar sua eficácia com respeito a atender a funcionalidade de prover abstrações de programação em um *middleware* para RSSF no contexto de desenvolvimento e implementação de aplicações;

G2: Analisar o PRISMA com o propósito de avaliar a sua escalabilidade em termos de suporte a um número crescente de requisições de aplicações.

Este conjunto de metas foi elaborado considerando os recursos disponíveis no laboratório e seu espaço físico para a realização dos testes. Devido ao número de unidades de sensoriamento e Arduinos UNO (sensores ordinários) disponíveis no laboratório não foi

possível uma avaliação de escalabilidade em termos de números de nós. Simuladores foram procurados, mas como a plataforma é recente nenhum simulador para montar uma topologia de rede foi encontrado, apenas simuladores específicos para testes de circuitos lógicos utilizando Arduinos.

Avaliamos a escalabilidade em termos de atendimento as requisições dos clientes. Esta estratégia foi escolhida considerando as tendências da área de redes de sensores sem fio. Como citado no Capítulo 1, o número de infraestruturas físicas de RSSF na mesma área de cobertura tende a aumentar consideravelmente, e há uma tendência em direção a se compartilhar e integrar os dados produzidos por tais redes entre diferentes aplicações. Além disso, estas infraestruturas irão comportar um grande número de aplicações simultâneas. Essa tendência de compartilhamento de uma mesma RSSF entre diferentes aplicações pode gerar um grande número de requisições em um pequeno intervalo de tempo, por isto decidimos por avaliar a escalabilidade em termos de requisições de clientes.

Estas metas foram refinadas em seis questões. As questões Q1 e Q2 são relacionadas meta G1, as questões Q3 a Q5 são relacionadas a G2. As questões são as seguintes:

- Q1: Quão custoso é criar uma aplicação utilizando o PRISMA, em termos de número de linhas de código?
- Q2: Quão custoso é criar uma aplicação utilizando o PRISMA, em termos de tempo de desenvolvimento?
- Q3: PRISMA escala bem em termos de servir a um grande número de requisições de novas aplicações?
- Q4: Qual é o *overhead* introduzido pelo PRISMA em termos de mensagens de controle e configuração?
- Q5: Qual é o *overhead* introduzido pelo PRISMA em termos de memória RAM para sua operação nos nós sensores?

As métricas a seguir foram definidas para responder às questões consideradas na avaliação. Cada métrica é denotada por M_{ij}, onde $_i$ corresponde ao identificador de questão, e $_j$ corresponde ao contador de métricas, quando existe mais de uma métrica para uma única questão.

O **número de linhas de código** (**M_{11}**) é a métrica utilizada para avaliar quão simples é criar uma aplicação de sensoriamento utilizando o PRISMA através da abstração provida por sua interface REST de criação de aplicações. Para computarmos esta métrica, coletamos o número de linhas de código necessárias para criar uma aplicação: (i) utilizando a abordagem provida pelo PRISMA através da interface REST de criação de aplicações e (ii) utilizando a programação diretamente no Arduino com a *Arduino programming language.*

O **tempo necessário para se criar uma aplicação** (**M_{12}**) é a métrica utilizada para avaliar quão simples é criar uma aplicação de sensoriamento utilizando o PRISMA através da abstração provida por sua interface Web. Para computarmos esta métrica, coletamos o tempo necessário para criar uma nova aplicação: (i) utilizando a abordagem provida pelo PRISMA através da interface Web e (ii) utilizando a programação diretamente no Arduino com a *Arduino programming language.* O tempo foi coletado através de um cronometro externo.

O **número máximo de requisições suportadas** (**M_{21}**) foi a métrica utilizada para verificar se o PRISMA é escalável com respeito a sua abordagem de abstração de programação através de interfaces REST.

O **tempo necessário para criar uma nova aplicação** quando o servidor *web* do PRISMA está sobre uma grande carga de trabalho (**M_{22}**) foi a métrica utilizada para verificar o tempo de resposta do *Gateway* quando um novo arquivo de configuração é enviado através da interface de criação de aplicação REST em uma situação em que muitas requisições estão sendo feitas simultaneamente. Requisições são geradas de forma crescente para as interfaces REST do *middleware* para determinar qual número máximo de requisições suportadas simultaneamente e o atraso esperado na criação de uma nova aplicação (tempo de resposta do *gateway*) ao variarmos o número de requisitos das aplicações e com isso gerando mensagens de tamanhos diferentes a serem enviadas à RSSF.

A métrica **tamanho das mensagens de controle transmitidas** dentro da RSSF (**M_{31}**) foi utilizada para avaliar o *overhead* introduzido pelas mensagens de controle/configuração que trafegam pela RSSF. Uma mensagem deve ser enviada a cada cluster para configuração da aplicação e o *Cluster Head* envia uma mensagem para cada nó que deve ser configurado com os parâmetros a serem configurados para cada tarefa de sensoriamento necessária. Ao

considerarmos os tamanhos de mensagens fixos (uma mensagem para cada tarefa a ser configurada) podemos calcular o número de bytes transmitidos em cada cenário.

A métrica de **consumo de memória RAM** gerado pelo PRISMA (**M_{41}**) para executar uma aplicação de sensoriamento foi utilizada para avaliar o overhead introduzido pelo PRISMA em termos de consumo de memória dos nós sensores, e a viabilidade de executar o middleware no ambiente restrito de uma RSSF. Esta métrica foi obtida ao compilarmos e enviarmos o código base do PRISMA para os nós.

6.2.1 Metodologia de avaliação e cenários

Para coletar os dados necessários para responder às questões, uma avaliação experimental foi realizada. No experimento, a rede foi planejada para que existam dois *clusters*, cada um destes com um conjunto de 3 nós sensores com diferentes capacidades de sensoriamento.

Uma topologia circular foi organizada para que o gateway se localizasse no centro, e, com isto, os nós sensores e nós *cluster heads* se mantivessem equidistantes do centro, reduzindo o fator da distância no atraso e no consumo de energia entre os *clusters*.

Os nós *Cluster Heads* foram pré-configurados nos nós sensores e a potência de transmissão foi mantida a mesma para todos os nós, além das rotas de comunicação também estarem definidas estaticamente. Todos os nós iniciam com o mesmo ciclo de trabalho e este ciclo somente será alterado se uma aplicação for atribuída a este nó. Caso nenhuma aplicação seja atribuída a este nó, ele permanecerá com ciclo de trabalho reduzido.

Quatro aplicações clientes foram desenvolvidas, duas para a solicitação de dados periódicos e outras duas para solicitação de dados baseados em eventos, a fim de coletar as métricas especificadas. Cada aplicação representa um cenário e todos os cenários tem como área geográfica alvo o laboratório “Lab1”. No primeiro cenário, a aplicação requisita uma amostra periódica de temperatura. As amostras deverão ser coletadas a cada 15 segundos e a aplicação deverá permanecer ativa por 5 minutos. O segundo cenário especifica uma aplicação que irá executar por 5 minutos e coletar dados periódicos de temperatura, umidade e luminosidade. Estas amostras serão coletadas a cada 15 segundos. O terceiro cenário especifica uma aplicação que irá executar por 10 minutos e coletar amostras de temperatura a cada 15 segundos. No entanto, neste caso o dado somente será enviado

pelos sensores se a temperatura exceder 30ºC. O quarto cenário especifica uma aplicação que executará por 10 minutos e coletará dados de temperatura e luminosidade a cada 15 segundos. Dados só serão enviados se a temperatura: exceder 40ºC ou estiver abaixo de 15ºC. O dado de luminosidade somente será enviado se a luminosidade da sala exceder 600 lumens.

Todas as aplicações são criadas através da interface de criação de aplicações que pode ser acessada na URL: http://<IPGateway>.com/prisma/aplicação/criar/. Uma mensagem POST é enviada contendo como corpo o XML a ser utilizado na criação da aplicação. O arquivo XML de cada cenário é descrito nas figuras: Figura 6.2, Figura 6.3, Figura 6.4 e Figura 6.5.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Aplicacao>
    <username>joserenato</username>
    <taxaColeta>15000</taxacoleta>
    <tempoDeVida>300000</tempoDeVida>
    <servicos>
        <fenomenoMonitorado>Temperatura</fenomenoMonitorado>
        <areaGeografica>Lab1</areaGeografica>
    </servicos>
</Aplicacao>
```

Figura 6.2: XML que representa a aplicação do cenário 1

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Aplicacao>
    <username>joserenato</username>
    <taxaColeta>15000</taxacoleta>
    <tempoDeVida>300000</tempoDeVida>
    <servicos>
        <fenomenoMonitorado>Temperatura</fenomenoMonitorado>
        <areaGeografica>Lab1</areaGeografica>
    </servicos>
    <servicos>
        <fenomenoMonitorado>Umidade</fenomenoMonitorado>
        <areaGeografica>Lab1</areaGeografica>
    </servicos>
    <servicos>
        <fenomenoMonitorado>Luminosidade</fenomenoMonitorado>
        <areaGeografica>Lab1</areaGeografica>
    </servicos>
</Aplicacao>
```

Figura 6.3: XML que representa a aplicação do cenário 2

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Aplicacao>
    <username>joserenato</username>
    <taxaColeta>15000</taxacoleta>
    <tempoDeVida>600000</tempoDeVida>
    <eventos>
        <servico>
            <fenomenoMonitorado>Temperatura</fenomenoMonitorado>
            <areaGeografica>Lab1</areaGeografica>
        </servico>
        <limiteSuperior>30</limiteSuperior>
        <limiteInferior>30</limiteInferior>
        <operadores>">"</operadores>
        <areaGeografica>Lab1</areaGeografica>
    </eventos>
</Aplicacao>
```

Figura 6.4: XML que representa a aplicação do cenário 3

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Aplicacao>
    <username>joserenato</username>
    <taxaColeta>15000</taxacoleta>
    <tempoDeVida>600000</tempoDeVida>
    <eventos>
        <servico>
            <fenomenoMonitorado>Temperatura</fenomenoMonitorado>
            <areaGeografica>Lab1</areaGeografica>
        </servico>
        <limiteSuperior>40</limiteSuperior>
        <limiteInferior>40</limiteInferior>
        <operadores>">"</operadores>
        <areaGeografica>Lab1</areaGeografica>
    </eventos>
    <eventos>
        <servico>
            <fenomenoMonitorado>Temperatura</fenomenoMonitorado>
            <areaGeografica>Lab1</areaGeografica>
        </servico>
        <limiteSuperior>15</limiteSuperior>
        <limiteInferior>15</limiteInferior>
        <operadores>"<"</operadores>
        <areaGeografica>Lab1</areaGeografica>
    </eventos>
    <eventos>
        <servico>
            <fenomenoMonitorado>Luminosidade</fenomenoMonitorado>
            <areaGeografica>Lab1</areaGeografica>
        </servico>
        <limiteSuperior>600</limiteSuperior>
        <limiteInferior>600</limiteInferior>
        <operadores>">"</operadores>
        <areaGeografica>Lab1</areaGeografica>
    </eventos>
</Aplicacao>
```

Figura 6.5: XML que representa a aplicação do cenário 4

O roteamento em todos os cenários é estaticamente configurado nos nós, pois, devido a restrições de tempo não foi possível explorarmos soluções de roteamento que pudessem ser utilizadas na plataforma Arduino. Cada nó sensor ordinário possui seu *Cluster Head* configurado e envia suas mensagens diretamente para o mesmo. Já no *Cluster Head* a mensagem é encaminhada diretamente ao *Gateway*, como nesta RSSF só existe um *sink*, este também foi configurado estaticamente.

Algumas características do rádio de comunicação são: adoção do protocolo MAC IEEE 802.15.4, conectividade ponto-multiponto, distância máxima de comunicação de 90 metros (com visada direta), taxa de transmissão 250kbps. Para o experimento o parâmetro de configuração SM (*Sleep Mode*) foi configurado para 1 – *Pin Hibernate*. Através da especificação deste parâmetro é possível configurar quando o rádio entra em estado de dormência e por quanto tempo ele permanece neste estado através da tensão fornecida no pino 9 do rádio. Inicialmente, antes de aplicações serem atribuídas aos nós, o rádio opera em modo ativo por 1 segundo e adormece por 4 segundos até que uma aplicação seja recebida. A partir deste momento o nó se adequa aos requisitos da aplicação e adormece nos intervalos em que não precisa coletar dados.

6.2.2 Análise dos resultados

Esta subseção discute os resultados obtidos ao extrairmos as métricas discutidas na seção anterior 6.2. Os cenários foram executados cada um 50 vezes e as requisições foram feitas de forma incremental, sendo que cada teste finalizava quando o tempo de resposta da requisição fosse superior a 800ms. Os resultados do número de requisições e tempo de resposta estão representados como a média obtida durante os experimentos e seu desvio padrão está listado logo ao lado.

Os resultados são mostrados na Tabela 3, a seguir:

Tabela 3: Resultados da avaliação do PRISMA

			Periódica				**Evento**			
Goal	**Question**	**# Serviços / Métrica**	**1**		**3**		**1**		**3**	
G1	**Q1**	**M_{11} (linhas)**	**A**	**P**	**A**	**P**	**A**	**P**	**A**	**P**
			15	10	37	18	20	16	50	34
	Q2	**M_{12} (minutos)**	A	P	A	P	A	P	A	P
			6:32	2:27	8:12	3:07	7:16	3:07	10:11	4:05
G2	**Q3**	**M_{21} (# requisições)**	1994 / 57 σ		947 / 53 σ		1093 / 68 σ		886 / 75 σ	
		M_{22}	111 ms / 114 σ		190 ms / 172 σ		186 ms / 153 σ		486 ms / 239 σ	
	Q4	**M_{31}**	74 Bytes		171 Bytes		82 Bytes		195 Bytes	
	Q5	**M_{41} (bytes)**	Uno	Mega	Uno	Mega	Uno	Mega	Uno	Mega
			13332	14918	13332	14918	13332	14918	13332	14918

Na Tabela 3, na linha correspondente ao objetivo G1, A se refere ao Arduino e P se refere ao PRISMA.

Com respeito ao objetivo G1 os resultados da métrica $\mathbf{M_{11}}$ indicam, como o esperado, que a abstração de programação provida pelo PRISMA (criação de aplicações através de um arquivo XML submetido através de interfaces REST) torna simples criar novas aplicações. Adicionalmente, a abstração de programação reduz o número de linhas necessárias para se criar uma nova aplicação quando comparado com o valor obtido na coluna A de Arduino. Isso ocorre uma vez que o cliente utiliza uma linguagem de alto-nível para especificar os requisitos. É importante notar que a diferença no número de linhas utilizado para especificar uma aplicação para o PRISMA e uma aplicação semelhante diretamente no Arduino aumenta de acordo com a complexidade da aplicação desejada. Ao compararmos os resultados podemos ver que no cenário 3 (terceira coluna da tabela) obtivemos o menor ganho, de apenas 20% e o maior ganho pode ser observado no cenário 2 (segunda coluna da tabela), onde observamos um ganho de 51% no número de linhas necessário para a aplicação.

Já os resultados da métrica $\mathbf{M_{12}}$ indicam, também como o esperado, que a abstração de programação provida pela interface Web torna a tarefa de criar uma aplicação mais simples até mesmo que criar o arquivo XML. Apesar do código fonte do Arduino se repetir e ser fácil implementar o monitoramento de novos fenômenos a serem detectados pela rede apenas copiando o código e o alterando em alguns pontos (o pino que deve ser consultado para obter o resultado, por exemplo), a facilidade que a interface Web traz é muito grande e seu uso mais amigável ao usuário final. Podemos ver que no caso mais simples (aplicação periódica monitorando um fenômeno físico, primeira coluna da tabela) o tempo necessário foi de apenas 37,5% do tempo de codificação no Arduino. Já no caso mais complexo, o tempo utilizando a interface Web foi 40% do tempo de codificação no Arduino. Apesar de não possibilitar a comunicação assíncrona, para um usuário que não possui conhecimentos de desenvolvedor esta abordagem é mais simples. Para a ilustração segue uma figura com a aplicação mais complexa sendo criada na interface web, Figura 6.6.

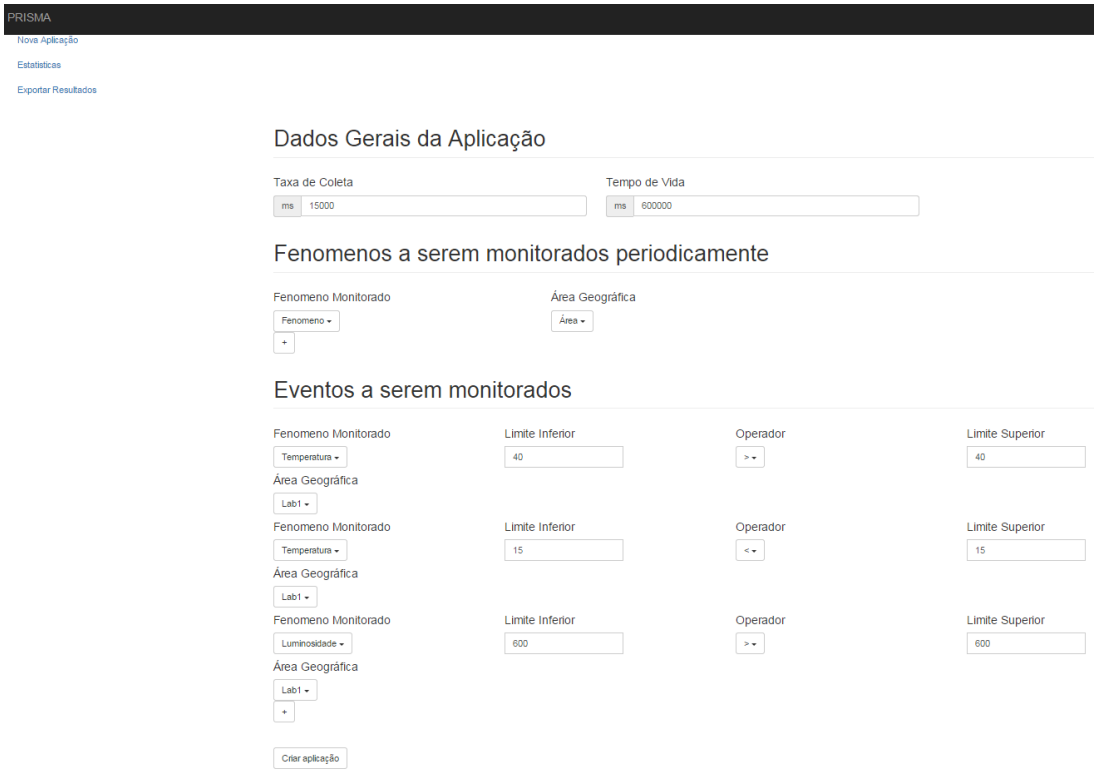

Figura 6.6: Aplicação baseada em eventos com 3 eventos a serem monitorados

Já referente ao objetivo G2, o resultado da métrica $\mathbf{M_{21}}$ indica o número máximo de requisições simultâneas antes do servidor Web do PRISMA parar de responder ou demostrar um tempo de resposta inaceitável. Consideramos qualquer tempo de resposta acima de 800 milissegundos como inaceitável. Podemos observar que no caso mais simples, uma aplicação que monitora penas um fenômeno físico periodicamente, o número de requisições simultâneas atendidas pelo PRISMA foi 82% maior que o caso mais simples de uma aplicação baseada em evento de apenas um fenômeno físico (terceira coluna da tabela). Este número é maior devido ao inferior número de dados a ser resgatado (quando comparado com os cenários 2 e 4, onde 3 dados são consultados) e devido ao menor número de consultas que devem ser realizadas ao banco de dados, já que não é necessário correlacionar a um evento específico.

O resultado da métrica $\mathbf{M_{22}}$ indica o tempo de resposta para implantar novas aplicações através da interface de criação do PRISMA. O tempo de resposta foi de 246 milissegundos na média. Este tempo de resposta é considerado pela literatura como próximo a imperceptível no ponto de vista de aplicações Web típicas, como descrito em [90]. Em [90], ainda é afirmado que não importa a qualidade dos dados apresentados, o usuário

pode desistir se o tempo de resposta for inaceitável em seu ponto de vista. Esta afirmação levou-nos a escolher um tempo mais rigorosos que o proposto pelo autor para garantirmos que o PRISMA está dentro de um padrão aceitável.

O tempo de resposta para a criação de aplicações baseadas em eventos é maior que o tempo de resposta para a criação de aplicações periódicas devido ao maior número de transações no banco de dados relacional necessárias.

As métricas $\mathbf{M_{31}}$ e $\mathbf{M_{41}}$ estão associadas ao *overhead* introduzido pelo PRISMA. A métrica $\mathbf{M_{31}}$ mede o número de bytes transmitidos pelos nós para criar uma nova aplicação em cada um dos cenários apresentados. Observamos que o número de bytes enviados para a rede com o objetivo de configurar uma nova aplicação aumenta de acordo com a complexidade da aplicação solicitada, 74 bytes na aplicação mais simples (cenário 1) e 195 bytes na aplicação mais complexa (cenário 4). Este aumento é relacionado ao número de mensagens que precisam ser trocadas para a configuração desta nova aplicação. No pior caso, uma mensagem para cada evento e/ou serviço é necessária. Isto acontece devido à necessidade de enviar a mensagem de configuração para o *Cluster Head* que irá encaminhar a parte desta configuração que é relevante para os nós que participarão da aplicação. Se ocorrer de cada serviço estar em um *Cluster Head* isolado iremos nos deparar com o caso descrito na tabela.

Já quanto à métrica $\mathbf{M_{41}}$, a tabela mostra o consumo de memória RAM utilizada no Arduino UNO e no Arduino MEGA. Podemos verificar que o consumo de memória RAM ao utilizarmos o PRISMA não se alterou entre os cenários e somente se alterou entre os modelos. A variação entre os modelos se deve ao tamanho do *bootloader* de cada modelo. É importante notar que os módulos atualmente implementados do PRISMA consomem menos de 50% da memória RAM disponível no Arduino UNO (32Kb), indicando que novos serviços e funcionalidades podem ser adicionados ao PRISMA sem esgotar os recursos típicos de um nó.

Analisando os resultados, podemos concluir que a complexidade da aplicação, em termos de tarefas de sensoriamento necessárias/eventos necessários, afeta o tamanho do XML necessário para criar uma aplicação e o número de bytes que devem ser transmitidos na RSSF para que esta nova aplicação seja implantada nos nós da rede. Entretanto, o PRISMA possibilita a criação de aplicações em tempo de execução. Isto evita que nós necessitem ser

removidos da área de instalação para a alteração de seu programa em execução para refletir as necessidades de uma nova aplicação, ou que novos nós devam ser introduzidos na rede para atender somente a esta nova aplicação. O número máximo de requisições suportadas, e o atraso percebido pelo cliente são primariamente afetados por características do *hardware* que foi utilizado para avaliar e implementar o *Gateway*. Quanto aos requisitos de memória dos dispositivos utilizados para a implantação do PRISMA, podemos concluir que são necessários exclusivamente para o PRISMA, aproximadamente 15KB, mas é necessário espaço adicional para armazenamento de variáveis e inclusão de novos serviços. Para oferecer uma liberdade ao desenvolvedor para criar novos serviços é recomendado que 32KB de memória RAM, como disponibiliza o Arduino, seja utilizado como requisito de memória para a implantação do PRISMA.

7 Conclusão e Trabalhos Futuros

Neste trabalho foi mostrado como uma plataforma de *middleware* pode auxiliar no desenvolvimento de aplicações para RSSF ao se adicionar camadas de abstração entre o usuário e a rede. O usuário não necessita saber de detalhes da plataforma do nó sensor para criar uma aplicação. Além disto, com o uso do paradigma de comunicação *publish-subscribe* o usuário recebe os dados desejados assincronamente. O PRISMA, um *middleware* orientado a recursos e *publish-subscribe* para redes de sensores sem fio foi desenvolvido. Os resultados obtidos demostram que o PRISMA pode ser implantado em nós reais e demostra que ele provê uma camada de abstração que torna mais simples o desenvolvimento de aplicações para a RSSF. Apesar das limitações apresentadas no Capítulo 5, um *middleware* ainda que limitado devido ao escopo do trabalho e limitações de tempo pode ser desenvolvido para a plataforma que é de fácil acesso, recente e *open-hardware*, ou seja variações desta plataforma podem ser criadas e o PRISMA pode ser facilmente adaptado a estes novos modelos.

Com o uso de interfaces REST, futuros desenvolvedores podem continuar a evoluir esta abordagem através de novos serviços ou clientes que possam consumir os dados gerados por uma RSSF que utiliza o PRISMA.

As principais contribuições deste trabalho são mostradas no Capítulo 5, mais especificamente na seção 5.4. Nesta seção mostramos as limitações encontradas durante o desenvolvimento do *middleware* ao utilizarmos a plataforma Arduino. Apesar de ser uma plataforma nova e promissora, em seu estado atual não recomendamos seu uso na área de RSSF, mas vale ressaltar que, como a plataforma está evoluindo rapidamente e novos concorrentes, por exemplo [91], estão surgindo com a mesma filosofia *open-hardware,* a plataforma poderá gerar um *hardware* específico para a área de RSSF.

Em síntese, quanto mais à plataforma for empregada em projetos de média e larga escala, maior será a visibilidade para os domínios empregados e com isto a possível expansão da plataforma visando atender necessidades exclusivas deste domínio. Pode-se citar como exemplo a construção de um *Shield* que forneça um melhor controle sobre o rádio ou até mesmo um rádio diferente que possibilite um melhor controle sobre sua

operação e um *hardware* que seja econômico em termos energéticos, sem LEDs para debug, com um tamanho reduzido para ser mais bem inserido no ambiente de monitoração. O que se pode concluir baseado nas dificuldades encontradas é que em seu estado atual a plataforma deixa a desejar quando aplicada no domínio de RSSF, principalmente pelo tempo de vida obtido nos testes observados, falta de bibliotecas que facilitem a manipulação do *hardware* e falta de protocolos de comunicação já implementados para a plataforma.

7.1 Direções futuras

Como trabalhos futuros planejamos desenvolver bibliotecas que facilitem a manipulação do *hardware,* bibliotecas que forneçam algumas funcionalidades básicas de *middleware*, como por exemplo, agregação de dados. Além disto, planejamos conduzir uma análise comparativa do PRISMA com o *middleware* Mires e analisar o impacto de vários parâmetros no desempenho do PRISMA (por exemplo, número de nós sensores na rede, requisitos de topologia e diferentes requisitos de aplicações combinados).

Visando atender a tendência de RSSF, planejamos fornecer suporte a atuadores, com o objetivo de atender à maior gama de aplicações possíveis.

Referências

[1] MEMSIC. **MEMSIC**. Dispoível em: http://www.memsic.com/. Acesso em: Junho/2012.

[2] TinyOS. **TinyOS**. Dispoível em: http://www.tinyos.net/. Acesso em: Junho/2012.

[3] Oracle. **Oracle**. Disponível em: http://www.oracle.com/br/index.html. Acesso em: Junho/2012.

[4] Arduino. **Arduino**. Disponível em: http://arduino.cc/. Acesso em: Junho/2012.

[5] P. Vicaire ; T. F. Abdelzaher ; T. He ; Q. Cao ; T. Yan; G. Zhou ; L. Gu ; L. Luo ; R. Stoleru ; J. a. Stankovic ; Achieving long-term surveillance in VigilNet ; *ACM Trans. Sens. Networks,* New York, v. 5, n. 1, p. 1–39, Feb. 2009.

[6] V. Berisha ; A. Spanias ; Real-time sensing and acoustic scene characterization for security applications ; In: *INT. SYMP. WIREL. PERVASIVE COMPUT.* 2008, 2008, Santorini. Proceedings of the IEEE International Symposium on Wireless Pervasive Computing 2008, 2008. v. 2, p. 755–758.

[7] M. J. Whelan ; K. D. Janoyan ; Design of a Robust, High-rate Wireless Sensor Network for Static and Dynamic Structural Monitoring ; *J. Intell. Mater. Syst. Struct.*, v. 20, n. 7, p. 849–863, Nov. 2008.

[8] S. Kim ; S. Pakzad ; D. Culler ; J. Demmel ; G. Fenves ; S. Glaser ; M. Turon ; Health monitoring of civil infrastructures using wireless sensor networks ; In: *6th Int. Symp. Inf. Process. Sens. Networks*, 2007, Cambridge, MA. Information Processing in Sensor Networks, 2007. p. 254–263.

[9] W. Broll ; I. Lindt ; J. Ohlenburg ; I. Herbst ; M. Wittkämper ; T. Novotny ; An infrastructure for realizing custom-tailored augmented reality user interfaces ; *IEEE Trans. Vis. Comput. Graph.*, v. 11, n. 6, p. 722–33, Nov. 2005.

[10] C. Efstratiou ; I. Leontiadis ; C. Mascolo ; J. Crowcroft ; A shared sensor network infrastructure ; In: *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, 2010, New York, NY. Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems, 2010. p. 367-368.

[11] I. Leontiadis ; C. Efstratiou ; C. Mascolo ; J. Crowcroft ; SenShare : Transforming Sensor Networks Into Multi-Application Sensing Infrastructures ; In: Proceedings of the 9th European conference on Wireless Sensor Networks, 2012, Trento, Italy : Springer-Verlag, Berlin, 2012. p. 65-81.

[12] K. Mechitov ; G. Agha ; Building portable middleware services for heterogeneous cyber-physical systems ; In: *Third Int. Work. Softw. Eng. Sens. Netw. Appl.*, 2012, Piscataway, NJ. p. 31–36.

[13] C. Efstratiou ; Challenges in Supporting Federation of Sensor Networks ; In: *NSF/FIRE Workshop on Federating Computing Resources*. Princeton, NJ, 2010.

[14] P. del Cid ; S. Michiels ; Middleware for resource sharing in multi-purpose wireless sensor networks ; In: Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on. IEEE, 2010. p. 1-8.

[15] X. Koutsoukos ; M. Kushwaha ; I. Amundson ; S. Neema ; J. Sztipanovits ; OASiS: A service-oriented architecture for ambient-aware sensor networks ; In: Composition of Embedded Systems. Scientific and Industrial Issues. Springer Berlin Heidelberg, 2008. p. 125-149.

[16] A. Taherkordi ; Q. Le-Trung ; R. Rouvoy ; F. Eliassen ; WiSeKit: A Distributed Middleware to Support Application-level Adaptation in Sensor Networks ; In: Distributed Applications and Interoperable Systems. Springer Berlin Heidelberg, 2009. p. 44-58.

[17] P. Boonma ; J. Suzuki ; BiSNET: A biologically-inspired middleware architecture for self-managing wireless sensor networks ; *Computer networks, v. 51, n. 16, p. 4599-4616, 2007.*

[18] B. Valente ; F. Martins ; A Middleware Framework for the Internet of Things ; In: *Conf. Adv. Futur. Internet*, n. c, p. 139–144, 2011.

[19] M.-M. Wang ; J.-N. Cao ; J. Li ; S. K. Dasi ; Middleware for Wireless Sensor Networks : A Survey ; *Journal of computer science and technology, v. 23, n. 3, p. 305-326, 2008.*

[20] S. Hadim ; N. Mohamed ; Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks ; *IEEE distributed systems online, n. 3, p. 1, 2006.*

[21] F. Delicato, Middleware baseado em serviços para redes de sensores sem fio. 2005 196 Tese (Doutorado em Ciências em Engenharia elétrica) COPPE/UFRJ, 2005.

[22] W. Masri ; Z. Mammeri ; Middleware for wireless sensor networks: Approaches, challenges, and projects ; In: Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on. IEEE, 2007. p. 1399-1402.

[23] F. Shang ; J. Liu ; Multi-hop Topology Control Algorithm for Wireless Sensor Networks ; *Journal of Networks, v. 7, n. 9, p. 1407-1414, 2012.*

[24] J. Akbari Torkestani ; An energy-efficient topology construction algorithm for wireless sensor networks ; *Computer Networks, v. 57, n. 7, p. 1714-1725, 2013.*

[25] A. Konstantinidis ; H. Chen ; Q. Zhang ; Energy-aware Topology Control for Wireless Sensor Networks Using Memetic Algorithms ; *Computer Communications, v. 30, n. 14, p. 2753-2764, 2007.*

[26] R. C. Abreu ; J. E. C. Arroyo ; A Particle Swarm Optimization Algorithm for Topology Control in Wireless Sensor Networks ; In: Computer Science Society (SCCC), 2011 30th International Conference of the Chilean. IEEE, 2011. p. 8-13.

[27] R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures. 2000. Tese de Doutorado. University of California, Irvine.

[28] J. Gubbi ; R. Buyya ; Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions ; Future Generation Computer Systems, v. 29, n. 7, p. 1645-1660, 2013.

[29] Y. Yao ; J. Gehrke ; The cougar approach to in-network query processing in sensor networks ; *ACM Sigmod Record, v. 31, n. 3, p. 9-18, 2002.*

[30] J. Heidemann ; F. Silva ; C. Intanagonwiwat ; R. Govindan ; D. Estrin ; D. Ganesan ; Building Efficient Wireless Sensor Networks with Low-Level Naming ; In: ACM SIGOPS Operating Systems Review. ACM, 2001. p. 146-159.

[31] A. E.-D. Mady ; G. Provan ; N. Wei ; Designing cost-efficient wireless sensor/actuator networks for building control systems ; In: Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings. ACM, 2012. p. 138-144.

[32] H. Sethu ; Cooperative topology control with adaptation for improved lifetime in wireless ad hoc networks ; *In: INFOCOM, 2012 Proceedings IEEE. IEEE, 2012. p. 262-270.*

[33] K. Khedo ; R. Perseedoss ; A. Mungur ; A wireless sensor network air pollution monitoring system ; *Int. J. Wirel. Mob. Networks*, v. 2, n. 2, p. 31–45, 2010.

[34] T. Riesgo ; J. Valverde ; V. Rosello ; G. Mujica ; J. Portilla ; A. Uriarte ; Wireless Sensor Network for Environmental Monitoring : Application in a Coffee Factory ; *International Journal of Distributed Sensor Networks, 2012.*

[35] K. Chintalapudi ; T. Fu ; J. Paek ; Monitoring civil structures with a wireless sensor network ; *Internet Computing, IEEE, v. 10, n. 2, p. 26-34, 2006.*

[36] M. Rahimi ; H. Shah ; G. S. Sukhatme ; J. Heideman ; D. Estrin ; Studying the Feasibility of Energy Harvesting in a Mobile Sensor Network ; In: Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on. IEEE, 2003. p. 19-24.

[37] B. J. Hill ; M. Horton ; R. Kling ; The Plataforms Enabling Wireless Sensor Networks ; *Communications of the ACM, v. 47, n. 6, p. 41-46, 2004.*

[38] J. Hill ; R. Szewczyk ; A. Woo ; S. Hollar ; D. Culler ; K. Pister ; System architecture directions for networked sensors ; In: ACM SIGOPS operating systems review. ACM, 2000. p. 93-104.

[39] D. D. Wentzloff ; B. H. Calhoun ; R. Min ; N. Ickes ; a. P. Chandrakasan ; Design considerations for next generation wireless power-aware microsensor nodes ; *In: VLSI Design, 2004. Proceedings. 17th International Conference on. IEEE, 2004. p. 361-367.*

[40] I. Demirkol ; C. Ersoy ; F. Alagöz ; MAC Protocols for Wireless Sensor Networks : a Survey ; *Communications Magazine, IEEE, v. 44, n. 4, p. 115-121, 2006.*

[41] J. Al-Karaki ; A. Kamal ; Routing techniques in wireless sensor networks: a survey ; *Wireless communications, IEEE, v. 11, n. 6, p. 6-28, 2004.*

[42] I. F. F. Akyildiz ; W. Su ; Y. Sankarasubramaniam ; E. Cayirci ; Wireless sensor networks: a survey ; *Computer networks, v. 38, n. 4, p. 393-422, 2002.*

[43] V. Raghunathan ; C. Schurgers ; S. Park ; M. B. Srivastava ; Energy-Aware Wireless Microsensor Networks ; *Signal Processing Magazine, IEEE, v. 19, n. 2, p. 40-50, 2002.*

[44] C. Alippi ; G. Anastasi ; Energy management in Wireless Sensor Networks with energy-hungry sensors ; *Instrumentation & Measurement Magazine, IEEE, v. 12, n. 2, p. 16-23, 2009.*

[45] A. Muqattash ; M. Krunz ; A single-channel solution for transmission power control in wireless ad hoc networks ; In: Proceedings of the 5th ACM international symposium on Mobile ad hoc networking and computing. ACM, 2004. p. 210-221.

[46] M. Z. Siam ; M. Krunz ; A. Muqattash ; S. Cui ; Adaptive multi-antenna power control in wireless networks ; In: Proceedings of the 2006 international conference on Wireless communications and mobile computing. ACM, 2006. p. 875-880.

[47] E. L. Lloyd ; S. S. Ravi ; R. Ramanathan ; R. Liu ; M. V. Marathe ; Algorithmic aspects of topology control problems for ad hoc networks ; *Mobile Networks and applications, v. 10, n. 1-2, p. 19-34, 2005.*

[48] L. Li ; J. Halpern ; P. Bahl ; A cone-based distributed topology-control algorithm for wireless multi-hop networks ; *Networking, IEEE/ACM Transactions on, v. 13, n. 1, p. 147-159, 2005.*

[49] Z. Yuanyuan ; X. Jia ; H. Yanxiang ; Energy efficient distributed connected dominating sets construction in wireless sensor networks ; In: Proceedings of the 2006 international conference on Wireless communications and mobile computing. ACM, 2006. p. 797-802.

[50] D. Chen ; P. Varshney ; QoS support in wireless sensor networks: A survey ; *In: International Conference on Wireless Networks. 2004. p. 227-233.*

[51] M. Sharifi ; M. a. Taleghan ; a. Taherkordi ; A Middleware Layer Mechanism for QoS Support in Wireless Sensor Networks ; *In: Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies, 2006. ICN/ICONS/MCL 2006. International Conference on. IEEE, 2006. p. 118-118.*

[52] M. Younis ; K. Akkaya ; M. Eltoweissy ; A. Wadaa ; On Handling QoS Traffic in Wireless Sensor Networks ; In: System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on. IEEE, 2004. p. 10.

[53] M. M.-A. Nef ; S. Karagiorgou ; G. I. Stamoulis ; P. K. Kikiras ; Supporting Service Differentiation in Wireless Sensor Networks ; *In: Informatics (PCI), 2011 15th Panhellenic Conference on. IEEE, 2011. p. 127-133.*

[54] C. Mascolo ; L. Capra ; W. Emmerich ; Mobile Computing Middleware ; In: Advanced lectures on networking. Springer Berlin Heidelberg, 2002. p. 20-58.

[55] R. Kay ; O. Kasten ; F. Mattern ; Middleware Challenges for Wireless Sensor Networks ; *ACM SIGMOBILE Mobile Computing and Communications Review, v. 6, n. 4, p. 59-61, 2002.*

[56] Y. Yu ; B. Krishnamachari ; V. K. Prasanna ; Issues in Designing Middleware for Wireless Sensor Networks ; *Network, IEEE, v. 18, n. 1, p. 15-21, 2004.*

[57] M. M. Molla ; S. I. Ahamed ; A Survey of Middleware for Sensor Network and Challenges ; *In: Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on. IEEE, 2006. p. 6.*

[58] C. Pautasso ; O. Zimmermann ; F. Leymann ; RESTful Web Services vs. 'Big' Web Services: Making the Right Architectural Decision. ; In: Proceedings of the 17th international conference on World Wide Web. ACM, 2008. p. 805-814.

[59] J. Sandoval ; RESTful Java Web Services, Master core REST concepts and create RESTful web services in Java ; *Packt Publishing Ltd, 2009.*

[60] Deborah N. ; Duncan Temple L. ; JavaScript Object Notation. In: XML and Web Technologies for Data Sciences with R. Springer New York, 2014. p. 227-253.

[61] R. Lucchi ; M. Millot ; C. Elfers ; Resource Oriented Architecture and REST ; *Assessment of impact and advantages on INSPIRE, Ispra: European Communities, 2008.*

[62] G. F. Coulouris ; J. Dollimore ; *Distributed systems: concepts and design*. Pearson Education A basic understanding of distributed systems as it is offered, eg, by the course Introduction to, 1989.

[63] J. Silva ; F. Delicato ; L. Pirmez ; Paulo F. P. ; Jesus M. T. P. ; Taniro C. R. ; Thais V. B. ; PRISMA: A Publish-Subscribe and Resource-Oriented Middleware for Wireless Sensor Networks ; In: *AICT 2014, The Tenth Advanced International Conference on Telecommunications*, Paris, France, 2014. p. 87–97.

[64] C.-H. Wu ; Y.-C. Chung ; Heterogeneous wireless sensor network deployment and topology control based on irregular sensor model ; In: Advances in Grid and Pervasive Computing. Springer Berlin Heidelberg, 2007. p. 78-88.

[65] H. Luo ; F. Ye ; J. Cheng ; S. Lu ; L. Zhang ; TTDD: Two-Tier Data Dissemination in Large-Scale Wireless Sensor Networks ; *Wireless Networks, v. 11, n. 1-2, p. 161-175, 2005.*

[66] A. A. Abbasi ; M. Younis ; A survey on clustering algorithms for wireless sensor networks ; *Computer communications, v. 30, n. 14, p. 2826-2841, 2007.*

[67] F. Xiangning ; S. Yulin ; Improvement on LEACH Protocol of Wireless Sensor Network ; In: *In: Sensor Technologies and Applications, 2007. SensorComm 2007. International Conference on. IEEE, 2007. p. 260-264.*

[68] S. Chand ; S. Singh ; B. Kumar ; Heterogeneous HEED Protocol for Wireless Sensor Networks ; *Wireless personal communications, v. 77, n. 3, p. 2117-2139, 2014.*

[69] Y.-L. Chen ; N.-C. Wang ; C.-L. Chen ; Y.-C. Lin ; A Coverage Algorithm to Improve the Performance of PEGASIS in Wireless Sensor Networks ; In: Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2011 12th ACIS International Conference on. IEEE, 2011. p. 123-127.

[70] XML. **XML**. Disponível em: http://www.w3.org/XML/. Acessado em: Setembro/2013

[71] HTTP. **HTTP**. Disponível em: http://www.w3.org/Protocols/. Acessado em: Setembro/2013.

[72] P. Boonma ; J. Suzuki ; TinyDDS: An Interoperable and Configurable Publish/Subscribe Middleware for Wireless Sensor Networks ; In: Wireless Technologies: Concepts, Methodologies, Tools and Applications, A. M. Hinze and A. Buchmann, Eds. IGI Global, 2011, pp. 819–846.

[73] K. K. Khedo ; R. K. Subramanian ; A Service-Oriented Component-Based Middleware Architecture for Wireless Sensor Networks ; In: International Journal of Computer Science and Network Security, v. 9, n. 3, p. 174-182, 2009.

[74] E. Souto ; G. Guimarães ; G. Vasconcelos ; M. Vieira ; N. Rosa ; C. Ferraz ; J. Kelner ; Mires: a publish/subscribe middleware for sensor networks ; In: Personal and Ubiquitous Computing, v. 10, n. 1, p. 37-44, 2006.

[75] F. C. Delicato ; J. M. T. Portocarrero ; J. R. Silva ; P. F. Pires ; R. P. M. de Araújo ; T. V. Batista ; MARINE : MiddlewAre for Resource and mIssion oriented sensor NEtworks ; In: ACM SIGMOBILE Mobile Computing and Communications Review, v. 17, n. 1, p. 40-54, 2013.

[76] A. Tomcat. **Apache Tomcat**. Disponível em: http://tomcat.apache.org/. Acessado em: Setembro/2013.

[77] MySQL. **MySQL**. Disponíel em: http://www.mysql.com/. Acessado em: Setembro/2013.

[78] Data Access Object. **Data Access Object**. Disponível em: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html. Acessado em Janeiro/2014.

[79] S. Text. **Sublime Text**. Disponível em: http://www.sublimetext.com/. Acessado em Outubro/2013.

[80] Stino. **Stino**. Disponível em: https://github.com/Robot-Will/Stino/. Acessado em: Dezembro/2013.

[81] Arduino Programming Language. **Arduino Programming Language**. Disponível em: http://arduino.cc/en/Reference/HomePage. Acessado em: Setembro/2012.

[82] Wiring. **Wiring**. Disponível em: http://wiring.org.co/. Acessado em: Setembro/2013.

[83] D. M. Ritchie ; The development of the C language ; In: ACM SIGPLAN Notices, v. 28, n. 3, p. 201-208, 1993.

[84] Digi International. **XBee**. Disponível em: http://www.digi.com/xbee/. Acessado em: Outubro/2013.

[85] A. Rapp. **XBee-Arduino**. Disponível em: https://code.google.com/p/xbee-arduino/. Acessado em: Outubro/2013.

[86] M. Hart. **PString**. Disponível em: http://arduiniana.org/libraries/pstring/. Acessado em: Novembro/2013.

[87] Narcoleptic. **Narcoleptic**. Disponível em: https://code.google.com/p/narcoleptic/. Acessado em: Novembro/2013.

[88] X-CTU. **X-CTU**. Disponível em: http://www.digi.com/support/productdetail?pid=3352. Acessado em: Outubro/2013.

[89] V. Basili ; G. Caldiera ; H. Rombach ; The goal question metric approach ; In: Encyclopedia of Software Engineering, 2002.

[90] F. Nah ; Fui-Hoon ; A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait? ; In: Behaviour & Information Technology, v. 23, n. 3, p. 153-163, 2004.

[91] Intel. **Intel Galileo**. Disponível em: http://www.intel.com.br/content/www/br/pt/do-it-yourself/galileo-maker-quark-board.html. Acessado em: Outubro/2014.

[92] Digi International. **XBee Quick Reference**. Disponível em: http://examples.digi.com/quick-reference /. Acessado em: Março/2014.

Apêndices

APÊNDICE A – INFORMAÇÃO ADICIONAL SOBRE O RÁDIO XBEE

Abaixo uma breve explicação dos modos de economia de energia presentes no rádio *XBee* e que podem ser configurados através do parâmetro SM:

- 0 - *No Sleep*: sem modo de dormência, o rádio fica ligado o tempo inteiro, modo indicado quando o coordenador está ligado a um computador para recepção de dados.
- 1 - *Pin Hibernate*: entra do modo de dormência após receber um nível alto de tensão no pino 9 do rádio. O tempo necessário para mudar do modo de dormência para o modo ativo (*wake-up*) é de 13,2 ms. É o modo com maior economia de energia. O rádio pode, então, ser controlado externamente por um evento ou por um microcontrolador ligado ao pino 9. Tem um consumo de energia menor do que 10 µA quando no estado de dormência. Nesse modo, quando é aplicado um nível de tensão CMOS de 3,3 volts, o rádio entra no estado de dormência e quando esse nível de tensão é retirado, ele volta à sua condição normal de operação. O rádio, no entanto, só volta ao nível de dormência após todos os dados em seu buffer sejam transmitidos.
- 2 - *Pin Doze*: entra do modo de dormência após receber um nível alto de tensão no pino 9 do rádio, sendo que tem um tempo de *wake-up* menor que o do *Pin Hibernate*, em torno de 2 ms, mas com uma economia de energia menor. Tem um consumo de energia menor do que 50 µA quando no estado de dormência.
- 3 - Reservada
- 4 - *Cyclic Sleep Remote*: possui o mesmo nível de economia de energia do *Pin Doze*, para entrar em estado de dormência após um determinado tempo ocioso. Esse modo possui um consumo de energia menor do que 50 µA quando no estado de dormência.
- 5 - *Cyclic Sleep Remote with PinWake-up*: equivalente ao *Cyclic Sleep Remote*. Esse modo possui um consumo de energia menor do que 50 µA quando no estado de dormência

No PRISMA o valor utilizado para o parâmetro SM do rádio é configurado para 1. Por termos intervalos bem definidos de coleta de dados, no restante do tempo em que ficaria *Idle* o rádio passa a estar em seu estado adormecido a fim de poupar energia e estender o tempo de vida da rede.

Ao utilizarmos o Arduino em conjunto com o rádio *XBee* para comunicação sem fio podemos encapsular os dados trocados pela rede em pacotes que seguem o padrão 802.15.4 [92]. Os pacotes são como mostrados na **Erro! Fonte de referência não encontrada.**:

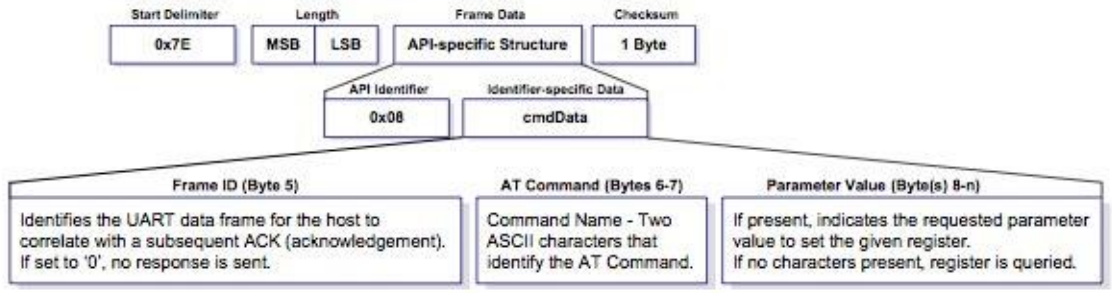

Figura 7.1: Formato do pacote de controle

Com este pacote da Figura 7.1**Erro! Fonte de referência não encontrada.** podemos alterar ou consultar os parâmetros de configuração do rádio XBee em tempo de execução. Tais parâmetros como, por exemplo: MY (endereço de 16-bits do nó), SM (*Sleep mode* do nó), SP (*Cyclic Sleep Period* do nó), etc. A lista completa de parâmetros pode ser encontrada em [92].

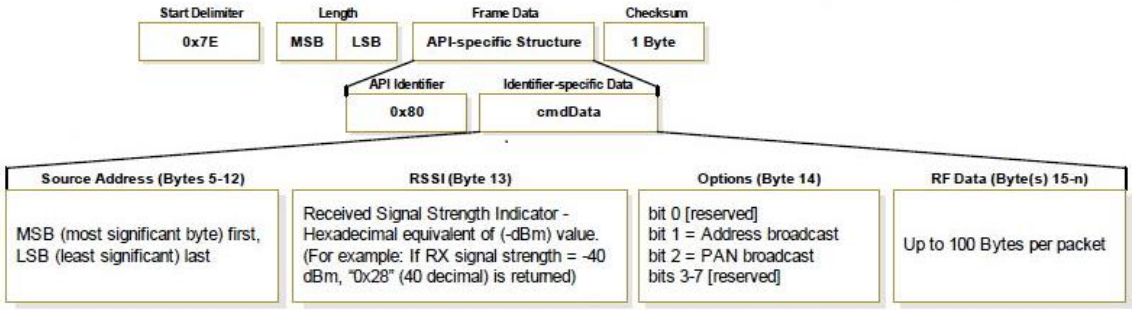

Figura 7.2: Formato do pacote de mensagem recebida

Este pacote mostrado na Figura 7.2 é o pacote que é recebido pelos nós. Nele encontramos o endereço de origem, a força do sinal e os dados encapsulados que foram coletados pelas aplicações que se encontram em execução.

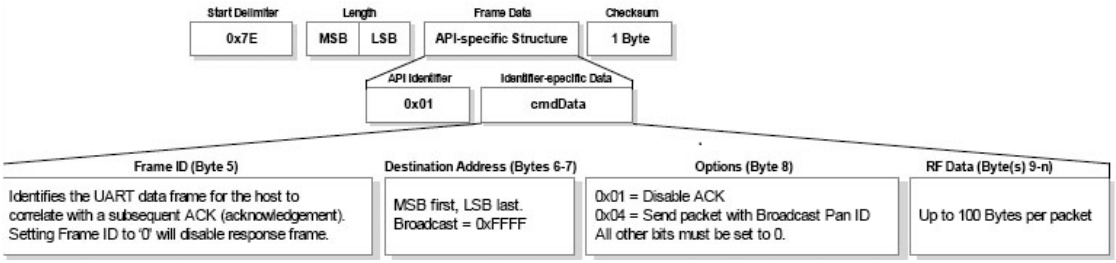

Figura 7.3: Formato do pacote de transmissão de mensagem

O pacote mostrado na Figura 7.3 é utilizado na transmissão de mensagens, tanto de dados quanto de controle. O pacote é similar ao pacote de mensagem recebida, a única alteração é a troca do endereço de origem pelo endereço de destino. Basicamente as mensagens são iguais, como o esperado, a diferença nesta representação dos pacotes se dá pelo fato de haver campos que são preenchidos automaticamente: na mensagem recebida o campo endereço destino é omitido por ser o próprio rádio que recebeu a mensagem e no pacote de transmissão o endereço origem é configurado automaticamente pelo rádio e por isto foi omitido.

APÊNDICE B – FIGURA QUE EXIBE A MODELAGEM DO BANCO DE DADOS

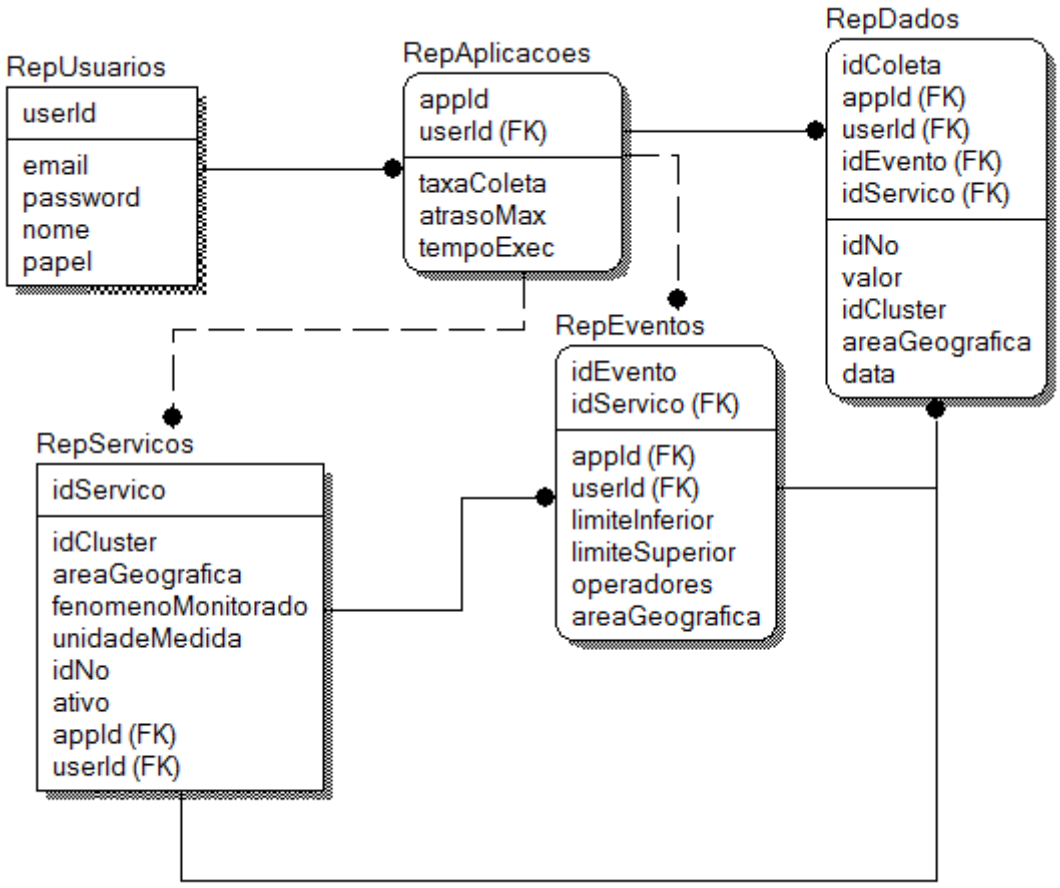

Figura 7.4: Modelo PRISMA