André Hahn Pereira
Henrique Carvalho Silva
Yuka Kyushima Solano

WARM: Arcabouço para programação de aplicações em redes de sensores sem fio

São Paulo
2015

Catalogação-na-publicação

Pereira, Andre
WARM: Arcabouço para programação de aplicações em redes de sensores sem fio / A. Pereira, H. Silva, Y. Solano -- São Paulo, 2015.
89 p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Redes de sensores sem fio 2.Programação 3.Gerenciamento de recursos 4.Desenvolvimento de aplicações 5.Gerenciamento de aplicações I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t. III.Silva, Henrique IV.Solano, Yuka

André Hahn Pereira
Henrique Carvalho Silva
Yuka Kyushima Solano

WARM: Arcabouço para programação de aplicações em redes de sensores sem fio

Trabalho de Formatura apresentado ao Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo para obtenção do Diploma de Engenheiro

Orientador: Profa. Dra. Cíntia Borges Margi
Coorientador: Msc. Bruno T. de Oliveira

São Paulo
2015

Resumo

A Internet das Coisas é cada vez mais um conceito presente no cotidiano com múltiplas aplicações. Entretanto, o correto funcionamento da Internet das Coisas depende da existência de redes de sensores presentes no ambiente para o fornecimento de informações para a tomada de decisões acertadas. Porém, atualmente a implementação e operação de redes de sensores são tarefas que dependem de conhecimentos de especialistas devido a sua alta complexidade. Dessa forma, o projeto se propõe a facilitar a implementação e operação de uma rede de sensores sem fio através da criação de um arcabouço para programação de aplicações em redes de sensores sem fio. O *framework* proposto, *WARM*, oferece como principais vantagens o fácil desenvolvimento e gerenciamento de tais aplicações, sem que seja necessário aos seus usuários conhecimento específico na área de redes de sensores sem fio.

Palavras-chaves: Redes de sensores sem fio; Programação; Desenvolvimento de aplicações; Gerenciamento de aplicações; Gerenciamento de recursos.

Abstract

The Internet of Things is a concept present in the daily life with several applications. Even so, the Internet of Things relies on the existence of sensor networks in the environment to provide data for a proper decision making. However, specialist knowledge in sensor networks is still required to implement and operate them due to their complexity. To try and solve this problem, this project proposes the creation of a framework for the programming of applications in wireless sensor networks. The proposed framework, *WARM*, has as main advantages the ease of development and management of such applications without the need of user specific knowledge about wireless sensor networks.

Key-words: Wireless sensor networks; Programming; Application development; Application management; Resource management.

Lista de Figuras

Lista de Tabelas

Lista de Siglas e Símbolos

API *Application Programming Interface*. 6, 13–15, 50, 65, 83–86

CORBA *Common Object Request Broker Architecture*. 6, 54

DAO *Data Access Object*. 6, 59, 68

ER Entidade-Relacionamento. 5, 6, 51, 59, 68

GPS *Global Positioning System*. 6, 36, 84

GSN *Global Sensor Networks*. 6, 19, 20

HTML *HyperText Markup Language*. 6, 64

HTTP *HyperText Transfer Protocol*. 6, 54, 65

ID identificador. 6, 36–49, 51

IDE *Integrated Development Environment*. 6, 57

IEEE *Institute of Electrical and Electronics Engineers*. 6, 17, 37, 60

JSON *JavaScript Object Notation*. 6, 65, 80

JVM *Java Virtual Machine*. 6, 56

LR-WPAN *Low Rate Wireless Personal Area Network*. 6, 21

MVC *Model-View-Controller*. 6, 55

nesC *network embedded systems C*. 6, 18, 59, 64, 69

ORM *Object-Relational Mapper*. 6, 58, 59

REST *Representational State Transfer*. 6, 10, 11, 13–15, 28, 29, 50, 54–56, 58, 63–65, 83–86

RPC *Remote Procedure Call*. 6, 54

RSSF Redes de Sensores sem Fio. 6, 9, 12–14, 17–19, 21–23, 25, 26, 28, 37, 59–61, 83–86

SDN *Software Defined Networks*. 6, 9, 12–14, 17, 20–24, 28, 35, 37, 50, 59, 75, 84, 86

SDWN *Software Defined Wireless Networks*. 6, 21

SMTP *Simple Mail Transfer Protocol*. 6, 54

SO Sistema Operacional. 6, 18, 21, 59, 85

SOAP *Simple Object Access Protocol*. 6, 10, 54, 55

SQL *Structured Query Language*. 6, 19, 20, 59

URI *Uniform Resource Identifier*. 6, 54

USB *Universal Serial Bus*. 6, 60

WSGI *Web Service Gateway Interface*. 6, 55

WSN *Wireless Sensor Networks*. 6, 17

XML *Extensible Markup Language*. 6, 19, 54

Sumário

1 Introdução

O advento da Internet das Coisas inclui a existência de uma rede de objetos físicos, e até de pessoas e animais, com a capacidade - a eles atribuída por meio de dispositivos eletrônicos embarcados providos de sensores e interfaces de comunicação - de se conectar uns aos outros e transferir dados entre si sem nenhuma interação homem-homem ou homem-máquina.

Uma parte importante da visão sugerida pelo conceito de Internet das Coisas, de acordo com Piuri e Minerva (2015), é o desenvolvimento de aplicações de Redes de Sensores sem Fio (RSSF). Uma RSSF é um tipo de rede composta por nós sensores autônomos com o objetivo de monitorar ambientes (CULLER; ESTRIN; SRIVASTAVA, 2004).

Apesar dessa importância, entretanto, a elaboração de uma uma aplicação para RSSFs ainda é uma tarefa muito custosa e complexa. Tal custo e complexidade são devidos ao fato de que o desenvolvimento de uma aplicação dessa tecnologia costuma requerer a especificação e a programação completa de toda a infra-estrutura de uma RSSF.

Isso acontece em razão das dificuldades envolvidas na reutilização de uma infra-estrutura de rede existente para o desempenho de novas tarefas, bem como no aproveitamento de seus recursos ociosos para potenciais novas aplicações. Não obstante, essa atividade de programação requer conhecimento específico relacionado ao *hardware* e ao sistema operacional dos nós sensores da rede, dificultando ainda mais esse custoso e complexo processo.

Em face desse problema, o trabalho proposto consiste na implementação de um *framework* (arcabouço) para o desenvolvimento de aplicações em RSSFs: o *WARM* (*Wireless Sensor Network Application development and Resource Management*).

WARM é uma ferramenta *open-source* baseada no paradigma de Redes Definidas por *Software*, do inglês *Software Defined Networks* (SDN), que tem por objetivo centralizar o controle de redes de computadores. Além disso, tem uma arquitetura altamente modular, evitando ao máximo a necessidade de interação direta com o *hardware* ou com tecnologias usada em camadas inferiores, de forma a permitir uma maior reutilização.

Com o objetivo de especificá-lo, foi realizada uma revisão bibliográfica com o propósito de obter conhecimento acerca do estado da arte no que se refere às soluções para o problema de se desenvolver e gerenciar aplicações para RSSFs. Com base no conteúdo pesquisado, *WARM* foi especificado e implementado de forma a atender aos requisitos

apresentados pelos trabalhos anteriores nessa área, também focando na reutilização da infra-estrutura disponível para múltiplas aplicações.

A arquitetura do *framework* proposto é, resumidamente, composta por dois componentes de *software*:

- Controlador: Responsável por fazer a interface do usuário com o controlador SDN que gerencia a infraestrutura de rede. O controlador do *WARM* interage com o usuário através de uma *Application Programming Interface* (API) REST, armazena informações sobre os nós presentes na rede e também conhece todas as tarefas sendo executadas atualmente, bem como os atributos e capacidades de cada nó.
- *Middleware* dos nós sensores: Responsável por fazer a interface entre a camada de rede e a camada de aplicação dos nós sensores participantes da rede mantida pelo arcabouço.

1.1 Objetivo

O objetivo do trabalho é a concepção, especificação, implementação, validação e análise de desempenho de um *framework open source* que facilite o desenvolvimento e o gerenciamento de aplicações em RSSFs. O *framework* deve permitir o aproveitamento da infra-estrutura de uma mesma rede existente por múltiplas aplicações simultâneas. Além disso, o usuário deve poder configurar novas aplicações sem que haja a necessidade de reprogramar cada nó sensor.

Não será necessário, para isso, que o usuário conheça a topologia de uma RSSF nem características específicas de cada nó. O *framework* deve ser capaz de abstrair esses detalhes, de forma que mesmo usuários sem conhecimento prévio da rede sejam capazes de utilizá-la.

Essa característica de fácil configuração e reconfiguração de múltiplas aplicações em uma mesma RSSF pode ser obtida através de um controle centralizado da mesma, possível com a aplicação do paradigma de redes definidas por software.

1.2 Metodologia de elaboração do projeto

A elaboração do projeto foi dividida em duas etapas principais, especificação e desenvolvimento, e a metodologia adotada é apresentada a seguir.

A primeira etapa começou com a definição do escopo do projeto, com base em interesses dos integrantes do grupo e sugestões dos orientadores. De posse da ideia inicial de escopo foi realizado o levantamento bibliográfico sobre o estado da arte na área de

interesse: suporte ao desenvolvimento de aplicações para RSSFs. Nesse levantamento, procurou-se dar maior relevância a artigos considerados como referências já consolidadas na área, mas também foram considerados trabalhos mais recentes que tivessem grande relação com a proposta deste projeto.

A princípio, a busca se concentrou em trabalhos que esclarecessem as definições e as características dos conceitos-chave para a realização do trabalho, como RSSF e SDN. Conforme se deu o amadurecimento de tais conceitos, o enfoque da pesquisa se voltou a trabalhos que pudessem trazer luz aos principais problemas existentes nas RSSFs e como eles poderiam ser solucionados através da proposta de um *framework* para o desenvolvimento e gerenciamento desse tipo de ambiente de rede.

A partir da revisão bibliográfica, foi realizado o levantamento dos requisitos do projeto e, com base neles, uma concepção de alto nível da sua arquitetura. Essa concepção permitiu que se tivesse uma ideia melhor dos dois módulos principais do *framework*, o controlador e o *middleware* já mencionados, bem como das interfaces entre eles.

Identificadas as interfaces entre os principais componentes do projeto, buscou-se então especificá-las em grande nível de detalhe. Tal empenho resultou na especificação de uma API REST para interface do controlador com o usuário e de um protocolo para a comunicação sem fio entre o controlador e o *middleware* presente nos nós sensores da rede.

Conhecendo-se as entradas e saídas dos dois principais módulos mencionados, pôde-se então especificar a arquitetura interna de cada um deles. O passo final da especificação consistiu na pesquisa e na decisão das tecnologias adequadas para a implementação de tais arquiteturas.

A segunda etapa do projeto consistiu na implementação do projeto e na realização de testes para verificar seu correto funcionamento e desempenho. Uma descrição mais detalhada da metodologia de desenvolvimento encontra-se na Seção 5.1.

1.3 Cronograma e atividades

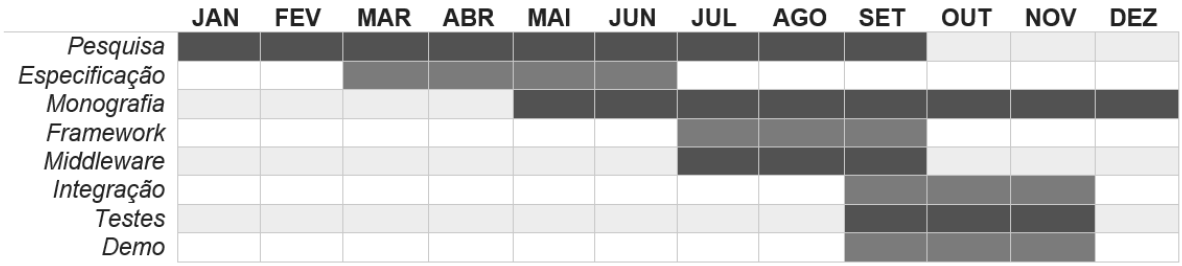
O projeto está dividido em seis partes principais:

- Pesquisa de tecnologias e estado da arte;
- Especificação dos componentes, protocolos de comunicação e tecnologias a serem usadas no projeto;
- Documentação do andamento do projeto e escrita da monografia;
- Implementação do controlador do *framework*;

- Implementação do *middleware*;
- Integração dos componentes;
- Realização dos testes, validação e análise de desempenho;
- Desenvolvimento da aplicação de demonstração.

O cronograma de execução das atividades é apresentado na Figura 1.

Figura 1 – Cronograma do projeto.



1.4 Organização do Documento

No Capítulo 2 é apresentada uma revisão bibliográfica sobre os temas pesquisados para a realização do projeto. Os temas de pesquisa relevantes são apresentados, junto com um breve resumo e referências bibliográficas dos assuntos estudados.

No Capítulo 3 é apresentada a especificação técnica do projeto, apresentando escopo do trabalho, requisitos funcionais e não funcionais, arquitetura da solução do projeto e os protocolos de comunicação entre as interfaces.

No Capítulo 4 são apresentadas as tecnologias consideradas e adotadas no projeto, bem como análises e justificativas das escolhas realizadas.

No Capítulo 5 são apresentados os métodos de desenvolvimento adotados para a implementação do projeto, incluindo versionamento, revisão, documentação e testes. Em seguida, são relatadas as implementações dos três componentes principais da arquitetura do *WARM*: o servidor da API REST, o controlador do *framework* e o *middleware* presente nos nós sensores.

No Capítulo 6 são apresentados os testes realizados para garantir e validar o funcionamento do *framework* implementado, incluindo testes de módulos e componentes, testes isolados dos principais componentes e testes de integração entre eles.

O Capítulo 7 dá encerramento ao trabalho, discutindo os resultados obtidos, relatando possíveis caminhos para o seu prosseguimento, e tecendo reflexões a respeito do significado que ele teve enquanto trabalho final de graduação.

Parte I

Planejamento

2 Revisão bibliográfica

Neste capítulo encontra-se uma revisão bibliográfica geral sobre assuntos que foram pesquisados e estudados para permitir a realização deste trabalho. Primeiro há uma apresentação sobre os conceitos de Redes de Sensores sem Fio (RSSF), seguido por uma apresentação de outros trabalhos presentes na literatura sobre gerenciamento e consultas para RSSF. Em sequência, apresenta-se a definição de *Software Defined Networks* (SDN), para então descrever alguns trabalhos da literatura sobre SDN aplicada a RSSF.

2.1 Redes de sensores sem fio

Redes de Sensores sem Fio (RSSF) (CULLER; ESTRIN; SRIVASTAVA, 2004) (em inglês *Wireless Sensor Networks* (WSN)) são redes compostas por sensores autônomos espacialmente distribuídos para monitorar condições físicas e ambientais, como temperatura e pressão. Os nós destas redes são dispositivos de baixo custo capazes de se comunicar entre si através de conexões sem fio, de modo a colaborar uns com os outros para a realização do propósito de uma aplicação de RSSF. Além disso, podem ainda ser móveis, heterogêneos e, em alguns cenários, podem suportar condições ambientais adversas.

Denominados de nós sensores, os elementos que compõem uma RSSF são dispositivos marcados por duas limitações principais: a de recursos computacionais − incluindo armazenamento e processamento − e a de fonte de energia − geralmente fornecida por uma bateria de baixa capacidade. Tais características exigem que a programação de aplicações para nós sensores seja eficiente na utilização dos recursos disponibilizados pelo *hardware* desses dispositivos. Alguns exemplos de plataformas de nós sensores utilizadas em aplicações para RSSF são o *TelosB* e o *micaZ*.

Outra característica importante de uma RSSF é a sua arquitetura de rede, que pode ser encarada como uma classe especial de redes *ad hoc* de múltiplos saltos (MARGI, 2015). Devido ao requisito adicional de baixo consumo de energia e as características dos dispositivos de rádio empregados nessas redes − como baixas taxas de transferência e latência elevada − são necessários protocolos diferenciados para a comunicação nesse tipo de rede. Um dos protocolos de comunicação de camada de enlace mais utilizados em RSSF é o IEEE 802.15.4, que prevê baixas taxas de transferência e tem foco no baixo consumo de energia.

Aplicações de RSSF estão se tornando cada vez mais populares em diversas áreas

da atividade econômica, como agricultura de precisão, construção civil, controle de tráfego, monitoramento de pacientes e logística de produtos. Ainda assim, o que vemos hoje é apenas uma sombra do verdadeiro potencial representado pelas RSSFs, que só se tornará claro com o advento da Internet das Coisas − que tornará extremamente populares as aplicações dessa tecnologia. No entanto, há diversos desafios para a popularização das RSSFs, a maioria relacionada à dificuldade de gerenciar e programar suas aplicações (PIURI; MINERVA, 2015).

2.1.1 *TinyOS*

O *TinyOS* (LEVIS et al., 2005) é um Sistema Operacional (SO) de código aberto para RSSF. Ele foi criado na Universidade da Califórnia em Berkeley com o objetivo de facilitar a programação de aplicações para RSSFs e é amplamente utilizado por pesquisadores na comunidade acadêmica.

O *TinyOS* é um SO orientado a eventos e sua programação é feita na linguagem *network embedded systems C* (nesC) (GAY et al., 2003), uma extensão da linguagem C. Ele foi projetado tendo em vista um baixo consumo de energia e de memória, de forma a permitir a operação em nós sensores com energia e memória limitadas. O SO é capaz de executar múltiplas tarefas concorrentes de maneira segura.

Foi desenvolvido também um simulador para o *TinyOS* chamado *TOSSIM* (LEVIS et al., 2003), que permite a simulação de aplicações para o *TinyOS* e possui interface de programação em C++ e *Python*.

2.2 Gerenciamento e consultas para RSSF

Nessa seção são apresentadas breves descrições de alguns dos principais trabalhos relacionados ao gerenciamento e ao monitoramento de redes sensores e internet das coisas. Eles serviram de base para a definição de requisitos funcionais e não-funcionais neste projeto.

2.2.1 *TinyDB*

O *TinyDB* (MADDEN et al., 2005) é um processador de consultas distribuído que roda sobre o *TinyOS*, desenvolvido na Universidade da Califórnia em Berkeley.

O *TinyDB* tem a maioria das características de um processador de consultas tradicional, mas também incorpora características projetadas para minimizar o consumo de energia através de técnicas aquisicionais. Nesse trabalho as consultas são feitas através de um PC, responsável por interpretar, otimizar e enviar as consultas para a RSSF.

As principais questões consideradas acerca do processamento de consultas em RSSFs por esse trabalho são: em que momento as amostras de uma consulta devem ser tomadas; quais nós têm dados relevantes para a consulta; qual a ordem na qual as amostras devem ser feitas; e como intercalar a amostragem com outras operações.

As consultas no *TinyDB* seguem o padrão básico do *Structured Query Language* (SQL), com o acréscimo da opção de definir um intervalo entre medidas, bem como a duração da consulta. Consultas também podem realizar ações, como o acionamento de evento em um nó ou de um atuador.

2.2.2 *PyoT*

O *PyoT* (AZZARA et al., 2014) é um *framework* de "macroprogramação" para internet das coisas. Esse é um trabalho recente, de 2014, desenvolvido em Pisa, na Itália.

Seu foco é esconder completamente os nós e a rede do usuário. Suas funcionalidades incluem: descobrir recursos disponíveis de maneira automática, monitorar dados de sensores, manipular seu armazenamento, controlar atuadores, definir eventos e suas ações, e interagir com seus recursos utilizando linguagem de *script*. O *PyoT* foi projetado para rodar em redes IP e a obtenção de dados da rede pode ser feita através de requisições do tipo *GET*.

O *PyoT* não leva em conta restrições de processamento, comunicação ou energia nos nós da rede, pois grande parte da lógica da rede é executada nos próprios nós, e protocolos web convencionais são utilizados para comunicação, além da programação ser realizada em *Python*, uma linguagem pouco eficiente do ponto de vista de processamento e memória.

2.2.3 *Global Sensor Networks*

O *Global Sensor Networks* (GSN) (ABERER; HAUSWIRTH; SALEHI, 2006) é um *middleware open-source* que suporta a descoberta e a integração flexível de RSSFs. Seus objetivos são tornar possíveis a configuração dinâmica de aplicações de RSSFs em operação e a realização de consultas, filtros e combinação de dados coletados por sensores de forma distribuída.

O principal obstáculo para a realização de tais objetivos está na heterogeneidade das plataformas de hardware e software disponíveis para RSSF atualmente. Para contornar esse problema, os autores propõem uma maneira − denominada por eles de "sensores virtuais" − de se abstrair os nós sensores através em descritores baseados em *Extensible Markup Language* (XML).

Os sensores virtuais são a peça principal no modelo de abstração provido pelo GSN. Eles abstraem o acesso aos dados produzidos por nós sensores e são administrados

e disponibilizados, pelo GSN, na forma de serviços. O modelo propõe que um nó sensor tenha um número qualquer de entradas de dados, advindos de sensores ou de outros nós da rede, mas apenas uma saída de dados.

Entre as informações que devem ser providas na especificação de um sensor virtual, vale a pena ressaltar as seguintes:

- Metadados para a identificação e descoberta dos sensores;
- A estrutura das entradas e saídas de dados;
- Uma especificação declarativa do processamento de dados que o sensor é capaz de realizar, que no caso do GSN é baseada em SQL;
- Propriedades funcionais relacionadas à persistência, tratamento de erros ocorridos, ciclo de vida e instalação física do sensor.

2.3 *Software Defined Networks*

Redes Definidas por *Software*, do inglês *Software Defined Networks* (SDN), é um paradigma para a arquitetura de redes de computadores caracterizado sobretudo pelo desacoplamento entre o plano de dados e o plano de controle da rede (KOLDEHOFE et al., 2012). Esse princípio permite que a rede seja vista como uma única entidade virtual, cuja configuração pode ser feita de forma centralizada e automatizada via *software* − ao invés de demandar a reprogramação manual de conjuntos distintos de configurações em centenas e até milhares de dispositivos.

As vantagens do paradigma para a administração dos recursos de uma rede estão relacionadas sobretudo à flexibilização da configuração, do gerenciamento, da segurança e da otimização dos recursos da rede. Essas vantagens contribuem ainda para facilitar a definição e a aplicação de políticas de rede. Uma outra vantagem é a abstração dos detalhes de funcionamento dos dispositivos presentes em uma rede, que, apesar de heterogêneos, podem ser todos configurados da mesma forma utilizando SDN.

O funcionamento de uma rede definida por *software* está baseado no conceito de fluxos para identificar o tráfego de dados na rede. Um pacote transmitido recebe um rótulo referente ao seu fluxo, que determina as regras para o seu tráfego na rede. Dispositivos que encaminham pacotes na rede, como *switches* e roteadores, são configurados com regras que determinam o que deve ser feito com um pacote recebido com o rótulo de um determinado fluxo. Essas regras podem ser simples, como “descartar”, “receber” ou “encaminhar”, mas também podem ser complexas, considerando partes do conteúdo do pacote e variáveis de estado da rede.

2.4 SDN aplicado a RSSF

Essa seção descreve três trabalhos relacionados à área de SDN aplicado a redes sem fio e redes de sensores sem fio.

2.4.1 *Software Defined Wireless Networks*

O trabalho de Costanzo et al. (2012) examina como o paradigma SDN pode ser benéfico a ambientes de rede sem fio sem infra-estrutura, denominados como *Low Rate Wireless Personal Area Network* (LR-WPAN) pelos autores, e como ele deveria ser expandido para levar em conta as características de tais ambientes. Tais vantagens não estariam relacionadas a aumentos de eficiência ou de desempenho, mas sim à flexibilidade na escolha de soluções de controle e administração de tais ambientes.

Algumas das características de SDN são bastante adequadas à sua aplicação a LR-WPAN, sobretudo no que diz respeito às restrições de consumo de energia desses ambientes. Entre essas características está a flexibilidade na definição de regras de fluxo e de algoritmos de otimização, que, entre outras coisas, possibilitam a agregação de dados através de regras de roteamento.

Além de apresentar as vantagens de se aplicar SDN a LR-WPAN, o trabalho também apresenta uma arquitetura de *Software Defined Wireless Networks* (SDWN), que põe em prática essa aplicação. Ela define uma estrutura geral baseada em dois componentes:

- Nó sorvedouro: um dispositivo conectado a um sistema embarcado rodando um SO baseado em Linux, capaz de desempenhar as tarefas do controlador SDN. A arquitetura de software proposta para esse dispositivo envolve principalmente duas camadas:
  - Camada de controle, responsável por aplicar as políticas de administração da rede;
  - Camada de virtualização, responsável por manter uma representação do estado atual da rede.
- Nós genéricos: demais dispositivos conectados à rede. A arquitetura de software proposta para eles envolve também duas camadas principais:
  - Camada de agregação, capaz de agregar informações circulando através da rede;
  - Camada de encaminhamento, capaz de encaminhar pacotes de acordo com os fluxos determinados pelo controlador SDN.

Por fim, o artigo também menciona alguns detalhes de implementação, entre eles alguns tipos de pacotes, como *data*, *beacon*, *report*, *rule/action request* e *rule/action response*, que transmitem desde dados e regras de fluxo até informações sobre a topologia da rede. Também são propostos quatro tipos de ações que uma regra poderia aplicar a um pacote de dados: modificar, descartar, agregar e desligar o rádio. A preocupação com o uso eficiente de memória no armazenamento das regras de fluxo é enfatizada.

2.4.2 *Sensor OpenFlow*

O trabalho de Luo, Tan e Quek (2012) tem por objetivo apontar como a arquitetura atual adotada para RSSF é complexa e deficitária de boas abstrações, sugerindo como solução a aplicação do paradigma SDN por meio do protocolo *Sensor OpenFlow*, baseado no padrão *OpenFlow* de SDN.

O fato das RSSFs terem sido concebidas para serem específicas a aplicações distintas lhes dá características como:

- Subutilização de recursos, pois múltiplas RSSFs são utilizadas para aplicações respectivas, ao invés de se utilizar uma única RSSF versátil para múltiplas aplicações;
- Contra-produtividade, devido à falta de interoperabilidade entre produtos desenvolvidos para RSSFs, o que prejudica a velocidade de desenvolvimento de uma aplicação;
- Rigidez das regras de negócio, devida à necessidade de reconfiguração manual de tais regras em cada um dos nós que compõem a rede.
- Dificuldade de administração, pois o desenvolvimento de um sistema de administração para RSSFs é uma tarefa difícil e suscetível a muitos erros.

Para combater esses problemas, os autores propõem a adoção de uma nova arquitetura para as RSSFs, baseada no paradigma de SDN. Tal arquitetura dividiria claramente os planos de dados e de comunicação em uma RSSF.

O plano de dados seria constituído pelos nós sensores da rede, que produzem e encaminham dados conforme os fluxos especificados por um controlador. Já o plano de controle estaria nas mãos de um ou mais nós controladores, que centralizaria as tarefas de roteamento e controle de qualidade de serviço. A comunicação entre esses planos se daria por meio do protocolo *Sensor OpenFlow*.

O objetivo principal é fazer com que os nós sensores no plano de dados sejam programados através da manipulação de regras de fluxo da rede. Isso tornaria uma RSSF versátil, flexível e fácil de administrar.

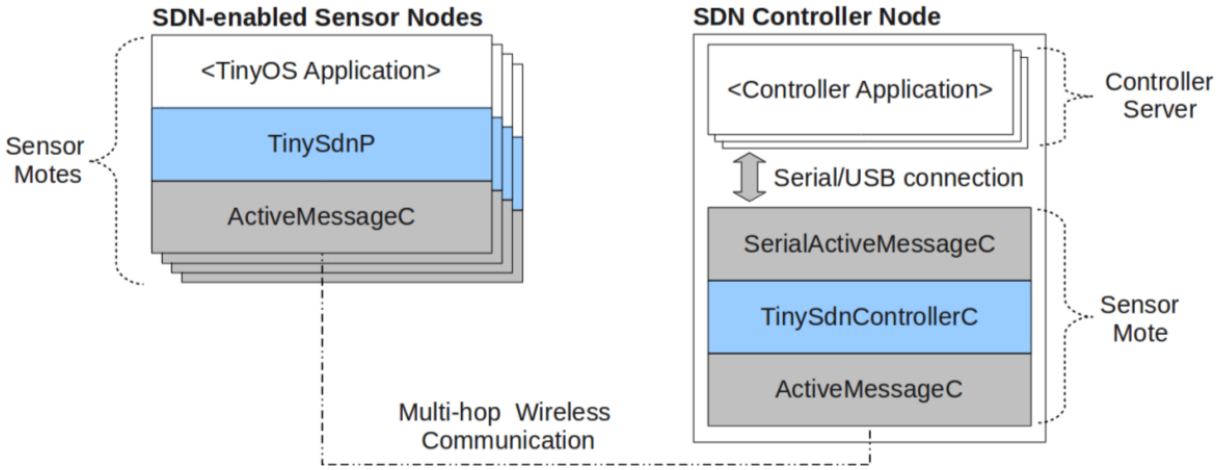
O artigo prossegue levantando os principais desafios técnicos para a implementação de tal arquitetura, sobretudo no que se refere à adaptação do protocolo *OpenFlow* ao caso das RSSFs, que foi inicialmente concebido para redes cabeadas. Entre os desafios está a necessidade de se lidar com a geração e o processamento de dados em rede, algo que não está previsto no paradigma SDN. Para os autores, entretanto, tal problema pode ser solucionado com a implementação, nos nós sensores, de módulos específicos para as aplicações a que eles se destinam.

2.4.3 *TinySDN*

O *TinySDN* (OLIVEIRA; MARGI; GABRIEL, 2014) é um *framework* de redes definidas por software para Redes de Sensores sem Fio (RSSF) que permite o uso de múltiplos controladores. Ele está sendo desenvolvido na Escola Politécnica da USP.

O *TinySDN* é implementado sobre o *TinyOS* e possui dois componentes principais: o nó sensor compatível com SDN, que consiste do switch SDN e de um dispositivo final SDN; e do nó controlador SDN, onde o plano de controle é programado. Os componentes são ilustrados na figura 2. A conexão entre um nó sensor e o controlador é feita através de uma rede sem fio de múltiplos saltos.

Figura 2 – Componentes do *TinySDN*.



Fonte: (OLIVEIRA; MARGI; GABRIEL, 2014)

Não existem mecanismos de confirmação de entrega de mensagens no *TinySDN*, dessa forma perdas de pacotes podem ocorrer. Quando um nó da rede recebe uma mensagem com fluxo ainda não presente em sua tabela de fluxos ele se comunica com o controlador, perguntando qual o próximo salto do fluxo a partir do nó. O controlador consulta sua programação e responde ao nó, que armazena a informação em sua tabela e trata o pacote de acordo com a resposta. Há duas ações possíveis de resposta do controlador: *forward* e *drop*, que direcionam o pacote ou o descartam, respectivamente.

Assim que um nó compatível com SDN é ligado ele busca um controlador SDN e associa-se a ele. Para identificação dos vizinhos os nós fazem broadcast de pacotes de *beacon* e aguardam respostas, medindo a qualidade do sinal dos pacotes recebidos. De posse da informação sobre os nós vizinhos, o nó envia esses dados ao controlador.

3 Especificação

No Capítulo 2 foram vistas quais as características desejadas na infraestrutura de uma RSSF e como já existem alguns trabalhos com o objetivo de determinar o papel que um *framework* deve ter na facilitação do desenvolvimento e do gerenciamento de aplicações em uma RSSF com essas características. Esse capítulo se propõe a especificar uma solução técnica para o desempenho desse papel.

O primeiro passo para isso é a definição, feita na Seção 3.1, do que deve ser incluído e do que deve ser excluído da solução técnica especificada, devido à limitação dos recursos disponíveis ao projeto. Em seguida, são levantados, na Seção 3.2, os requisitos funcionais e não-funcionais para o cumprimento dos itens especificados no escopo.

A partir dos requisitos do projeto, foi então proposta, na Seção 3.3, uma arquitetura simplificada da solução do projeto, com o objetivo de dividir o papel de cada componente do projeto no cumprimento dos requisitos levantados. Com base nesse diagrama, foram definidas as interfaces entre os componentes do projeto, para então especificar protocolos para elas nas Seções 3.4 e 3.5.

Por fim, tendo sido definidas e especificadas as interfaces entre os principais componentes do projeto, foi possível dividi-los em módulos que desempenhem funções relacionadas às entradas e saídas previstas pelas interfaces elaboradas. Essa divisão em módulos é apresentada nas seções 3.6 e 3.7, sendo base para a descrição que se faz do comportamento de cada um.

3.1 Escopo do projeto

O Capítulo 2 mostrou que um *framework* que facilite o desenvolvimento de aplicações em RSSFs tem como principais objetivos e características:

- Fácil configuração de uma aplicação para uma RSSF;
- Ausência da necessidade de se programar, especificamente para uma aplicação configurada, cada um dos nós sensores presentes na infraestrutura da RSSF;
- Compartilhamento, por múltiplas aplicações, da infraestrutura existente de uma RSSF;
- Fácil configuração dos parâmetros da rede, incluindo protocolos de roteamento, otimizações e ciclos de trabalho;

- Fácil adição e remoção de nós sensores a uma RSSF existente;
- Possibilidade de suporte a múltiplas plataformas de nós sensores;
- Ausência da necessidade de que o usuário conheça detalhes do funcionamento e da programação de cada um dos nós sensores.

Devido à limitação de tempo e de pessoas envolvidas no projeto, é necessário limitar seu escopo a somente alguns desses itens, cuja importância se sobressaia em relação aos demais. Tais itens foram escolhidos tendo em vista a viabilidade de sua especificação e implementação com os recursos disponíveis, e também a sua importância para o cumprimento dos objetivos do projeto. Eles serão detalhados na Seção 3.2.

Os itens listados a seguir não serão incluídos na especificação, principalmente devido a um ou mais dentre os seguintes motivos:

1. Necessidade de um esforço incompatível com os recursos disponíveis para a sua especificação e implementação;
2. Existência de uma ampla gama de soluções para problemas similares;
3. Possibilidade de se estender o projeto futuramente, com o objetivo de incluí-los;
4. Importância secundária para que o objetivo principal do projeto seja alcançado.

Segue uma relação do que não será nem especificado nem implementado no projeto:

- Autenticação do usuário, devido às razões 2, 3 e 4 apresentadas acima;
- Confidencialidade, integridade e autenticidade na transmissão de dados entre nós sensores da rede, devido às razões 1 e 3;
- Isolamento entre os dados pertencentes a aplicações diferentes, devido às razões 1 e 4;
- Módulos de extensão externos que provenham outros tipos de usabilidade no acesso às funcionalidades disponibilizadas pelo *framework*, devido às razões 1 e 3;
- Reprogramação remota dos nós sensores que se encontram em campo, devido às razões 1 e 3;

3.2 Requisitos do projeto

Na subseções que se seguem, são apresentados os requisitos funcionais e não-funcionais levantados para o escopo do projeto.

3.2.1 Requisitos funcionais

RF1 Usuário deve ser capaz de configurar aplicações que utilizem a infraestrutura da rede de sensores sem fio;

RF2 Usuário deve ser capaz de obter uma lista dos nós sensores baseada em identificadores ou em localização geográfica;

RF3 Usuário deve ser capaz de obter informações sobre os nós sensores que compõem a rede, incluindo as tarefas que podem desempenhar e dados sobre o seu desempenho;

RF4 Usuário deve ser capaz de agendar tarefas, a serem desempenhadas pelos nós sensores da rede que componham a aplicação configurada, sem que seja necessário reprogramar os nós;

RF5 Usuário deve ser capaz de especificar parâmetros de configuração específicos de cada tarefa a ser realizada pelos nós sensores;

RF6 Usuário deve ser capaz de referenciar os dados produzidos por nós sensores através de rótulos, de forma a especificar tarefas encadeadas;

RF7 Usuário deve ser capaz de determinar um destino, dentro da rede, dos dados recolhidos para a sua aplicação.

3.2.2 Requisitos não-funcionais

RNF1 Controle centralizado da rede;

RNF2 Novos nós que passem a integrar a rede devem ser incorporados automaticamente à infraestrutura disponibilizada ao usuário;

RNF3 O usuário leigo no funcionamento dos nós sensores e da rede deve ser capaz de obter dados a partir deles;

RNF4 Extensibilidade no que se refere à inclusão de novas plataformas de hardware e software de nós sensores e novos tipos de funcionalidades que elas envolvam;

RNF5 Interoperabilidade com outros sistemas, de forma que os serviços providos possam ser acessados de forma transparente;

RNF6 Garantir o uso balanceado dos recursos disponíveis na rede e nos nós sensores;

RNF7 Portabilidade do sistema em relação a plataformas de hardware e software;

RNF8 Mapeamento automático de sequências de tarefas especificadas pelo usuário através dos nós sensores da rede.

3.2.3 Garantias para os requisitos não-funcionais

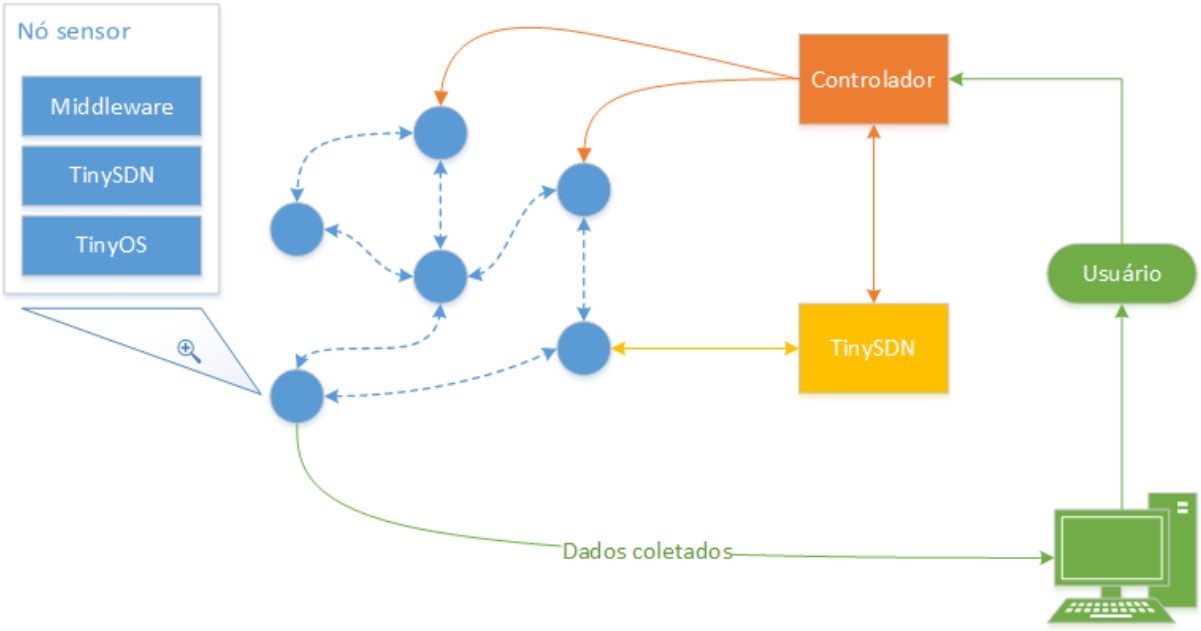
Os requisitos não-funcionais RNF1, RNF2 e RNF6 podem ser garantidos através do paradigma de SDN. Os requisitos RNF3 e RNF5 podem ser alcançados através do uso de um protocolo de interface com o usuário que seja popular, como o REST, que é descrito na Seção 4.1.

Uma arquitetura que desacople as funcionalidades dos nós sensores do restante do sistema irá satisfazer os requisitos não-funcionais RNF4 e RNF7 que, juntamente com o RNF5, também são favorecidos com uma solução modular para o sistema.

3.3 Diagrama de arquitetura simplificado

A arquitetura do *framework* é composta por um controlador e pelo *middleware* presente nos nós sensores da rede, como ilustrado na figura 3.

Figura 3 – Diagrama de arquitetura simplificado.



O controlador é responsável por fazer a interface do usuário com o controlador SDN. O controlador do *framework* irá interagir com o usuário através de arquitetura REST, por meio da qual o usuário conseguirá fazer consultas, agendar tarefas e, de maneira geral, gerenciar aplicações da RSSF sem a necessidade de saber detalhes específicos da topologia da rede ou dos nós existentes. O controlador armazena informações sobre os nós presentes na rede e também conhece todas as tarefas sendo executadas atualmente, bem como os atributos de cada nó.

O *middleware* nos nós sensores é responsável por fazer a interface entre a camada de rede e a camada de aplicação dos nós sensores participantes do *framework*. Ele é responsá-

vel por receber as mensagens e processá-las, extraindo as informações como agendamento de tarefas e envio de dados requisitados por alguma aplicação. Também é responsável por fazer o *bootstrap* do *framework* quando o nó é adicionado à rede, conectando o nó sensor ao controlador e passando a ele informações sobre as capacidades do nó.

3.4 Protocolo de interface com o usuário

Essa seção descreve o protocolo de interface entre o controlador e o usuário.

Através desta interface, será possível agendar tarefas para os nós sensores executarem. Será possível, também, solicitar informações da rede, dos nós sensores e das tarefas disponíveis, assim como os parâmetros necessários para o agendamento de cada tarefa.

Para garantir a interoperabilidade na comunicação, essa interação entre usuário e controlador será feita através de um protocolo REST. Dessa forma o usuário poderá realizar as solicitações ao controlador do *framework* independentemente da linguagem e da máquina que sua aplicação utilizar.

A seguir, encontram-se as solicitações disponibilizadas pelo servidor REST, bem como os parâmetros necessários e o formato de respostas de cada uma.

3.4.1 Características dos nós sensores

Método que recupera as características dos nós sensores, como o sistema operacional, dispositivo, localização do nó e tarefas disponíveis. É possível limitar a busca estabelecendo uma área para que, dessa forma, sejam retornados apenas os nós sensores dentro da região delimitada. Além disso, se for especificado o identificador do nó, apenas as informações deste nó serão retornadas.

```
GET /NODES
```

Parâmetros de entrada opcional:

- *node_id*: Inteiro que representa o identificador do nó sensor.
- *latitude*: Decimal que representa a latitude do centro da área para realizar a busca.
- *longitude*: Decimal que representa a longitude do centro da área para realizar a busca.
- *range*: Decimal que representa o raio da área para realizar a busca em metros.

Observação: Para realizar a busca por região é necessário fornecer os três parâmetros: *latitude*, *longitude* e *range*.

O método retorna uma lista contendo as características dos nós sensores. Cada item da lista apresenta os seguintes parâmetros:

- *node_id*: Identificador do nó sensor;
- *device*: Modelo do dispositivo (e.g. *TelosB*, *MicaZ*).Esse objeto possui o identificador do dispositivo, nome e descrição;
- *operating_system*: Sistema operacional em execução (e.g. *TinyOS*, *Contiki*). Esse objeto possui o identificador do sistema operacional, nome e descrição;
- *mobility*: Grau de mobilidade do nó. Estático: *false*; Móvel: *true*;
- *energy_autonomy*: Fonte de energia do nó. Contínua (i.e. rede elétrica): *false*; ou bateria: *true*;
- *latitude*: Latitude do nó;
- *longitude*: Longitude do nó;
- *height*: Altura do nó;
- *periodic_task_qtty*: Quantidade de tarefas periódicas;
- *max_periodic_task_qtty*: Quantidade máxima de tarefas periódicas;
- *data_task_qtty*: Quantidade de tarefas instantâneas;
- *max_data_task_qtty*: Quantidade máxima de tarefas instantâneas;
- *occupied_ram_percentage*: Porcentagem de memória RAM ocupada;
- *current_battery_level*: Atual nível de bateria;
- *tasks*: Lista de tarefas disponíveis no nó. A seguir, são apresentados os parâmetros de cada item desta lista de tarefas.

Cada item da lista de tarefas apresenta os seguintes parâmetros:

- *task_id*: Identificador da tarefa;
- *description*: Descrição da tarefa;
- *generates_data*: Indica se a tarefa gera dados;
- *controls_actuator*: Indica se a tarefa controla atuador;
- *aggregates_data*: Indica se a tarefa agrega dados;

- *data_sink*: Indica se a tarefa foi definida como tarefa sorvedouro, com dados destinados a sair da rede;
- *type*: Tipo da tarefa (*Instantaneous* ou *Periodic*). Esse objeto possui o identificador do tipo e a sua descrição;
- *currently_scheduled_task_instances*: Quantidade de instâncias dessa tarefa que se encontram agendadas;
- *max_scheduled_task_instances*: Quantidade máxima de instâncias dessa tarefa que podem ser agendadas.

3.4.2 Disponibilidade das tarefas

Método que retorna uma lista de tarefas disponíveis. É possível limitar a busca estabelecendo uma área, um identificador do nó sensor ou o tipo da tarefa. Além disso, se for especificado o identificador da tarefa, apenas as informações desta tarefa serão retornadas.

```
GET /TASKS
```

Parâmetros de entrada opcionais:

- *node_id*: Inteiro que representa o identificador do nó sensor.
- *task_id*: Inteiro que representa o identificador da tarefa.
- *type*: Inteiro que representa o identificador do tipo de tarefa. Tarefa instantânea: 1; tarefa periódica: 2.
- *latitude*: Decimal que representa a latitude do centro da área para realizar a busca.
- *longitude*: Decimal que representa a longitude do centro da área para realizar a busca.
- *range*: Decimal que representa o raio da área para realizar a busca em metros.

Observação: Para realizar a busca por região é necessário fornecer os três parâmetros: *latitude*, *longitude* e *range*.

Retorna uma lista contendo de tarefas disponíveis.

3.4.3 Parâmetros de agendamento das tarefas

Método que retorna os parâmetros necessários para o agendamento das tarefas. Dependendo da tarefa e do nó sensor os parâmetros necessários para agendamento podem mudar. Por exemplo, para agendamento de uma tarefa periódica, é preciso do período e duração da tarefa, já para o agendamento de uma tarefa de agregação de dados, como um somador, é preciso saber de onde vem os dados que se deseja somar e a quantidade de dados.

```
GET /PARAMETERS
```

Parâmetros de entrada obrigatórios:

- *task_parameter_id*: Inteiro que representa o identificador do parâmetro.
- *task_id*: Inteiro que representa o identificador da tarefa.

Retorna as características dos parâmetros necessários de entrada para o agendamento e as características dos parâmetros de saída.

- *task_id*: Identificador da tarefa;
- *task_parameter_id*: identificador do parâmetro;
- *description*: Descrição do parâmetro;
- *input*: Define se é parâmetro de saída ou de entrada;
- *max_qtty_per_task_execution*: Quantidade máxima que pode ser utilizada em uma única execução (e.g. quantos operandos em uma soma);
- *support_floating_point*: Define se parâmetro suporta o tipo de dado ponto flutuante;
- *support_fixed_point*: Define se parâmetro suporta o tipo de dado ponto fixo;
- *support_integer*: Define se parâmetro suporta o tipo de dado inteiro;
- *support_unsigned_integer*: Define se parâmetro suporta o tipo de dado inteiro sem sinal;
- *support_bit_array*: Define se parâmetro suporta o tipo de dado vetor de bits;
- *support_8bits*: Define se parâmetro suporta o comprimento de dados de 8 bits;
- *support_16bits*: Define se parâmetro suporta o comprimento de dados de 16 bits;
- *support_32bits*: Define se parâmetro suporta o comprimento de dados de 32 bits;
- *support_64bits*: Define se parâmetro suporta o comprimento de dados de 64 bits;

3.4.4 Agendamentos em execução

Método que retorna uma lista de agendamentos em execução. É possível limitar a busca fornecendo o identificador do nó ou da tarefa. Assim, apenas os agendamentos presentes neste nó ou desta tarefa serão retornados. Além disso, se for especificado o identificador do agendamento, apenas as informações deste agendamento serão retornadas.

```
GET /SCHEDULES
```

Parâmetros de entrada opcional:

- *node_id*: Inteiro que representa o identificador do nó sensor.
- *task_id*: Inteiro que representa o identificador da tarefa.
- *task_scheduling_id*: Inteiro que representa o identificador do agendamento.

Retorna uma lista contendo os agendamentos em execução. Cada item da lista apresenta os seguintes parâmetros:

- *task_scheduling_id*: Identificador do agendamento;
- *scheduling_instance_number*: Número da instância do agendamento;
- *task_id*: Identificador da tarefa;
- *node_id*: Identificador do nó;
- *execution_period*: Período de execução;
- *parameter_validity_period*: Período de validade para a utilização dos dados recebidos da rede para processamento por uma tarefa;
- *period_precision_millis_micro*: Indica se a precisão da tarefa é de milissegundos ou microssegundos.

3.4.5 Agendamento de uma tarefa

Método que agenda uma tarefa. Dependendo do tipo da tarefa os parâmetros necessários para agendamento podem mudar, por exemplo, para agendamento de uma tarefa periódica, é preciso do período e duração da tarefa, já para o agendamento de uma tarefa de agregação de dados, como um somador, é preciso saber de onde vem os dados que se deseja somar e quantidade de dados. A seguir, alguns desses parâmetros são apresentados. Para saber exatamente quais são os parâmetros necessários de cada tarefa usa-se o método GET /PARAMETERS/:TID descrito na subseção 3.4.3.

```
POST /SCHEDULES/
```

Parâmetros de entrada:

- *NID*: Identificador do nó onde a tarefa será agendada;
- *TID*: Identificador da tarefa a ser agendada no nó;
- *Period*: Período de execução da tarefa periódica;
- *Duration*: Duração para execução da tarefa periódica;
- *Address*: Endereço do nó para destino dos dados coletados, esse endereço não é obrigatório;
- *Ref*: Refêrencia para o agendamento desta tarefa. Se for necessário usar os dados coletados desta tarefa em outra, é possível usar essa referência. Por exemplo, usar os dados coletados da leitura de sensor de temperatura para alimentar um somador;
- *Data*: Referência para onde buscar os dados a serem usados como entrada de uma tarefa de agregação, como no exemplo do somador. Ela pode ser uma expressão de comparação, por exemplo *"tempValue > 10"*, nesse caso o valor só será utilizado se atender à restrição. A expressão de comparação deve ser separada da referência por espaço;
- *Quantity*: Quantidade de dados para usar, indicando, por exemplo, quantos dados devem ser utilizados em uma única soma.

Retorna uma mensagem confirmando o agendamento ou informando a falha.

A seguir é mostrado um exemplo de aplicação composta por dois agendamentos: no primeiro é feito o agendamento de uma tarefa periódica e no segundo é feito o agendamento de uma tarefa instantânea.

O primeiro agendamento solicita a instanciação de uma tarefa para ler o sensor de temperatura TS1 a cada 10 segundos por 1 hora. É feita atribuição da referência “tempSensor” para este agendamento.

```
POST /SCHEDULES/
Body = {
          NID: 5,
          TID: 3,
          Period: 10,
          Duration: 3600,
```

```
            Ref: 'tempSensor'
        }
```

Já o segundo consiste no agendamento de uma tarefa para calcular a média de 10 dados do sensor de temperatura TS1 e enviar o resultado para o endereço IP 194.66.82.12. É atribuída a referência "avgTempSensor" para o resultado da média.

```
POST /SCHEDULES/
Body = {
            NID: 6,
            TID: 9,
            Data: 'tempSensor',
            Quantity: 10,
            Address: 194.66.82.12,
            Ref: 'avgTempSensor'
        }
```

3.4.6 Cancelamento de um agendamento

Método que cancela o agendamento de uma tarefa. Dado o identificador do agendamento é possível cancelar o mesmo.

```
DELETE /SCHEDULES/:SID
```

Retorna uma mensagem confirmando o cancelamento ou informando a falha.

3.5 Protocolo de interface com o *middleware*

Essa seção descreve o protocolo de descrição, agendamento e encadeamento de tarefas, responsável por padronizar a interface entre o controlador do *framework* e o *middleware* de cada nó sensor da rede.

Os objetivos desse protocolo são, portanto, a obtenção de informações relacionadas a características, relatórios e descrições de um dispositivo e suas tarefas, além de agendar tarefas que ele deve executar. Com relação a esse último aspecto em especial, o protocolo faz bastante uso do conceito de fluxo proposto pelo paradigma SDN, fazendo referência a fluxos de dados que carregam informação de como transmitir e tratar pacotes de dados recebidos por um nó.

A seguir, serão listados cada um dos pares de requisição-resposta desse protocolo, bem como as informações manipuladas por cada um. Cada um dos campos presentes nas

tabelas especificando o formato dos pacotes é descrito juntamente a elas, com exceção de campos *"X"*, que representam bits de *padding*.

3.5.1 Descrição de características de um nó sensor

Pacote enviado pelo *middleware* de um nó sensor ao controlador do *framework*, com o objetivo de inscrever seus serviços e de informar as características do dispositivo. Como resposta, o *middleware* espera uma confirmação de inscrição.

Os pacotes de descrição de características e de confirmação de inscrição de um nó sensor têm a estrutura descrita, respectivamente, nas tabelas 1 e 2.

Tabela 1 – Estrutura do pacote de descrição de características de um nó sensor.

<table>
<tr><td colspan="8">0</td><td colspan="8">1</td><td colspan="8">2</td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td colspan="3">PID</td><td colspan="11">NID</td><td>M</td><td>E</td><td colspan="8">DID</td></tr>
<tr><td colspan="8">DID</td><td colspan="8">OS</td><td colspan="8">LOC - Latitude</td></tr>
<tr><td colspan="24">LOC - Latitude</td></tr>
<tr><td colspan="24">LOC - Longitude</td></tr>
<tr><td colspan="8">LOC - Longitude</td><td colspan="16">LOC - Altitude</td></tr>
<tr><td colspan="16">LOC - Altitude</td><td colspan="8">PTQ</td></tr>
<tr><td colspan="8">IDQ</td><td colspan="8">NTQ</td><td colspan="6">NTN</td><td colspan="2">X</td></tr>
<tr><td colspan="16">TID [x NTN]</td></tr>
</table>

Tabela 2 – Estrutura do pacote de confirmação da inscrição de um nó sensor.

<table>
<tr><td colspan="8">0</td><td colspan="8">1</td><td colspan="8">2</td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td colspan="4">PID</td><td colspan="20">CFI</td></tr>
</table>

As informações manipuladas por esses pacotes são listadas a seguir:

- **PID**: identificador (ID) do tipo de pacote do protocolo, no caso 0;
- **NID**: ID do nó sensor;
- **M** : Grau de mobilidade do nó: estático ou móvel;
- **E** : Fonte de energia: contínua (i.e. rede elétrica) ou bateria;
- **OS** : Sistema operacional em execução (e.g. *TinyOS*, *Contiki*);
- **DID**: Modelo do dispositivo: ID do modelo (e.g. *TelosB*, *MicaZ*), ou 0 se não tiver um modelo padrão, como o caso de um arduino;
- **LOC**: Localização do nó sensor: coordenadas *Global Positioning System* (GPS) do nó sensor (latitude, longitude e altitude) expressas em ponto fixo;

- **PTQ**: Quantidade máxima de tarefas periódicas que pode ser agendada;
- **IDQ**: Quantidade máxima de dados que podem ser recebidos da rede como entrada para tarefas agendadas;
- **NTQ**: Quantidade máxima de tarefas que podem enviar dados à rede;
- **NTN**: Número de tarefas distintas carregadas no nó;
- **TID**: Para cada uma das tarefas carregadas no nó, um ID relacionando-a à sua descrição;
- **CFI**: ID do fluxo SDN que leva as respostas de protocolo do *middleware* até o controlador do *framework*.

É importante observar a restrição de um valor máximo de 31 para o campo NTN, de modo que o tamanho dos pacotes da requisição especificada na tabela 1 e da resposta especificada na tabela 16 não exceda 90 bytes, que é o limite de tamanho do *payload* de um quadro do IEEE 802.15.4, tecnologia de rádio preponderante na comunicação em RSSF. Um quadro de IEEE 802.15.4 tem tamanho de 127 bytes, subtraindo o tamanho do cabeçalho padrão e de informações de outras camadas, como a de SDN, restam cerca de 90 bytes.

3.5.2 Descrição de um dispositivo

A descrição das características de um nó sensor prevista pela seção 3.5.1 assume que este se trata de um dispositivo padrão, disponível no mercado, cujas características imutáveis podem ser armazenadas em um banco de dados. Essas características, portanto, não precisariam ser consultadas pelo controlador ao dispositivo via rede.

No entanto, o nó sensor pode não ser um dispositivo canônico, podendo ter sido montado, por exemplo, a partir de um arduino. Nesse caso precisamos de um par requisição-resposta para pedir informações mais específicas sobre um dispositivo. Contudo, considera-se que uma descrição detalhada de tais pacotes fuja ao escopo desse trabalho e, por isso, somente será feito aqui um levantamento de algumas características de dispositivos que merecem estar presentes em tal pacote:

- Características e quantidade de memória RAM;
- Modelo e características do processador;
- Modelo e características do rádio do nó sensor;
- Capacidades de sensoreamento e atuação;

- Número de fontes de energia distintas que se encontram disponíveis.

Mesmo não descrevendo-o em detalhes, reservamos o PID de valor 1 para esse par de pacotes requisição-resposta.

3.5.3 Descrição de uma tarefa

Requisição feita pelo controlador do *framework* ao *middleware* de um nó sensor, com o objetivo de obter as seguintes informações sobre uma determinada tarefa que pode ser desempenhada pelo dispositivo.

Os pacotes de requisição e resposta da descrição de uma tarefa têm a estrutura descrita, respectivamente, nas tabelas 3 e 4.

Tabela 3 – Estrutura do pacote de requisição da descrição de uma tarefa.

0								1								2							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
PID								TID															

Tabela 4 – Estrutura do pacote de resposta da descrição de uma tarefa.

0								1								2							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
PID				TID																TTI		X	
MSQ								ODF								IPN						X	
D	A	G	S	X												IPQ [0]							
IPF [0]								...								IPQ [IPN - 1]							
IPF [IPN - 1]																							

As informações inquiridas pelo controlador com esse tipo de requisição são as seguintes:

- **PID**: ID do tipo de pacote do protocolo, no caso 2;
- **TID**: ID da tarefa descrita;
- **TTI**: ID do tipo de tarefa descrita, podendo ser uma tarefa periódica, instantânea ou um *trigger*;
- **MSQ**: Quantidade máxima de agendamentos (i.e. instâncias) suportados pela tarefa;
- Descritores da terefa:
  - **D**: Bit indicando se a tarfa gera dados;

  - **A**: Bit indicando se a tarefa controla um atuador;
  - **G**: Bit indicando se a tarefa agrega dados;
  - **S**: Bit indicando se a tarefa atua como sorvedouro;

- Parâmetro de saída:
  - **ODF**: Tipos de dado e comprimentos suportados. Vetor de 10 bits em que cada bit representa uma possibilidade de tipo ou comprimento de dado aceito:
    0. *Floating point*;
    1. *Fixed point*;
    2. *Integer*;
    3. *Unsigned integer*;
    4. *Bit array*;
    5. *Boolean*;
    6. 8 bits;
    7. 16 bits;
    8. 32 bits;
    9. 64 bits;

- **IPN**: Número de parâmetros de entrada (i.e. operandos da tarefa);
- Descrição de cada parâmetro de entrada:
  - O ID do parâmetro, relacionando-o à sua descrição, é dado pelo *offset* de sua descrição em relação ao oitavo byte do pacote (i.e. o parâmetro de ID 0 é descrito nos bytes 8 e 9, o de ID 1 é descrito nos bytes 10 e 11, etc.);
  - **IPQ**: Quantidade máxima que pode ser utilizada em uma única execução (e.g. quantos operandos em uma soma);
  - **IPF**: Tipos de dado e comprimentos suportados. Vetor de 10 bits em que cada bit representa uma possibilidade de tipo ou comprimento de dado aceito:
    0. *Floating point*;
    1. *Fixed point*;
    2. *Integer*;
    3. *Unsigned integer*;
    4. *Bit array*;
    5. *Boolean*;
    6. 8 bits;
    7. 16 bits;
    8. 32 bits;
    9. 64 bits;

3.5.4 Agendamento de uma tarefa periódica

Requisição feita pelo controlador do *framework* ao *middleware* de um nó sensor, com o objetivo de agendar uma das tarefas de um fluxo de tarefas especificado pelo usuário. Uma tarefa periódica é aquela que deve ser executada uma vez em um certo intervalo de tempo. Como resposta, o controlador espera uma confirmação de que o agendamento foi efetuado com sucesso.

Os pacotes de requisição e resposta para o agendamento de uma tarefa periódica têm sua estrutura descrita, respectivamente, nas tabelas 5 e 6.

Tabela 5 – Estrutura do pacote de requisição do agendamento de uma tarefa periódica.

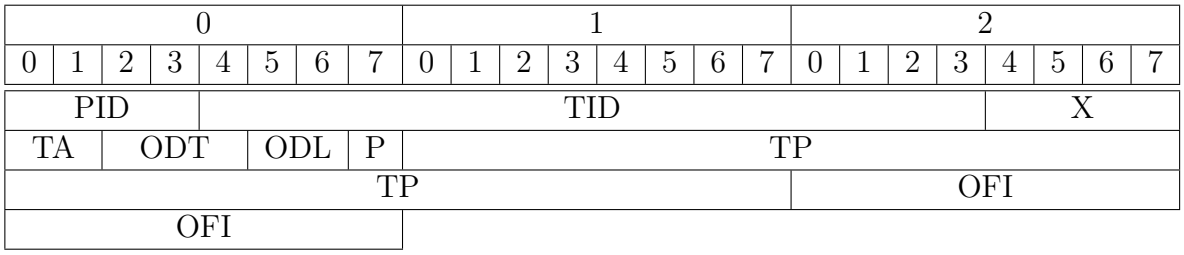
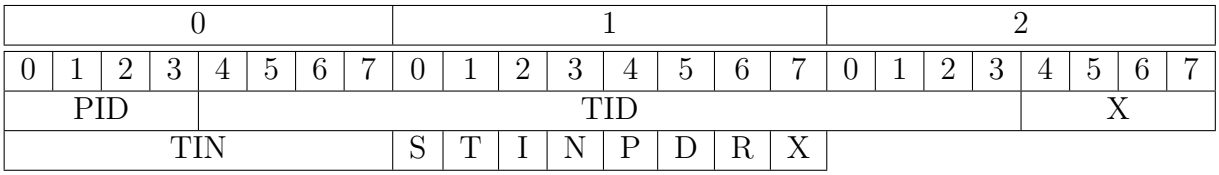


Tabela 6 – Estrutura do pacote de confirmação do agendamento de uma tarefa periódica.



As informações que o *middleware* necessita para o agendamento de uma tarefa periódica são as seguintes:

- **PID**: ID do tipo de pacote do protocolo, no caso 3;
- **TID**: ID da tarefa que se deseja agendar;
- **TA** : Acionamento por *trigger*:
  - 00: não é ativada por *trigger*;
  - 10: após ativação por um *trigger*, estará sempre ativa;
  - 11: cada nova execução necessita de um *trigger*.
- **P** : Precisão da periodicidade da tarefa:
  - 0: Se o período for de milissegundos;
  - 1: Se o período for de microssegundos.
- **TP** : Periodicidade da tarefa (valor do período ou 0 indicando execução única);

- Resultado produzido (se houver):
  - **ODT**: Tipo de dado do resultado produzido pela tarefa:
    - 000: *Floating point*;
    - 001: *Fixed point*;
    - 010: *Integer*;
    - 011: *Unsigned integer*;
    - 100: *Bit array*;
    - 101: *Boolean*;
  - **ODL**: Comprimento do dado do resultado produzido pela tarefa:
    - 00: 8 bits;
    - 01: 16 bits;
    - 10: 32 bits;
    - 11: 64 bits;
  - **OFI**: ID do fluxo de dados que levará o resultado ao seu destino;
- **TIN**: ID da instância da tarefa que corresponde ao agendamento realizado, que foi alocada pelo *middleware* para o agendamento realizado;
- **S** : Bit indicando sucesso da operação;
- Indicadores de erro de agendamento:
  - **T** : Bit indicando erro por não haver mais instâncias da tarefa disponíveis para agendamento;
  - **I** : Bit indicando erro por não haver mais capacidade de recepção de dados da rede como entrada da tarefa no nó sensor;
  - **N** : Bit indicando erro por não haver mais capacidade de envio de dados à rede no nó sensor;
  - **P** : Bit indicando erro por não haver mais capacidade de agendamento de tarefas periódicas no nó sensor;
  - **D** : Bit indicando erro por não haver mais capacidade de agendamento de tarefas instantâneas;
  - **R** : Bit indicando erro por não haver mais capacidade de agendamento de tarefas de trigger.

3.5.5 Agendamento de uma tarefa instantânea

Requisição feita pelo controlador do *framework* ao *middleware* de um nó sensor, com o objetivo de agendar uma das tarefas de um fluxo de tarefas especificado pelo usuário. Uma tarefa instantânea é aquela que pode ser executada assim que todos os seus parâmetros de entrada estiverem disponíveis. Como resposta, o controlador espera uma confirmação de que o agendamento foi efetuado com sucesso.

Os pacotes de requisição e resposta para o agendamento de uma tarefa instantânea tem a estrutura descrita, respectivamente, nas tabelas 7 e 8. Note que a resposta para o agendamento de uma tarefa instantânea tem o formato idêntico ao da resposta para o agendamento de uma tarefa periódica.

Tabela 7 – Estrutura do pacote de requisição do agendamento de uma tarefa instantânea.

<table>
<tr><td colspan="8">0</td><td colspan="8">1</td><td colspan="8">2</td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td colspan="3">PID</td><td colspan="13">TID</td><td colspan="4">IPN</td></tr>
<tr><td colspan="2">TA</td><td colspan="3">ODT</td><td colspan="2">ODL</td><td>P</td><td colspan="16">PTW</td></tr>
<tr><td colspan="24">PTW</td><td colspan="8">OFI</td></tr>
<tr><td colspan="8">OFI</td><td colspan="24">IFI [x IPN]</td></tr>
<tr><td colspan="8">PN [x IPN]</td><td colspan="16">ARG [x 31]</td><td colspan="8">0x00</td></tr>
</table>

Tabela 8 – Estrutura do pacote de confirmação do agendamento de uma tarefa instantânea.

<table>
<tr><td colspan="8">0</td><td colspan="8">1</td><td colspan="8">2</td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td colspan="4">PID</td><td colspan="16">TID</td><td colspan="4">X</td></tr>
<tr><td colspan="8">TIN</td><td>S</td><td>T</td><td>I</td><td>N</td><td>P</td><td>D</td><td>R</td><td>X</td></tr>
</table>

As informações que o *middleware* necessita para o agendamento de uma tarefa instantânea são as seguintes:

- **PID**: ID do tipo de pacote do protocolo, no caso 4;
- **TID**: ID da tarefa que se deseja agendar;
- **IPN**: Número de parâmetros de entrada (i.e. operandos da tarefa);
- **TA** : Acionamento por *trigger*:
  - 00: não é ativada por *trigger*;
  - 10: após ativação por um *trigger*, estará sempre ativa;
  - 11: cada nova execução necessita de um *trigger*.
- **P** : Precisão da janela de tempo máxima para a espera pela chegada de dados:

  - 0: Se o período for de milissegundos;
  - 1: Se o período for de microssegundos.

- **PTW**: Janela de tempo máxima para a espera pela chegada dos dados utilizados em uma execução;
- Para cada parâmetro de entrada:
  - **IFI**: ID do fluxo de dados que trará os valores para esse parâmetro de entrada;
  - **PN** : Número de dados desse tipo de parâmetro que devem ser utilizados em uma única execução.
- Resultado produzido (se houver):
  - **ODT**: Tipo de dado do resultado produzido pela tarefa, de acordo com a descrição da seção 3.5.4;
  - **ODL**: Comprimento do dado do resultado produzido pela tarefa, de acordo com a descrição da seção 3.5.4;
  - **OFI**: ID do fluxo de dados que levará o resultado ao seu destino.
- **ARG**: Sequencia de caracteres com comprimento máximo de 31 *bytes* que pode ser utilizada como parâmetro de configuração de uma tarefa instantânea. O último *byte* desse pacote deve ser nulo, limitando o tamanho máximo da *string* passada como parâmetro.

3.5.6 Agendamento de um *trigger*

Requisição feita pelo controlador do *framework* ao *middleware* de um nó sensor, com o objetivo de agendar uma das tarefas de um fluxo de tarefas especificado pelo usuário. Um *trigger* é um tipo de tarefa especial que tem o papel de monitorar dados coletados na rede, compará-los e acionar a execução de outras tarefas com base no resultado da comparação efetuada. Como resposta, o controlador espera uma confirmação de que o agendamento foi efetuado com sucesso.

Os pacotes de requisição e resposta para o agendamento de um *trigger* têm a estrutura descrita, respectivamente, nas tabelas 9 e 10. Note que a resposta para o agendamento de um *trigger* tem o formato idêntico ao da resposta para o agendamento de uma tarefa periódica.

As informações que o *middleware* necessita para o agendamento de um *trigger* são as seguintes:

- **PID**: ID do tipo de pacote do protocolo, no caso 5;

Tabela 9 – Estrutura do pacote de requisição do agendamento de um trigger.

0								1								2							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
PID				TID																TMC			
TSC		TCS			TRC			TMI															
TMI																							
TMI																							
TSI																							
TSI																							
TSI																TRI							
TRI																							
TRI																							
TRI								OFI															

Tabela 10 – Estrutura do pacote de confirmação do agendamento de um trigger.

0								1								2							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
PID				TID																X			
TIN								S	T	I	N	P	D	R	X								

- **TID**: ID da tarefa que se deseja agendar;
- **TMC**: Configuração do minuendo:
  - 000: constante;
  - 001: parâmetro de entrada;
  - 010: valor armazenado do subtraendo − regra: valor mais recente;
  - 011: valor armazenado do subtraendo − regra: valor maior que o atual;
  - 100: valor armazenado do subtraendo − regra: valor menor que o atual;
  - 101: valor armazenado da referencia − regra: valor mais recente;
  - 110: valor armazenado do referência − regra: valor maior que o atual;
  - 111: valor armazenado do referência − regra: valor menor que o atual.
- **TSC**: Configuração do subtraendo:
  - 000: constante;
  - 001: parâmetro de entrada;
  - 010: valor armazenado do minuendo − regra: valor mais recente;
  - 011: valor armazenado do minuendo − regra: valor maior que o atual;
  - 100: valor armazenado do minuendo − regra: valor menor que o atual;
  - 101: valor armazenado da referencia − regra: valor mais recente;

  - 110: valor armazenado do referência − regra: valor maior que o atual;
  - 111: valor armazenado do referência − regra: valor menor que o atual.

- **TCS**: Sinal de comparação utilizado (<, >, <=, >=, ==, !=, &&, ||);
  - 000: ==;
  - 001: !=;
  - 010: <;
  - 011: >;
  - 100: <=;
  - 101: >=;
  - 110: &&;
  - 111: ||.

- **TRC**: Configuração da referência:
  - 000: constante;
  - 001: parâmetro de entrada;
  - 010: valor armazenado do minuendo − regra: valor mais recente;
  - 011: valor armazenado do minuendo − regra: valor maior que o atual;
  - 100: valor armazenado do minuendo − regra: valor menor que o atual;
  - 101: valor armazenado da subtraendo − regra: valor mais recente;
  - 110: valor armazenado do subtraendo − regra: valor maior que o atual;
  - 111: valor armazenado do subtraendo − regra: valor menor que o atual;

- **TMI**: Valor de inicialização para o minuendo:
  - Valor numérico no caso de constante ou valor armazenado;
  - ID do fluxo de dados que trará os valores no caso de um parâmetro de entrada.

- **TSI**: Valor de inicialização para o subtraendo;
  - Valor numérico no caso de constante ou valor armazenado;
  - ID do fluxo de dados que trará os valores no caso de um parâmetro de entrada.

- **TRI**: Valor de inicialização para a referência;
  - Valor numérico no caso de constante ou valor armazenado;
  - ID do fluxo de dados que trará os valores no caso de um parâmetro de entrada.

- Sinal de *trigger* produzido:
  - **OFI**: ID do fluxo de dados que levará o sinal de *trigger* ao seu destino;

3.5.7 Cancelamento de uma tarefa agendada

Requisição feita pelo controlador do *framework* ao *middleware* de um nó sensor, com o objetivo de cancelar a execução de uma tarefa agendada para um fluxo de tarefas especificado pelo usuário. Como resposta, o controlador espera uma confirmação de que o cancelamento foi efetuado com sucesso.

Os pacotes de requisição e resposta para o cancelamento de uma tarefa tem a estrutura descrita, respectivamente, nas tabelas 11 e 12.

Tabela 11 – Estrutura do pacote de requisição do cancelamento de uma tarefa agendada.

0								1								2							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
PID				TID																TIN			
TIN																							

Tabela 12 – Estrutura do pacote de resposta do cancelamento de uma tarefa.

0								1								2							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
PID				TID																TIN			
TIN				S																			

As informações que o *middleware* necessita para cancelar uma tarefa agendada são as seguintes:

- **PID**: ID do tipo de pacote do protocolo, no caso 6;
- **TID**: ID da tarefa que se deseja agendar;
- **TIN**: ID da instância da tarefa que corresponde ao agendamento realizado, que foi alocada pelo *middleware* para o agendamento realizado;
- **S** : Bit indicando sucesso da operação.

3.5.8 Relatório de execução de uma tarefa agendada

Requisição feita pelo controlador do *framework* ao *middleware* de um nó sensor, com o objetivo de descobrir quantas vezes foi executada uma tarefa específica, agendada para um fluxo de tarefas especificado pelo usuário.

Os pacotes de requisição e resposta para a requisição do relatório de execução de uma tarefa agendada têm a estrutura descrita, respectivamente, nas tabelas 13 e 14.

As informações que o *middleware* necessita para identificar a tarefa cuja quantidade de execuções ele deverá retornar são as seguintes:

Tabela 13 – Estrutura do pacote de requisição do relatório de execução de uma tarefa agendada.

0								1								2							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
PID				TID												X							
TIN																							

Tabela 14 – Estrutura do pacote de resposta do relatório de execução de uma tarefa agendada.

0								1								2							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
PID				TID												X							
TIN								TEN															

- **PID**: ID do tipo de pacote do protocolo, no caso 7;
- **TID**: ID da tarefa que se deseja agendar;
- **TIN**: ID da instância da tarefa que corresponde ao agendamento realizado, que foi alocada pelo *middleware* para o agendamento realizado;
- **TEN**: número de execuções da tarefa consultada.

3.5.9 Estado de um nó sensor

Requisição feita pelo controlador do *framework* ao *middleware* de um nó sensor, com o objetivo de descobrir o uso dos recursos disponíveis no dispositivo.

Os pacotes de requisição e resposta para a requisição do relatório sobre o estado de um nó sensor tem a estrutura descrita, respectivamente, nas tabelas 15 e 16.

Tabela 15 – Estrutura do pacote de requisição do estado de um nó sensor.

0								1								2							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
PID																							

Tabela 16 – Estrutura do pacote de resposta do estado de um nó sensor.

0								1								2							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
PID				CPQ								CIQ								CNQ			
CNQ				X				CBL								CRP							
CL - Latitude																							
CL - Latitude								CL - Longitude															
CL - Longitude																CL - Altitude							
CL - Altitude																							
CSQ [x NTN]																							

As informações que o controlador pode obter são as seguintes:

- **PID**: ID do tipo de pacote do protocolo, no caso 8;
- **CPQ**: Quantidade atual de tarefas periódicas agendadas;
- **CIQ**: Quantidade atual de dados agendados para recebimento da rede como entrada de tarefas agendadas;
- **CNQ**: Quantidade atual de tarefas produzindo dados enviados à rede;
- **CBL**: Nível da bateria;
- **CRP**: Percentual de memória RAM ocupada.
- **CL** : Localização atual do nó sensor;
- **CSQ**: Para cada tipo de tarefa (ordenado por TID): quantidade atual de agendamentos;

3.5.10 Transmissão de dados

Requisição feita por qualquer nó sensor que deseje enviar, a outro dispositivo, o resultado produzido por uma tarefa. O nó sensor pode esperar uma confirmação do recebimento do dado pelo destinatário.

O pacote de transmissão de dados tem a estrutura descrita na tabela 17.

Tabela 17 – Estrutura do pacote de transmissão de dados.

0								1								2							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
PID				ODT			ODL	X								TD							
TD																							
TD																							
TD																							

As informações que o recipiente necessita para saber como encaminhar o dado recebido são as seguintes:

- **PID**: ID do tipo de pacote do protocolo, no caso 9;
- **ODT**: Tipo do dado transmitido, de acordo com a descrição da seção 3.5.4;
- **ODL**: Comprimento do dado transmitido, de acordo com a descrição da seção 3.5.4;
- **TD** : Dado a ser transmitido.

3.5.11 Transmissão de sinal de *trigger*

Requisição feita por qualquer nó sensor que deseje enviar, a outro dispositivo, o sinal de acionamento emitido por um *trigger*. O nó sensor pode esperar uma confirmação do recebimento do dado pelo destinatário.

O pacote de transmissão de sinal de *trigger* tem a estrutura descrita na tabela 18.

Tabela 18 – Estrutura do pacote de transmissão de um sinal de *trigger*.

0								1								2							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
PID																							

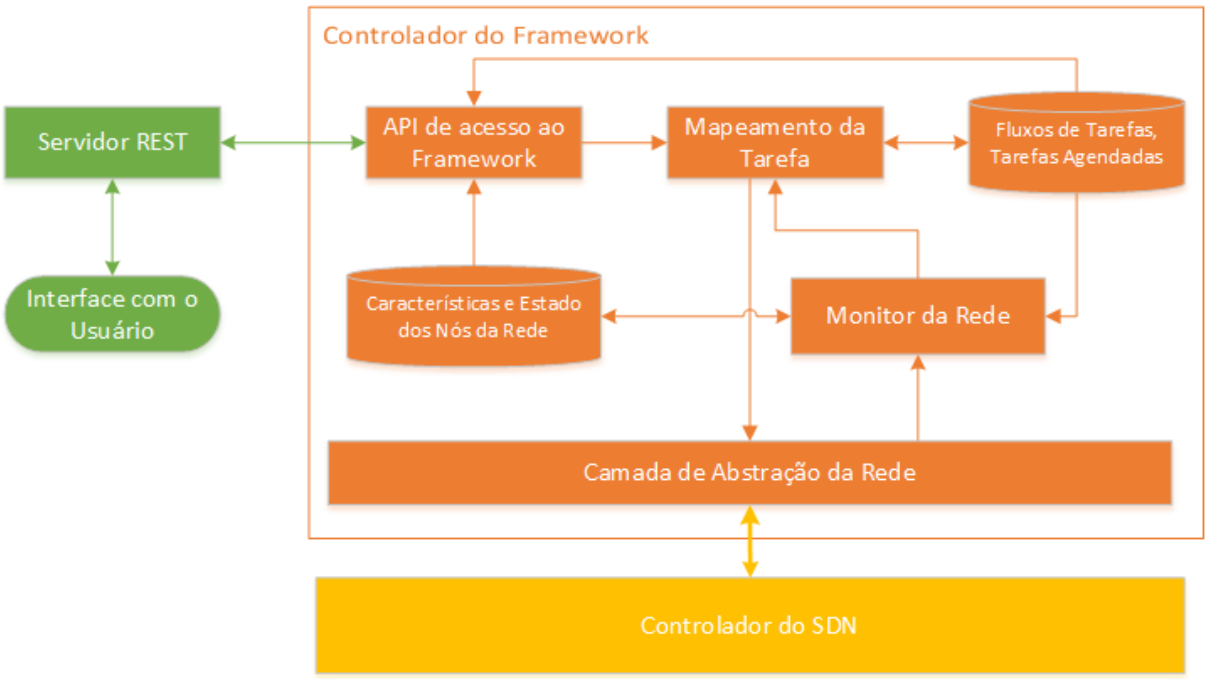
As informações que o recipiente necessita para saber como encaminhar o sinal recebido são as seguintes:

- **PID**: ID do tipo de pacote do protocolo, no caso 10;

3.6 Arquitetura do controlador

A figura 4 mostra a estrutura interna do controlador do *framework*, formulada a partir dos protocolos de comunicação que especificam suas interfaces. Essa estrutura evidencia cada um de seus módulos, que serão explicados ao longo dessa seção.

Figura 4 – Diagrama de arquitetura do controlador.



- `Servidor REST`: módulo externo ao controlador que faz uso de sua API para prover serviços a um usuário remoto.

- `API de Acesso ao` *`Framework`*: conjunto de métodos que implementa as funcionalidades previstas na seção 3.4, e que são disponibilizados ao usuário final por meio do servidor REST.

- `Mapeamento de Tarefas`: módulo responsável por:

  – Quebrar um fluxo de tarefas descrito pelo usuário, no formato previsto pelo protocolo descrito na Seção 3.4, em uma sequência de tarefas no formato previsto pelo protocolo descrito na Seção 3.5;

  – Alocar cada uma das tarefas em um nó sensor, considerando eficiência e o balanceamento do uso dos recursos da rede;

  – Emitir requisições de agendamento ou de cancelamento das tarefas aos nós sensores escolhidos;

  – Armazenar as informações de mapeamento de tarefas agendadas na rede no banco de dados destinado a isso.

- `Monitor da Rede`: módulo responsável por receber alertas de eventos na rede, enviados ao controlador tanto por nós sensores quanto pelo controlador SDN. Esses alertas devem ser processados de maneira que o banco de dados que guarda as características e o estado dos nós da rede possa ser mantido sempre atualizado. Além disso, no caso de um alerta comunicando o desligamento de um nó sensor, esse módulo deve requisitar um novo mapeamento das tarefas que estavam agendadas no nó desligado.

- `Camada de Abstração da Rede`: módulo responsável pela interface com o controlador SDN, abstraindo todos os métodos de envio e recepção de mensagens através da rede, bem como o acesso a funcionalidades providas pelo controlador SDN, como a criação de fluxos SDN. O objetivo dessa camada é tornar os demais módulos independentes de uma implementação específica de um controlador SDN, garantindo assim a portabilidade do controlador do *framework*.

3.6.1 Bancos de dados do controlador

Como visto na Figura 4, o controlador possui dois bancos de dados para armazenamento de informações. A descrição deles é a seguinte:

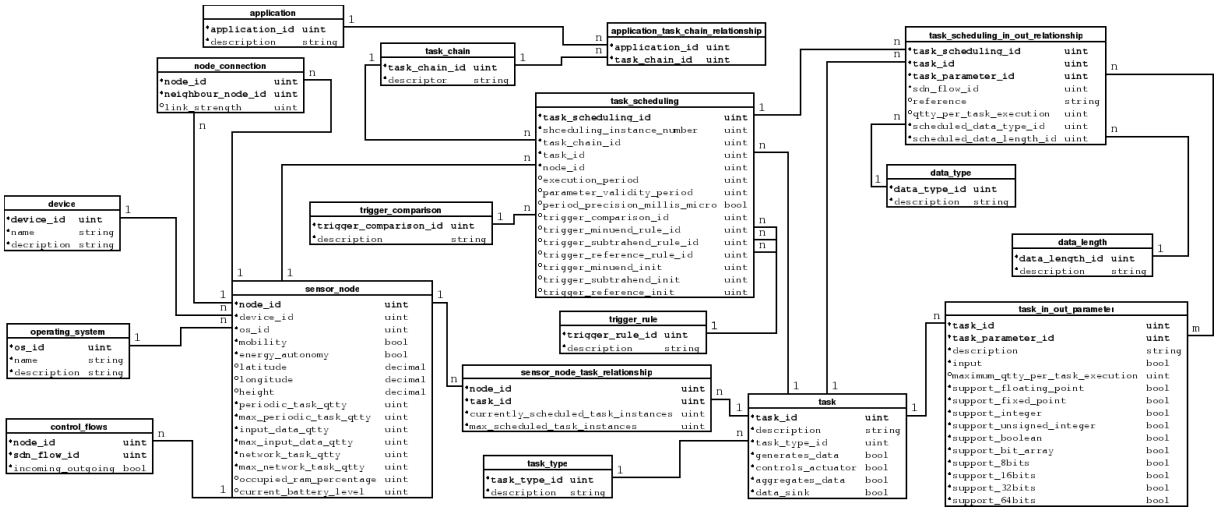
- `Características e estado dos nós da rede`: responsável por armazenar informações de topologia da rede (adjacências entre nós, bem como qualidade

das conexões) e características dos nós (grau de mobilidade, sistema operacional, ID do nó sensor, localização, tipo da fonte de alimentação, modelo do dispositivo, IDs das tarefas implementadas).

- `Fluxos de tarefas e tarefas agendadas`: armazena as informações das tarefas agendadas e em quais nós elas foram agendadas, bem como a ordem de execução entre elas, se houver.

A figura 5 é um diagrama Entidade-Relacionamento (ER) contendo a especificação de um modelo unificado desses dois bancos de dados. A opção de unificação dos dois bancos de dados na hora de modelá-los foi realizada devido à grande quantidade de relacionamentos entre as informações armazenadas nos dois.

Figura 5 – Diagrama ER do banco de dados do controlador.

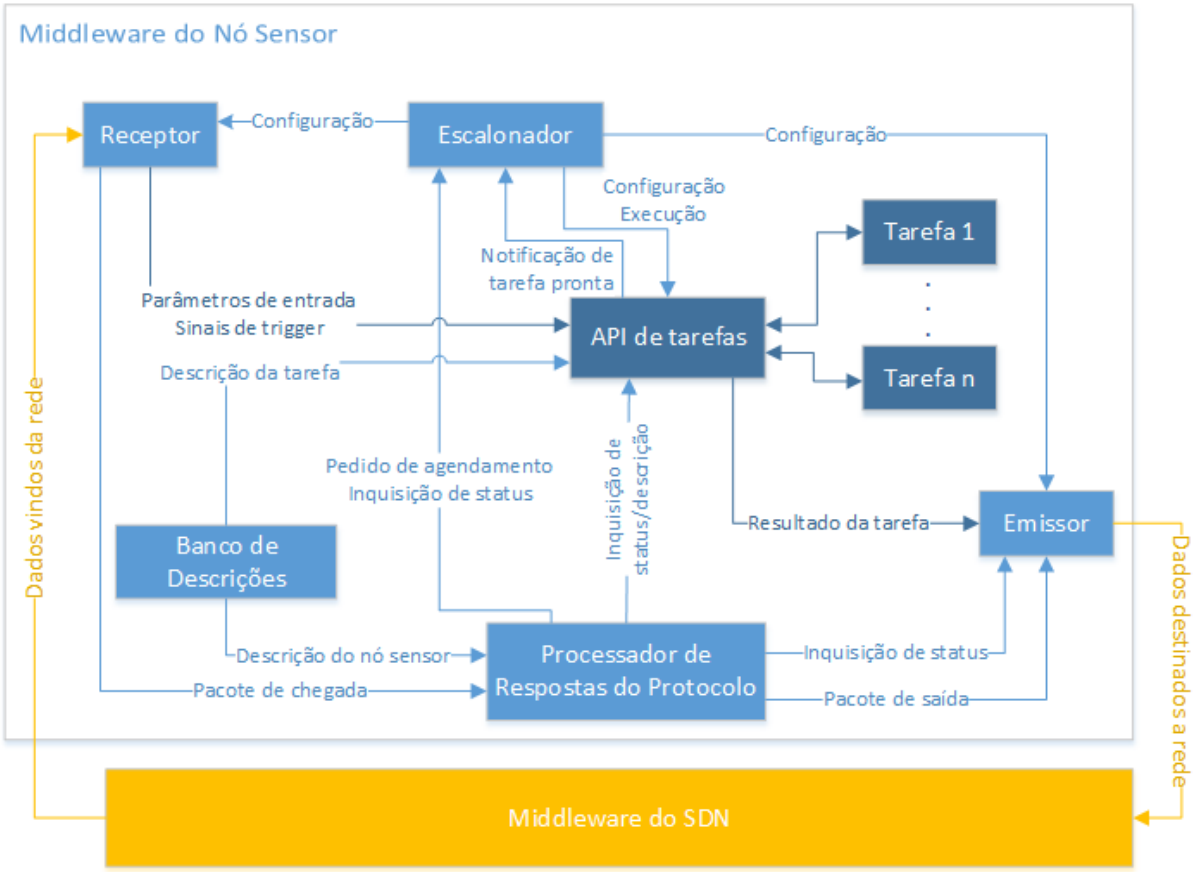


3.7 Arquitetura do *middleware*

A figura 6 mostra a estrutura interna do *middleware* presente nos nós sensores da rede, formulada a partir do protocolo de comunicação especificado na Seção 3.5. Essa estrutura evidencia cada um de seus módulos, que serão explicados ao longo dessa seção.

- `Receptor`: responsável por receber e desmontar pacotes vindos da rede, remetendo seu conteúdo ao módulo do *middleware* que deverá tratá-lo:
  - Parâmetros de tarefas devem ser enviados a tarefas;
  - Requisições do protocolo, como agendamento de tarefas e descrições de tarefas e de características do nó sensor devem ser enviadas ao Processador de Respostas do Protocolo.

Figura 6 – Diagrama de arquitetura do *middleware*.



- `Escalonador`: responsável por manter controle das tarefas agendadas:
  - Coordena a execução de tarefas periódicas, disparando-as nos instantes adequados;
  - Dispara as tarefas instantâneas que já tiverem todos os parâmetros necessários à sua execução, mantendo controle do tempo de expiração dos parâmetros recebidos por cada tarefa;
  - Ao agendar uma tarefa, configura os módulos receptor e emissor com seus parâmetros de entradas e saída, e pede ao Processador de Respostas do Protocolo para enviar uma confirmação de que a tarefa foi agendada;
  - Deve manter controle de quantas tarefas foram agendadas.
- `Processador de Respostas de Protocolo`: responsável por desmontar pacotes de requisição vindos do controlador do *framework* através da rede, de coordenar a realização das solicitações representadas por tais pacotes, através do acionamento dos demais componentes do *middleware*. Finalmente, também é responsável por receber os resultados das ações comandadas, e de empacotá-los para enviar as respostas aos pedidos feitos pelo controlador.

- `Emissor`: responsável por receber um dado produzido por uma tarefa, montar um pacote de dados com ele, e enviá-lo à rede em direção ao seu destinatário.

- `Tarefas`: são abstrações de aplicações de um nó sensor da rede, implementadas de maneira a utilizar os recursos do nó com o objetivo de realizar uma tarefa ou ação específica, como a coleta de dados, o acionamento de um atuador ou a agregação de dados coletados na rede. Em geral, elas podem possuir vários parâmetros de entrada, um parâmetro de saída e diversos tipos de configurações, de acordo com a especificação do protocolo na seção 3.5.

- `API de Tarefas`: responsável por abstrair o restante da estrutura do *middleware* para a implementação das tarefas a serem executadas pelo nó sensor. Seu papel é facilitar a implementação de tarefas, possibilitando ao programador que se concentre apenas nos parâmetros recebidos pela tarefa, nos recursos disponibilizados pelo nó sensor e em como programar a rotina que executa o objetivo de uma tarefa − tudo isso sem se preocupar com todo o gerenciamento que é feito pelo escalonador ou com o empacotamento e o desempacotamento de dados realizados pelo emissor e pelo receptor.

4 Tecnologias

Este capítulo descreve as tecnologias que foram consideradas para a implementação deste trabalho. As vantagens e desvantagens de cada uma das tecnologias consideradas são discutidas e as decisões tomadas são descritas.

4.1 Tecnologias de interface com o usuário

Duas tecnologias de interface com o usuário foram consideradas para a implementação no projeto. Elas são REST e SOAP e são descritas e analisadas a seguir.

4.1.1 REST

Transferência de Estado Representacional (em inglês, *Representational State Transfer* - REST) (FIELDING; TAYLOR, 2000) é um estilo de arquitetura para projetar aplicações de rede, que substitui mecanismos mais complexos de conexão como *Common Object Request Broker Architecture* (CORBA), *Remote Procedure Call* (RPC) e SOAP. Para realizar essa comunicação o REST usa o protocolo *HyperText Transfer Protocol* (HTTP) que é um protocolo cliente-servidor sem estado bastante utilizado para comunicação na web.

No sistema REST cada recurso é unicamente direcionado através da sua *Uniform Resource Identifier* (URI) e é possível realizar as operações CRUD (*Create*, *Read*, *Update* e *Delete*) solicitando os métodos *PUT*, *GET*, *POST* e *DELETE* do protocolo HTTP. Isso torna o uso de REST mais simples que as demais arquiteturas citadas e mais rápida também, pois não é necessário um processamento de dados extensivo.

4.1.2 SOAP

Protocolo Simples de Acesso a Objeto (em inglês, *Simple Object Access Protocol* - SOAP) (World Wide Web Consortium, 2015) é um protocolo de comunicação via Internet. Ele disponibiliza uma forma de comunicação entre aplicações que são executadas em diferentes sistemas operacionais com diferentes tecnologias e linguagens de programação. SOAP utiliza o formato XML para suas mensagens de solicitação e resposta e usa qualquer protocolo de transporte para transmissão destas mensagens, sendo os mais utilizados os protocolos HTTP e *Simple Mail Transfer Protocol* (SMTP).

O protocolo consiste de três partes, um envelope, que define a estrutura da mensagem; um conjunto de regras de codificação para representar as instâncias dos tipos definidos pela aplicação; e uma convenção para representar chamadas de rotinas e respostas.

4.1.3 Escolha e motivos

A princípio havia sido decidida a adoção do SOAP para a interface devido a sua maior versatilidade. Entretanto, após pesquisas a respeito de *frameworks* para o protocolo, foi notado que ele não é tão utilizado atualmente e que não existem boas implementações completas de SOAP.

Dessa forma, foi decidida a adoção do REST, pois ela é uma arquitetura muito popular e utilizada e que possui muitas implementações disponíveis, mas ainda com a capacidade de comunicação necessária para este projeto.

4.2 Tecnologias de implementação para o servidor REST

Para auxiliar na implementação do servidor REST optou-se pela utilização de um *framework web*, que é uma coleção de pacotes e módulos que permite que desenvolvedores escrevam aplicações web ou serviços web sem a necessidade tratar detalhes de baixo nível, como protocolos, *sockets* e gerenciamento de processos e/ou *threads*.

4.2.1 Django

Um dos *frameworks* de *full-stack* mais populares para *Python* é o *Django* (Django Project, 2015). Ele é um *framework* de código aberto que segue o padrão arquitetural *Model-View-Controller* (MVC). Seu objetivo é facilitar o desenvolvimento de sites complexos focados em bancos de dados.

Justamente pelo fato de *Django* ser um *framework web* completo, focado em sistemas complexos com modelo MVC, decidiu-se que ele não era o ideal para este projeto.

4.2.2 Flask

O *Flask* (Flask, 2015) é um *microframework web* também bastante popular para *Python*, que não é *full-stack*, mas que fornece a base de necessária para criar um servidor web. Ele é baseado nas tecnologias *Werkzeug* e *Jinja 2* e possui conformidade com *Web Service Gateway Interface* (WSGI) e solicitações REST. O *Flask* não força o desenvolvedor a utilizar nenhuma ferramenta ou biblioteca específica, ao invés disso ele possui suporte a extensões que adicionam funcionalidades às aplicações como se fossem nativas.

Devido a sua maior simplicidade e flexibilidade, o *Flask* foi considerado mais adequado para o projeto, pois dessa forma é possível implementar serviços web com REST de forma simples.

4.3 Linguagens para o controlador do *framework*

Algumas linguagens foram analisadas para a definição da melhor opção de linguagem para o controlador do *framework*. Elas são descritas e analisadas a seguir.

4.3.1 Go

Go (golang, 2015), frequentemente chamada também de *golang*, é uma linguagem de programação desenvolvida no Google em 2007. Sua sintaxe é fracamente derivada da linguagem C, mas a linguagem em si conta com a adição de *garbage collection*, *type safety* e algumas capacidades de tipagem dinâmica. *Go* já possui versões estáveis e suas futuras atualizações serão retrocompatíveis com as especificações atuais da linguagem.

Desenvolvida para ser usada em sistemas de larga escala, a linguagem possui compiladores extremamente rápidos, gerenciamento remoto de pacotes e também características de linguagens dinâmicas, como inferência de tipos. *Go* possui primitivas de processamento paralelo embutidas na linguagem e ferramentas que por padrão geram binários estaticamente ligados sem dependências externas. Faz parte da filosofia de criação da linguagem a ideia de que as especificações da linguagem devem ser simples o suficiente para que o programador consiga memorizá-las.

Suas vantagens incluem possuir maneiras simples e eficientes de paralelismo, além de eficiência por ser compilada. O sistema de gerenciamento de pacotes facilita a utilização de bibliotecas e ferramentas desenvolvidas por terceiros e sua comunidade é ativa. Dessa forma, ela possui diversas características favoráveis para um sistema que tem que lidar com múltiplas tarefas simultâneas.

Seu principal problema nesse projeto é a total ausência de experiência por parte dos integrantes do grupo com a linguagem.

4.3.2 Java

Java (Oracle Corporation, 2015) é uma linguagem de programação que foi lançada em 1996. A linguagem é compilada para *bytecode* que é executado pela *Java Virtual Machine* (JVM), de forma que o mesmo código compilado pode ser executado em qualquer sistema que possua uma JVM. *Java* é uma linguagem com suporte a concorrência, baseada em classes e orientada a objetos.

A linguagem foi desenvolvida para ser de uso geral, permitindo seu uso nos mais diversos tipos de aplicações. Ela possui uma vasta comunidade de desenvolvedores, sendo uma das linguagens mais populares do mundo. Por ser bem consolidada já possui uma sólida biblioteca padrão de implementações e também um grande número de materiais de referência na Internet a respeito dos mais diversos tipos de uso da linguagem. O desenvolvimento de aplicativos para o sistema operacional *Android* é feito em *Java*.

Suas principais vantagens incluem uma sólida adoção por parte da comunidade de desenvolvedores e ferramentas sólidas de desenvolvimento para a linguagem, como as populares *Integrated Development Environment* (IDE) *Eclipse* e *Netbeans*. O fato da linguagem possuir mecanismos embutidos de documentação de código também ajuda na colaboração de desenvolvedores no desenvolvimento de programas.

O principal problema de *Java* é o tamanho do código necessário para o desenvolvimento de *software* nesta linguagem, por sua sintaxe muito estrita. Tais características a tornam menos indicada para a rápida prototipagem de projetos, algo essencial neste trabalho devido ao curto tempo de implementação previsto no cronograma apresentado na seção 1.3. Outro problema é a ausência de possibilidade de utilizar outros paradigmas além da orientação a objetos.

4.3.3 Python

Python (Python Software Foundation, 2015) é uma linguagem de programação que surgiu em 1991. Sua filosofia enfatiza facilidade de leitura de código, de forma a permitir a utilização de um número menor de linhas de código do que outras linguagens como C++ e *Java*. *Python* tem suporte a diversos paradigmas de programação, incluindo orientação a objetos, funcional e procedural, é uma linguagem interpretada, portanto não existe a necessidade de recompilar o código após alterações.

A linguagem possui atualmente duas versões, *Python 2* e *Python 3*. Essa diferença existe pois a versão 3 não é retrocompatível com a 2. O *Python 3* foi lançado em 2008 e é considerado o presente e o futuro da linguagem, enquanto o *Python 2* é considerado legado. Entretanto, o suporte de bibliotecas ainda é mais extenso para a versão 2.

Python possui tipagem dinâmica, gerenciamento automático de memória e uma vasta biblioteca padrão. Diferentemente da grande maioria das linguagens, os blocos de códigos da linguagem são delimitados por indentação. A linguagem possui uma grande adoção em todo mundo, sendo uma das linguagens mais populares para ensino de programação devido a sua simplicidade. Graças a sua popularidade, ela possui um grande número de referências na Internet e muitos projetos são implementados em *Python*, permitindo encontrar exemplos de código para as mais diversas aplicações.

Suas vantagens incluem ser de fácil utilização, além de ser amplamente adotada

nas mais diversas áreas. *Python* possui sistemas de gerenciamento de pacotes com a disponibilidade de milhares de bibliotecas desenvolvidas por terceiros e sua comunidade é bem ativa. Essas características, somadas ao domínio da linguagem por parte dos integrantes do grupo, tornam-na uma opção atraente para o projeto.

O seu principal problema gira em torno de sua eficiência, pois ela é interpretada e portanto muito mais lenta do que linguagens compiladas. Ainda no quesito eficiência, seu interpretador não permite paralelismo verdadeiro durante a execução de múltiplas *threads* para manter a integridade de memória. Dessa forma ela não é a linguagem mais indicada em contextos multi-processados em que a eficiência é fundamental.

4.3.4 Escolha e motivos

Baseado nos pontos apresentados nessa seção, a linguagem *Python* foi escolhida para o controlador do *framework*. Ela é uma linguagem sólida e simples, dominada pelo grupo, permitindo a implementação de sistemas de maneira rápida. A escolha do *Flask* para o servidor REST também contribuiu para a decisão pelo uso da linguagem *Python*, já que ele também é escrito nessa linguagem.

4.4 Banco de Dados

Essa seção descreve um pouco sobre o sistema gerenciador de banco de dados utilizado e a ferramenta *Object-Relational Mapper* (ORM) escolhida para trabalhar com ele.

4.4.1 SQLite

SQLite (SQLite, 2015) é um sistema de gerenciamento de banco de dados relacional escrito em linguagem *C*. Ele é em grande parte autossuficiente, exigindo pouquíssimo suporte de bibliotecas externas e do sistema operacional. Outra vantagem do *SQLite* é que não há necessidade de comunicação com um servidor: se um processo solicita acessar o banco de dados, ele simplesmente lê e escreve diretamente nos arquivos do banco no disco. Dessa forma, não existe um servidor que precise ser iniciado e configurado; programas que usam o SQLite não precisam de um suporte administrativo para configurar o mecanismo do banco antes de serem executados.

Assim, o *SQLite* é ideal para este projeto, já que o banco de dados necessário é simples e seu sistema gerenciador deve ser leve para poder ser incorporado ao *framework*, que muitas vezes poderá ser executado em sistemas embarcados.

4.4.2 SQLAlchemy

O *SQLAlchemy* é um conjunto de ferramentas *open-source* de mapeamento objeto-relacional de bancos de dados SQL para a linguagem *Python*. Por se tratar de um dos ORMs mais populares para *Python*, o *SQLAlchemy* é compatível com os principais sistemas gerenciadores de bancos de dados relacionais, como *MySQL*, *Postgresql* e *SQLite*, o que o torna bastante flexível.

O uso do *SQLAlchemy* facilita a implementação de uma interface com banco de dados através de *Data Access Objects*(DAOs), pois torna desnecessária a escrita de qualquer linha em linguagem SQL. Ao programador, basta a criação de classes de mapeamento para as entidades em um diagrama ER, deixando para o *SQLAlquemy* a tarefa de gerar as tabelas SQL correspondentes de forma automática.

A abstração proporcionada pelo *SQLAlchemy* torna mínimas as alterações necessárias ao código desenvolvido no caso de uma mudança de plataforma. Dessa forma, outra das principais vantagens do seu uso é a portabilidade do código entre as diversas plataformas de sistemas gerenciadores de bancos de dados mencionadas.

4.5 Middleware

Essa seção descreve as tecnologias adotadas para a implementação do *middleware*, incluindo sistema operacional, suporte a SDN, plataforma de *hardware* para foco do desenvolvimento e ferramenta de simulação para testes. A escolha dessas tecnologias está fortemente condicionadas a vínculos de dependência do *TinySDN*, *framework* de SDN escolhido para utilização no projeto.

4.5.1 *TinyOS*

Como dito na subseção 2.1.1, o *TinyOS* é um SO de código aberto para RSSFs. Ele é orientado a eventos e programado em nesC, extensão da linguagem C, e é focado em baixo consumo de energia e operação em sistemas com processamento e memória limitados.

Como o *TinyOS* é um dos SOs mais utilizados pela comunidade acadêmica na área de RSSF e já possui todo um ecossistema − além de o *TinySDN* atualmente só dispor de implementação para esse SO − sua adoção foi considerada pertinente para a implementação do *middleware*.

4.5.2 *TinySDN*

O *TinySDN* (OLIVEIRA; MARGI; GABRIEL, 2014) é um *framework* de SDN para RSSFs e que atualmente é implementado sobre o *TinyOS*. Sua descrição mais deta-

lhada pode ser encontrada na subseção 2.4.3. Ainda assim, cabe dizer que ele é interessante para o projeto pois permite um controle centralizado da rede, inclusive com a possibilidade de múltiplos controladores.

O controle centralizado é uma característica crucial da arquitetura do projeto, pois é ele que permite que a rede seja facilmente gerenciável e vista como entidade única pelo usuário. O *TinySDN* torna-se então o componente responsável pela capacidade de abstração da topologia da rede ao lidar com a interação dos nós e fazer o roteamento das mensagens na rede.

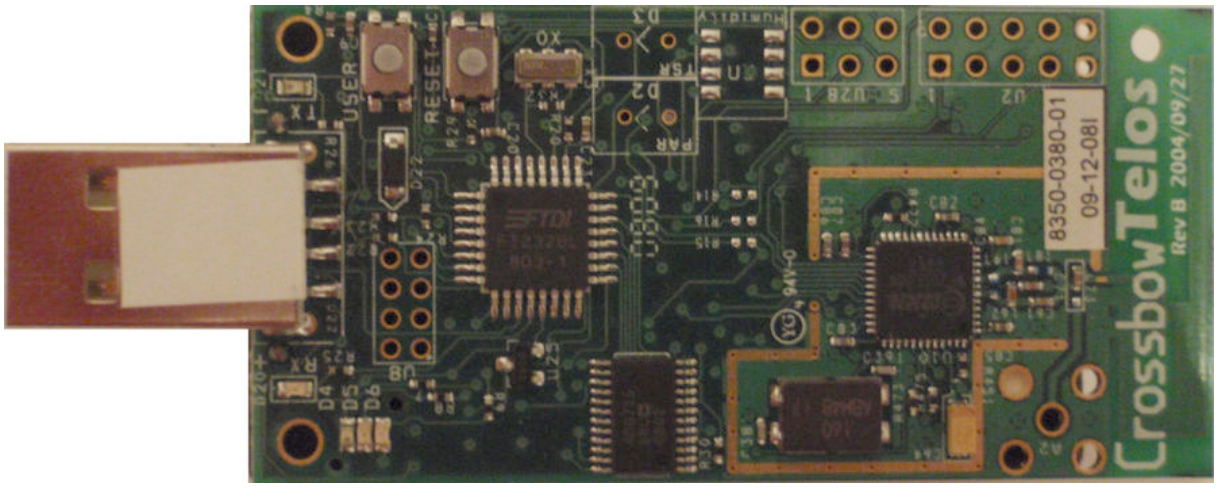
4.5.3 *TelosB*

O *TelosB* (TELOSB, 2015) é um nó sensor da *Memsic*, desenvolvido e publicado para a comunidade acadêmica pela Universidade da Califórnia Berkeley. A plataforma do *TelosB* tem código aberto e foi desenvolvida para permitir a realização de experimentos com RSSF pela comunidade científica.

Ele possui rádio compatível com o protocolo IEEE 802.15.4 e com o *ZigBee*, taxa de transferência de até 250kbps e antena integrada. Seu processador é um *TI MSP430* de 8MHz com 10kB de memória RAM e 48 kB de memória flash programável. Possui ainda programação via *Universal Serial Bus* (USB), baixo consumo de energia e a versão utilizada no projeto também possui sensores de umidade, temperatura e luminosidade, além de LEDs programáveis. Sua alimentação se dá através do conector USB ou através de duas pilhas do tipo AA.

O *TelosB* é compatível com o *TinyOS* e foi escolhido como foco inicial para a implementação do projeto. A figura 7 apresenta o modelo de *TelosB* utilizado.

Figura 7 – Foto do *TelosB* visto de cima.



Fonte: wikimedia.org

4.5.4 *COOJA*

O *COOJA* é um simulador de RSSF desenvolvido originalmente para o sistema operacional *Contiki*, mas que é capaz de simular nós sensores com código desenvolvido para *TinyOS* também.

Devido a sua facilidade de uso o *COOJA* foi usado para as simulações realizadas durante o desenvolvimento e testes do sistema ao invés do *TOSSIM*, que é o simulador desenvolvido para o *TinyOS*. Ele utiliza um simulador do nó sensor como um todo − processador, LEDs e chip de rádio − e permite a definição de localização espacial dos nós sensores, bem como as condições da comunicação (envio de mensagens com ou ou sem perdas, alcance do rádio, entre outros).

A utilização do *COOJA* é simples, basta criar uma nova simulação e adicionar nós com o código compilado da aplicação desejada. É inclusive possível adicionar nós com diferentes aplicações instaladas em uma mesma simulação.

Graças a sua agilidade de uso o *COOJA* foi a solução escolhida para a realização dos testes de unidade dos componentes do *middleware*. O fato dele também permitir a simulação de conexão serial foi fundamental para os testes de integração entre o controlador e o *middleware*.

Parte II

Aplicação

5 Desenvolvimento

Esse capítulo trata do desenvolvimento de uma implementação da especificação feita no capítulo 3 utilizando as tecnologias escolhidas no capítulo 4. Primeiramente são descritas as metodologias utilizadas para o desenvolvimento do projeto. Em seguida, são relatados os detalhes de implementação para os principais componentes do *WARM*: o servidor REST, o controlador do *framework* e o *middleware*.

5.1 Metodologias de desenvolvimento

Para o desenvolvimento do código para o *WARM*, procurou-se seguir boas práticas de programação tendo em vista a posterior divulgação do código-fonte para outros interessados, além de permitir uma melhor qualidade do produto final desenvolvido.

5.1.1 Versionamento de código

Para o desenvolvimento do projeto foi utilizado um sistema de versionamento de código. O sistema escolhido foi o Git (Git, 2015), sobretudo devido à familiaridade dos integrantes do grupo com esse sistema. Além da familiaridade dos integrantes, ele possui características interessantes para desenvolvimento realizado em equipe, sendo distribuído e facilitando a criação de ramos locais sem causar interferência com outros desenvolvedores até a hora de juntar o ramo local ao ramo principal.

Buscou-se fazer bom uso da facilidade de criação e união de ramos de código, definindo-se o uso de um ramo para cada área principal do *framework* (interface de usuário, controlador e *middleware*). Fora esses ramos existe ainda o ramo de desenvolvimento, para onde os ramos individuais são unidos quando há alterações significativas e por fim existe o ramo mestre, que só deve conter versões funcionais do código. Com esse modelo é possível garantir que o ramo principal sempre terá uma versão funcional e estável do código, permitindo o lançamento de *releases* periódicos, após terem sido devidamente testados no ramo de desenvolvimento.

5.1.2 Revisão de código

Para garantir a corretude do código e a boa qualidade do mesmo, foi adotada a metodologia de revisão por pares, do inglês *peer review*, onde o código escrito por um dos membros do grupo é revisado por ao menos um outro membro antes dele ser adicionado

ao repositório principal. Utilizando a metodologia de revisão por pares é possível detectar um número maior de erros no código antes de ele poder causar problemas maiores. Além de contribuir para o melhor conhecimento da base de código por todos os integrantes da equipe.

5.1.3 Testes de componentes

Para garantir que cada componente de código completado funcione de acordo com as especificações propostas, foi adotada a metodologia de testes de componentes. Segundo ela, cada componente funcional que estiver completo dentro do código é submetido a uma rotina de testes automáticos que avaliam o funcionamento correto de suas interfaces. Tal metodologia é bastante importante para garantir o funcionamento isolado dos componentes e, dessa forma, facilitar sua integração, além de facilitar a introdução de futuras modificações, garantindo que elas não comprometam o bom funcionamento de componentes já funcionais.

5.1.4 Documentação de código

O código é implementado com o objetivo de ser legível e auto-documentado, com comentários relevantes sobre seu funcionamento para facilitar o entendimento de um possível leitor.

No caso do controlador a documentação de funções, classes e métodos é feita seguindo o padrão definido pela PEP 257[1]. Esse padrão estipula um formato para as *docstrings* de descrição do código. Para gerar a documentação, foi utilizada a ferramenta *Sphinx* (Sphinx, 2015), que é capaz de fazer a geração da documentação em formato *HyperText Markup Language* (HTML) para visualização externa e para servir como referência das funções disponíveis sem necessidade de acessar diretamente o código.

Para o *middleware*, a documentação de módulos, configurações e interfaces é gerada através da ferramenta *nesdoc*, desenvolvida especialmente para a documentação de código na linguagem nesC. O padrão de documentação é baseado em blocos de comentários, de acordo com a especificação presente no manual da ferramenta *Doxygen* (HEESCH, 2015) − cujo uso é bastante popular para geração de documentação de código.

5.2 Servidor REST

O servidor REST foi criado para facilitar a utilização do *framework*. Com esta interface, o usuário pode solicitar informações da rede e agendar tarefas simplesmente

[1] `http://legacy.python.org/dev/peps/pep-0257`

utilizando o protocolo HTTP. Deixando-o livre para escolher a linguagem de programação e a plataforma mais apropriada para a sua aplicação.

Além de utilizá-lo, o usuário também tem a opção de realizar as solicitações através da API desenvolvida na linguagem *Python*. Os métodos disponíveis por meio da API são os mesmos que os disponibilizados pelo servidor REST, com parâmetros de entrada equivalentes. Entretanto, a saída fornecida pelas duas APIs é um pouco diferente: enquanto o servidor retorna objetos *JavaScript Object Notation* (JSON), a API em *Python* retorna um objeto dessa linguagem.

O servidor REST foi implementado utilizando o *microframework Flask*, descrito na subseção 4.2.2, que facilitou bastante o seu processo de desenvolvimento. Com o *Flask* é necessário apenas definir as rotas e o que será retornado para o usuário. Para a inicialização de um servidor para uma aplicação do *framework*, basta a execução do método "run()".

A seguir, está apresentada a maneira com que a rota "/nodes" foi desenvolvida. Analisando o código do método "get_nodes()", podemos notar que, primeiramente, os parâmetros de entrada recebidos são separados, de modo a serem fornecidos à API em *Python* do controlador. A API é chamada logo em seguida, de modo a retornar um resultado que necessita ser convertido para o formato JSON. É esse resultado convertido que é finalmente retornado ao usuário.

```
1  from flask import Flask
   from API import GetNodes, GetNodesJSON
3
   app = Flask(__name__)
5
   @app.route('/nodes', methods=['GET'])
7  def get_nodes():
       #Get parameters
9      node_id = str2float(request.args.get('node_id'))
       latitude = str2float(request.args.get('latitude'))
11     longitude =  str2float(request.args.get('longitude'))
       range = str2float(request.args.get('range'))
13
       #Call API
15     nodes = GetNodes(node_id=node_id, latitude=latitude, longitude=
       longitude, range=range)
       return jsonify({'nodes': GetNodesJSON(nodes)})
17


19 if __name__ == "__main__":
       app.run()
```

Essa forma de implementação torna o servidor REST totalmente desacoplado do controlador do *framework*. Ele utiliza diretamente a API desenvolvida em *Python*, da

mesma forma com que um usuário poderia optar por utilizá-la. As demais rotas "/tasks", "/schedules" e "/parameters", são implementadas de maneira similar à da rota "/nodes".

5.3 Controlador

O controlador é composto de múltiplas partes, cujas ideias e técnicas adotadas em sua implementação serão explicadas a seguir.

5.3.1 Mapeamento de tarefas

O mapeamento de tarefas é responsável por receber os pedidos de agendamento e cancelamento de tarefas e convertê-los em agendamentos de instâncias de tarefas na rede de sensores.

Existem muitas características que o controlador deve analisar ao receber um pedido de agendamento antes de criar a tarefa e os fluxos de dados entre os nós sensores. As principais dessas características são:

- O nó existe na rede?
- O nó é capaz de realizar a tarefa? Se sim, ainda pode agendar mais tarefas desse tipo?
- Já existe outro nó recebendo os dados de referência que essa tarefa deseja receber? Se sim, a tarefa não pode ser criada pois a referência só pode ser recebida por um nó devido à ausência de suporte do *TinySDN* a *multicast*.
- Já existem nós que desejam receber os dados da referência de dados gerados pela tarefa? Se sim, é preciso criar um fluxo ligando os nós.

Após determinar a resposta para essas perguntas, é necessário que o controlador consulte o banco de dados e obtenha as informações necessárias para criar os fluxos e agendar as tarefas. A maneira como uma tarefa é agendada depende do seu tipo, sendo diferente, por exemplo, se ela receber ou não dados vindos da rede como parâmetros de entrada.

5.3.2 Monitoramento da rede

Na implementação atual existem dois módulos distintos responsáveis pelo monitoramento da rede, um que se comunica com o controlador do *TinySDN* e o outro que se comunica diretamente com um nó sensor através de comunicação serial.

A necessidade de dois módulos diferentes advém do fato de que o controlador do *TinySDN* não dá suporte ao envio e recebimento de mensagens direto da rede. Essa restrição torna necessário que o controlador se comunique com a rede através de um nó dedicado, de modo a poder enviar e receber do *middleware* os pacotes do protocolo especificado na seção 3.5.

5.3.2.1 Controlador do *TinySDN*

A comunicação com o controlador do *TinySDN* é feita através de uma conexão por *socket* e é responsável por duas atribuições: criação de fluxos na rede e recebimento de informações sobre a topologia da rede.

A criação de fluxos é feita através da passagem de uma lista de arestas junto com o número do fluxo para o controlador do *TinySDN*. No controlador do *WARM*, o responsável pela criação de fluxos é o módulo mapeador de tarefas, que cria os fluxos para conectar nós que precisam realizar troca de informações.

A topologia da rede é passada pelo controlador do *TinySDN* através de uma lista de arestas que indica as conexões entre os nós existentes na rede. Quem chama a função de obtenção da topologia é também o módulo de mapeamento de tarefas, que utiliza as informações recebidas para decidir o caminho a ser seguido pelos fluxos.

5.3.2.2 Nó da rede conectado por serial

A comunicação direta com a rede − para o envio e recebimento de mensagens de controle do *WARM*, bem como de mensagens de dados com destino fora da rede − é feita através da comunicação serial com um nó sensor rodando o *TinySDN*.

A comunicação serial consiste de duas partes: um nó sensor rodando versão modificada do *TinySDN* capaz de realizar envio e recepção de dados através de comunicação serial; e um módulo do controlador do *WARM*, que gerencia a conexão serial e faz a interface entre o controlador e a conexão serial.

A programação do nó sensor transmite, através da comunicação serial, os *payloads* dos pacotes recebidos com destino ao controlador. Juntamente, são transmitidas também informações acerca do nó de origem e do comprimento do *payload* recebido. No outro sentido, esse nó sensor é capaz de receber comunicação serial de envio de pacotes para a rede. Para isso ele recebe o *payload*, comprimento do pacote e fluxo para o seu envio. A implementação desse nó de *gateway* foi feita utilizando como base o *TinySDN* em um nó sensor do tipo *TelosB*.

Por sua vez, o módulo de monitoramento da rede através de serial é implementado em *Python* e é o responsável pela interpretação dos *payloads* dos pacotes recebidos, bem como pela elaboração de *payloads* dos pacotes a serem enviados para a rede. Para

lidar com a comunicação serial em si, uma *thread* é criada pelo controlador para cuidar especificamente da conexão serial. A criação de pacotes e extração de dados dos mesmos é feita utilizando-se funcionalidades providas pelo *TinyOS* para a geração de código em *Python* que faça o processamento de pacotes.

5.3.3 Banco de dados

Para a implementação do banco de dados, foi utilizado o *SQLAlchemy*, como mencionado na subseção 4.4.2. Fazendo uso de suas ferramentas, foram escritas classes de mapeamento para as entidades no diagrama ER da figura 5. A partir de tais classes, as tabelas *SQLite* correspondentes foram geradas de forma automática.

Além das entidades, foi implementada uma classe DAO para tratar toda a conexão com o banco. Dessa forma, para realizar qualquer acesso ao banco é preciso apenas usar o método "GetSession()" do DAO. A seguir, é apresentada a implementação dessa classe.

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class DAO:
  """DAO"""
  def __init__(self):
    self.engine = create_engine('sqlite:///warm.db', echo=True)
    Base.metadata.create_all(self.engine)

  def GetSession(self):
    Session = sessionmaker(bind=self.engine)
    self.session = Session()
    return self.session
```

A seguir, vemos um caso simples que exemplifica o uso da classe DAO:

```
dao = DAO().GetSession()

#Para criar uma instancia de um no usamos a classe Sensor_Node.

node = Sensor_Node()
dao.add(node)
dao.commit()

#Para buscar todos os nos na tabela sensor_node usamos o metodo query

nodes = dao.query(Sensor_Node).all()
```

5.4 *Middleware*

Na seção 3.7, a arquitetura do *middleware* presente nos nós sensores foi especificada como uma série de componentes interconectados de acordo com o diagrama da figura 6. Tal estrutura é bastante condizente com o paradigma de programação adotado pela linguagem nesC, que condiciona o desenvolvimento de aplicações do *TinyOS* à implementação de módulos e suas respectivas configurações.

A linguagem dita que os módulos implementados devem utilizar ou prover interfaces, e que as configurações desses módulos devem relacionar as interfaces utilizadas com seus provedores. Tais interfaces, por sua vez, são constituídas basicamente pela especificação das assinaturas dos métodos que um módulo deve implementar quando provê uma determinada interface.

Dessa forma, um sistema é composto por uma configuração que relaciona os diversos sub-módulos implementados por meio de suas interfaces, o que faz com que a tarefa de implementar o *middleware* se torne basicamente o esforço de implementar cada um dos componentes da figura 6, e de especificar as interfaces constituídas pelas ligações entre tais componentes.

A seguir são descritos os principais detalhes de implementação desses componentes e suas interfaces.

5.4.1 Processador de Respostas do Protocolo

Esse componente recebe todos os pacotes de controle vindos do controlador do *framework* e aciona os demais de maneira que eles desempenhem as solicitações do controlador ou que providenciem os dados solicitados por ele. Dessa forma, uma de suas principais tarefas é desmontar os pacotes do protocolo especificado na seção 3.5 e identificar tais comandos e solicitações de dados.

Para isso, cada um dos pacotes do protocolo foi descrito através de estruturas de dados alinhadas, as *nx_structs* da linguagem nesC, utilizando o recurso *bit field*, também presente na linguagem *C*. Uma vez lidos os campos de um pacote, o Processador de Respostas do Protocolo precisa solicitar aos demais componentes, através das interfaces adequadas, as ações ou os dados correspondentes à solicitação do controlador. Em seguida, de posse da resposta enviada pelos módulos consultados, deve montar o pacote de resposta ao controlador, fazendo novamente uso das estruturas de dados que descrevemos.

5.4.2 Escalonador

Esse componente tem diversas funções relacionadas à execução de tarefas carregadas em um nó sensor, que podemos agrupar em dois conjuntos de funcionalidades: o

primeiro relacionado ao **agendamento** de uma tarefa, e o segundo relacionado ao comando de sua **execução** nos momentos convenientes.

No que diz respeito ao agendamento, que é acionado pelo Processador de Respostas do Protocolo, está sob a responsabilidade do escalonador desempenhar as seguintes funções no momento do agendamento de uma tarefa:

- Pedir à API de Tarefas que reserve uma instância da tarefa que está sendo agendada;
- Pedir ao Receptor que encaminhe parâmetros de entrada vindos da rede com determinados rótulos à tarefa que está sendo agendada, caso ela receba parâmetros de entrada;
- Pedir ao Emissor que empacote e envie os dados de saída produzidos pela tarefa que está sendo agendada, caso haja algum;
- Comunicar o sucesso ou o fracasso do agendamento, dependendo do resultado dos pedidos feitos anteriormente.

Para desempenhar tais funções, o Escalonador só precisa solicitar os pedidos descritos aos componentes adequados, utilizando as interfaces apropriadas.

No que se refere ao ao grupo de funcionalidades que estão relacionadas ao comando de execução de instâncias agendadas de uma tarefa nos momentos convenientes, há diversas variáveis cujo controle precisa ser mantido pelo Escalonador:

- Se a tarefa for do tipo periódica, é necessário comandar a execução de suas instâncias nos intervalos de tempo que correspondem ao seu período, e somente em tais instantes;
- Se a tarefa depender de parâmetros recebidos da rede, é necessário comandar a execução de suas instâncias somente quando essas tiverem à sua disposição todos os dados de que precisam para executar;
- No caso de a tarefa depender de parâmetros recebidos da rede, há ainda a necessidade de se verificar se os parâmetros que uma instância já tem disponíveis ainda não expiraram, isto é, perderam sua validade pelo fato de ter transcorrido um certo período de tempo desde que foram recebidos;
- Se a instância de uma tarefa precisar ser acionada por um *trigger*, é necessário saber se ela já recebeu o sinal de *trigger* que sinaliza a possibilidade de sua execução;

Para saber se a instância de uma tarefa está pronta para ser executada − isto é, dispõe de todos os parâmetros de entrada necessários ou foi acionada pelo sinal de *trigger*

adequado − o Escalonador recebe notificações da API de Tarefas, que as envia assim que uma instância de uma tarefa fica pronta. Se a instância de uma tarefa fica pronta e ela não é periódica, ela pode ser executada imediatamente.

Para tratar os casos de tarefas periódicas ou de tarefas cujos parâmetros de entrada recebidos têm tempo de validade, o Escalonador faz uso basicamente de duas filas: uma fila de instâncias de tarefas periódicas e uma fila de instâncias de tarefas cuja validade dos parâmetros irá expirar. Dispondo dessas duas filas, o componente programa dois *timers* (um para cada fila) em cujo evento de disparo ele irá extrair o primeiro da fila correspondente e realizar a ação apropriada: a execução da tarefa, no caso de uma instância de tarefa periódica pronta; ou o descarte dos parâmetros da instância, no caso de a validade dos seus parâmetros ter expirado.

Por fim, o Escalonador também é responsável por coordenar o cancelamento de um agendamento. Isso implica na remoção da instância da tarefa da fila em que se encontra, e da comunicação, aos demais componentes − Receptor, Emissor e API de Tarefas −, de que os recursos alocados por eles para o agendamento em questão podem ser liberados. Feito tudo isso, é necessário enviar a confirmação do cancelamento ao Processador de Respostas do Protocolo, para finalizar a ação de cancelamento.

5.4.3 API de Tarefas

Esse componente é responsável por abstrair as funcionalidades dos demais módulos da lógica de execução de uma tarefa. Para isso, seu papel constitui em tornar transparentes, para o programador de uma tarefa, as seguintes funcionalidades do *middleware*:

- A recepção de dados da rede, que devem se tornar disponíveis a uma tarefa programada sem que seja necessário checar sua procedência, fazer o seu desempacotamento ou identificar a que parâmetro de entrada um dado recebido se refere. O programador deve se preocupar basicamente em utilizar os parâmetros que estão disponíveis, com a consciência de que eles estarão disponíveis quando a tarefa for executada;

- O envio de dados à rede, que deve poder ser feito sem que seja necessário se preocupar com o destino do dado ou fazer o seu empacotamento de maneira adequada. O programador deve se preocupar basicamente com a sinalização do dado que ele deseja que seja enviado à rede;

- A notificação de que uma tarefa está pronta para ser executada pelo escalonador, algo que não deve ser uma preocupação do programador − que só deve se preocupar com o uso dos dados recebidos e não com sua disponibilidade;

- A alocação de instâncias de uma tarefa para um agendamento, cujo controle não deve ser uma preocupação do programador, que só precisa informar qual o número máximo de instâncias suportado por uma tarefa;
- A manutenção e o envio de relatórios de execução de uma tarefa;
- O envio da descrição de uma tarefa quando esta for solicitada − o que, no caso da presente implementação, já cumpre todos os papéis que seriam designados ao módulo "Banco de Descrições", mostrado na figura 6.

Para a realização de tais objetivos, a API de Tarefas serve como uma interface entre os demais componentes do *middleware* − que efetivamente realizam grande parte das funcionalidades listadas acima − e as tarefas carregadas em um nó sensor. Isso é feito com a disponibilização de funções, macros e componentes que implementam grande parte das funcionalidades de comunicação com os demais componentes do *middleware*. Isso inclui módulos que atuam como parâmetros de entrada da tarefa, gerenciando os dados que são recebidos para cada instância e notificando quando todos os dados necessários se encontram disponíveis.

Fazendo uso das ferramentas oferecidas pela API de Tarefas, o processo de programar uma nova tarefa se resume a instanciar os parâmetros de entrada da tarefa, declarar os componentes que provêm interface para os recursos do nó que se deseja utilizar, e implementar duas funções: uma de inicialização − chamada quando o nó sensor é ligado − e uma de execução − chamada quando o Escalonador a executa. Além de simplificar a estrutura do módulo de uma tarefa, essas ferramentas também incluem macros e funções que simplificam descrever uma tarefa, informar ao Emissor o resultado da tarefa e acessar os dados recebidos da rede que foram endereçados à tarefa.

O uso da API de Tarefas para a implementação de módulos de tarefas é demonstrado nos dois exemplos que se seguem.

Abaixo temos o exemplo da implementação de uma tarefa periódica que utiliza o conversor analógico-digital para amostrar dados de temperatura ambiente e depois enviá-los à rede.

```
1 module SenseTemperatureP {
    provides interface Task;
3   uses interface Read<uint16_t> as temperatureSensor;
}
5 implementation {

7   uint8_t currentTaskInstance;

9   PERIODIC_TASK_API_HELPER
```

```
    /**
     * @brief Task initialization routine.
     */
    void taskInit(void) {
        /** Initialize task description */
        INITIALIZE_TASK_DESCRIPTION(SUPPORT_INT |
                                    SUPPORT_2_BYTE_LENGTH, // int16_t
                                    OUTPUT_TO_NETWORK_TRUE,
                                    CONTROL_ACTUATOR_FALSE,
                                    AGGREGATE_DATA_FALSE,
                                    DATA_SINK_FALSE);
    }

    /**
     * @brief Task execution routine.
     */
    uint8_t taskRoutine(uint8_t taskInstance) {

        currentTaskInstance = taskInstance;
        call temperatureSensor.read();

        return 1;
    }

    /**
     * @brief Event signalizing ADC result is ready.
     */
    event void temperatureSensor.readDone(error_t result, uint16_t data) {
        OUTPUT_TO_NETWORK(currentTaskInstance, data);
    }
}
```

A seguir temos o exemplo da implementação de uma tarefa instantânea que recebe parâmetros vindos da rede como termos para o cálculo de uma média, efetua esse cálculo e em seguida envia o seu resultado à rede.

```
module AverageIntP {
    provides interface Task;

    uses {
        interface TaskParameterInput as input[uint8_t index];
        interface TaskArrayParameterAccess<TERM_INPUT_TYPE> as terms;
    }
}
implementation {

    INSTANTANEOUS_TASK_API_HELPER
```

```
13      TASK_OUTPUT_TYPE averageOutput;

15      /**
         * @brief Task initialization routine.
17       */
        void taskInit(void) {
19          /** Initialize task description */
            INITIALIZE_TASK_DESCRIPTION(SUPPORT_INT |
21                                      SUPPORT_2_BYTE_LENGTH, // int16_t
                                        OUTPUT_TO_NETWORK_TRUE,
23                                      CONTROL_ACTUATOR_FALSE,
                                        AGGREGATE_DATA_TRUE,
25                                      DATA_SINK_FALSE);

27          INITILIZE_TASK_INPUT_DESCRIPTION(TERM_INPUT_ID,
                                             NUMBER_OF_TERMS,          // size
29                                           SUPPORT_INT |
                                             SUPPORT_2_BYTE_LENGTH); // int16_t
31      }

33      /**
         * @brief Task execution routine.
35       */
        uint8_t taskRoutine(uint8_t taskInstance) {
37          uint8_t i;

39          uint8_t numberOfTerms = call terms.getInstanceSize(taskInstance);

41          /** Sum terms in order to compute average */
            averageOutput = 0;
43          for (i = 0; i < numberOfTerms; i++) {
                averageOutput += call terms.get(taskInstance, i);
45          }

47          /** Compute average by dividing sum result */
            averageOutput /= numberOfTerms;
49
            return 1;
51      }
}
```

5.4.4 Receptor

Esse componente é responsável por receber dados da rede e transmiti-los aos componentes do *middleware* a que eles se destinam. Só há dois destinos possíveis para um

pacote recebido: o Processador de Respostas do Protocolo, no caso de um pacote de controle (vide subseções 3.5.1 a 3.5.9), e a API de Tarefas, no caso de um pacote de dados (vide subseções 3.5.10 e 3.5.11).

Pacotes de controle podem ser transmitidos na íntegra para o seu componente de destino, mas pacotes de dados precisam ser mapeados para a instância da tarefa a que se destinam como entrada. Para isso, o Receptor mantém uma tabela que mapeia − para cada parâmetro de entrada da instância de uma tarefa − o identificador do fluxo SDN por meio do qual um pacote de dados encaminhado a ela poderá ser recebido.

Consultando essa tabela, o Receptor pode informar à API de Tarefas o dado recebido e a que parâmetro da instância de uma tarefa ele se refere. Essa tabela é atualizada quando o Receptor recebe comandos vindos do Escalonador, no momento do agendamento ou do cancelamento de uma instância de uma tarefa.

5.4.5 Emissor

Esse componente é responsável por enviar à rede os dados de saída produzidos pelos demais componentes do *middleware*. Só há duas possíveis origens de um dado que se quer destinar à rede: o Processador de Respostas do Protocolo, no caso de uma resposta a um pacote de controle, e a API de Tarefas, no caso de um dado produzido por uma instância de uma tarefa.

Pacotes de resposta ao controlador do *framework* podem ser simplesmente enviados à rede rotulados com o identificador do fluxo SDN que os levará até o controlador. Dados vindos da API de Tarefas, no entanto, precisam ser mapeados ao nó e à instância da tarefa a que se destinam. Para isso, o Emissor mantém uma tabela que mapeia − para cada instância de uma tarefa − o identificador do fluxo SDN por meio do qual o pacote carregando o dado chegará ao seu destino, além de outros detalhes relativos ao tipo e ao comprimento do dado enviado.

Consultando essa tabela, o Emissor descobre como montar e rotular o pacote de dados que irá transmitir, via rede, o dado produzido por uma instância de uma tarefa. Essa tabela é atualizada quando o Emissor recebe comandos vindos do Escalonador, no momento do agendamento ou do cancelamento de uma instância de uma tarefa.

6 Testes

Este capítulo descreve os testes realizados e resultados obtidos. Primeiramente, são descritos os testes realizados com o controlador do *framework*, de maneira a verificar seu correto funcionamento. Em seguida, são descritos os testes realizados com o objetivo de verificar o correto funcionamento do *middleware*. Depois da verificação do funcionamento isolado dos componentes do projeto, descreve-se como foram feitos os testes de integração entre o controlador do *framework* e o *middleware* presente nos nós sensores. Por fim, apresenta-se a demonstração de funcionamento do projeto *WARM* como um todo.

6.1 Testes do controlador

Esta seção apresenta os testes realizados com o objetivo de verificar o funcionamento correto do controlador.

6.1.1 Testes de componentes

Para a realização dos testes iniciais foi utilizada a metodologia de testes descrita na subseção 5.1.3, utilizando testes de unidade para testar as funcionalidades de cada novo componente e também testes de integração parcial sempre que possível para verificar a interação correta entre os módulos.

As unidades de teste são desenvolvidas de maneira separada do código principal, utilizando uma instância diferente do banco de dados, preenchida de acordo com as necessidades dos testes a serem realizados. O código dos módulos é também aparelhado com verificações especiais para emular funcionalidades de outros módulos quando necessário. Dessa forma, é possível isolar completamente o seu funcionamento e verificar a sua corretude de maneira independente dos demais módulos.

Os arquivos utilizados para testes foram mantidos no repositório de código do projeto e podem ser vistos e até mesmo utilizados como referência de uso das funções que eles testam.

6.1.2 Teste de funcionamento

Para realizar o teste de funcionamento completo do controlador também foi utilizado o simulador *COOJA*, descrito na subseção 4.5.4. Primeiramente foi criada uma simulação com apenas um nó sensor rodando uma versão modificada do controlador do

WARM. Essa versão modificada emula a presença de mais nós na rede, criando mensagens de descrição de nós e respondendo a mensagens enviadas pelo controlador.

Após os testes realizados com a versão modificada do nó sensor foram iniciados os testes de integração, descritos na seção 6.3.

6.2 Testes do *middleware*

Esta seção apresenta os testes realizados com o objetivo de verificar o funcionamento correto do *middleware*.

6.2.1 Testes de componentes

Como parte da metodologia de desenvolvimento, foram previstos testes de componentes para verificar o funcionamento correto de cada um dos módulos do *middleware* que são mostrados na figura 6. Essas unidades de testes foram elaboradas juntamente com o desenvolvimento dos componentes do *middleware*, de maneira a verificar que cada um funcionasse de forma isolada, a cada incremento de funcionalidade previsto na fase de implementação. Além disso, elas também colaboram para que a inclusão de novas funcionalidades nos componentes possa ser feita sem comprometer funcionalidades que já operem corretamente.

A implementação das unidades de teste de componentes é feita na forma de componentes complementares àqueles que se deseja testar, isto é: de componentes que implementam, de forma simplificada e controlada, as interfaces que o componente testado utiliza e que estimulam, com entradas variadas, as interfaces implementadas pelo componente testado. As ações das rotinas de testes e seus resultados são informados por meio da comunicação serial com o nó. De maneira a facilitar a realização dos testes, as unidades foram executadas fazendo uso da ferramenta de simulação *COOJA*, descrita na subseção 4.5.4.

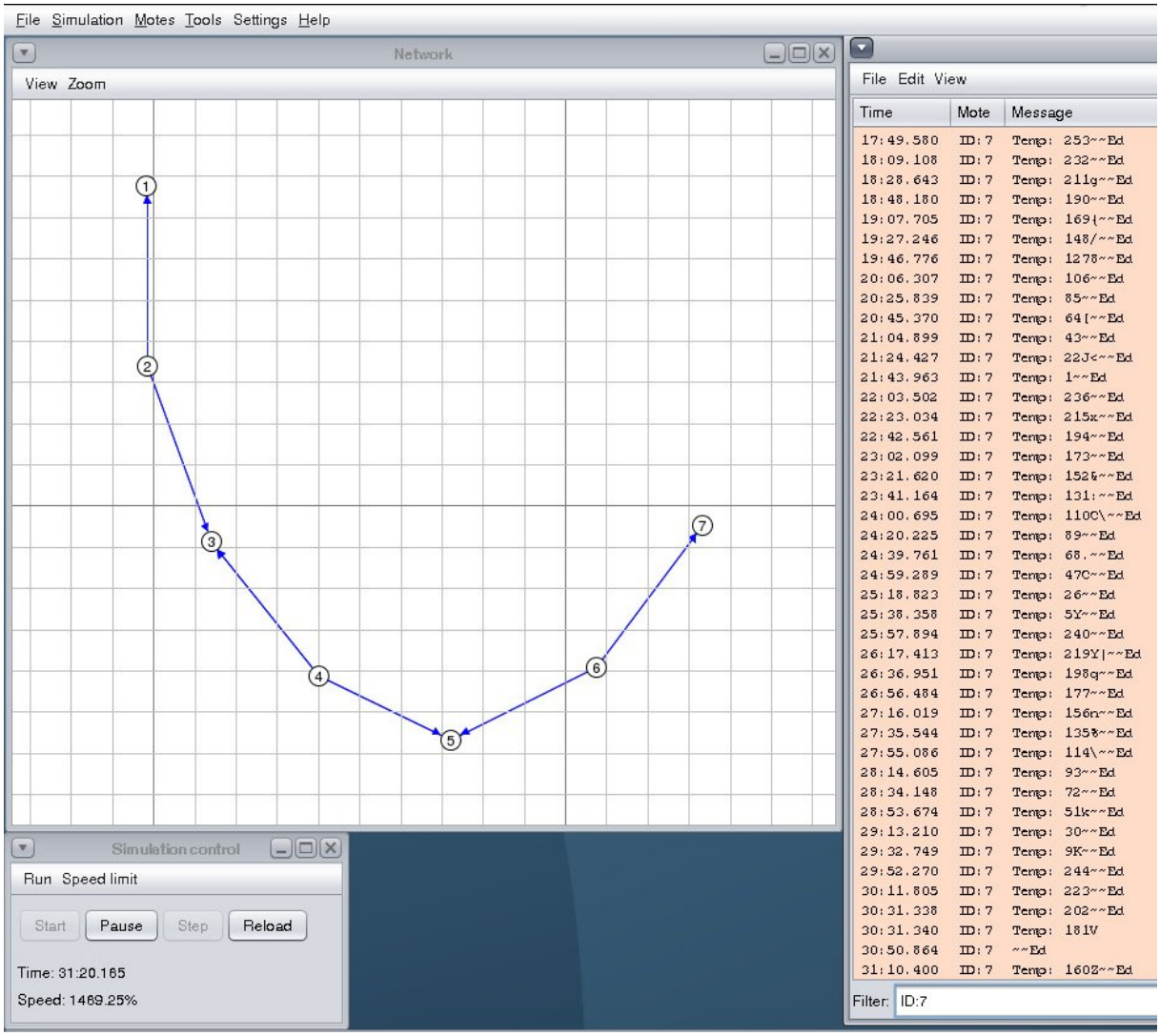
6.2.2 Teste de funcionamento

Para testar o funcionamento do *middleware* como um todo, de forma isolada do controlador do *framework*, foi realizado um *setup* de testes com sete nós sensores:

- Um, de endereço 1, contendo uma versão extremamente simplificada do controlador do *TinySDN*, que simplesmente configurava fluxos de dados entre três dos nós sensores, de endereços 3, 5 e 7;

- Um, de endereço 3, contendo uma versão extremamente simplificada do controlador do *framework*, que simplesmente enviava pacotes predeterminados do protocolo descrito na seção 3.5 para os dois nós sensores programados com o *middleware*;
- Dois, de endereços 5 e 7, programados com o *middleware* desenvolvido, ambos com somente duas tarefas carregadas, sendo uma periódica que amostra dados de temperatura e outra instantânea que recebe dados da rede e os "imprime" via serial;
- Três, de endereços 2, 4 e 6, programados somente com a camada do *TinySDN* de maneira a atuar como roteadores e confirmar o funcionamento de troca de informações na rede mesmo em cenários com múltiplos saltos.

Figura 8 – Teste do *middleware* realizado com o simulador *COOJA*.



Utilizando o simulador *COOJA*, foi criada uma simulação em que estes nós foram dispostos em fileira, de modo que cada um só pudesse se comunicar com os dois nós que lhe fossem adjacentes. Nela, o nó 3 agenda uma instância da tarefa que amostra temperatura

no nó 5, e no nó 7 uma instância da tarefa que recebe os dados de temperatura amostrados e os imprime via comunicação serial. Esse cenário pode ser visto na figura 8.

A execução desse teste comprovou o correto funcionamento do *middleware*, com o envio correto de respostas às requisições feitas pelo controlador simplificado, e também com o desempenho correto das tarefas agendadas.

6.2.3 Desempenho

A partir da execução do teste de funcionamento descrito na seção 6.2.2, foi possível obter resultados relacionados ao desempenho do *middleware*. Esses resultados podem ser vistos na tabela 19.

Tabela 19 – Tempos de execução do *middleware*

Funcionalidade	**Tempo aproximado (s)**
Boot de um nó sensor programado com o *middleware*	10,690
Associação de um nó sensor ao controlador	0,010
Requisição da descrição de uma tarefa	0,005
Agendamento de uma tarefa periódica	0,005
Agendamento de uma tarefa instantânea	0,005
Cancelamento de uma tarefa periódica	0,005
Cancelamento de uma tarefa instantânea	0,005
Requisição de relatório de execução de uma tarefa	0,005
Requisição de relatório de estado de um nó sensor	0,005

Dada a proximidade dos resultados obtidos para os tempos de requisição e resposta, pode-se inferir que o *overhead* de processamento das requisições pelo *middleware* é desprezível se comparado ao tempo da troca de mensagens na rede. É necessário lembrar que tais resultados consideram somente o tempo de resposta do *middleware*, não levando em conta os tempos de processamento que o controlador levaria para processar respostas do usuário, ou realizar ações como a inscrição de um nó associado antes de enviar uma confirmação.

Além disso, foi possível verificar o desempenho do *middleware* em termos de espaço em memória. O *middleware* programado nos nós sensores do teste realizado, carregado com somente duas tarefas, ocupava 34962 bytes de memória de programa e 5516 bytes de memória de dados. Em um nó sensor como o *TelosB*, descrito na seção 4.5.3 e utilizado nos testes realizados, isso significa uma ocupação de 72,84 % da memória de programa e de 55,16 % da memória de dados.

6.3 Testes de integração

Após a realização dos testes separados do *middleware* e do controlador foram realizados também testes de integração. Os testes de integração consistem de testes para garantir que o sistema inteiro interage corretamente entre as partes e está funcionando corretamente como um todo.

Os testes iniciais de integração também utilizaram o simulador *COOJA*, utilizando inclusive a mesma topologia de rede descrita na subseção 6.2.2. A única alteração na configuração de testes com relação ao utilizado nos testes do *middleware* foi a substituição da versão simplificada do controlador do *WARM*pela versão completa, com comunicação serial. Nesses testes ainda foi utilizada a versão simplificada do controlador do *TinySDN*.

A partir dessa simulação simplificada foi possível testar o correto funcionamento do sistema como um todo, desde o agendamento de tarefas, passando pela comunicação serial e indo até a correta comunicação entre o controlador e o *middleware*.

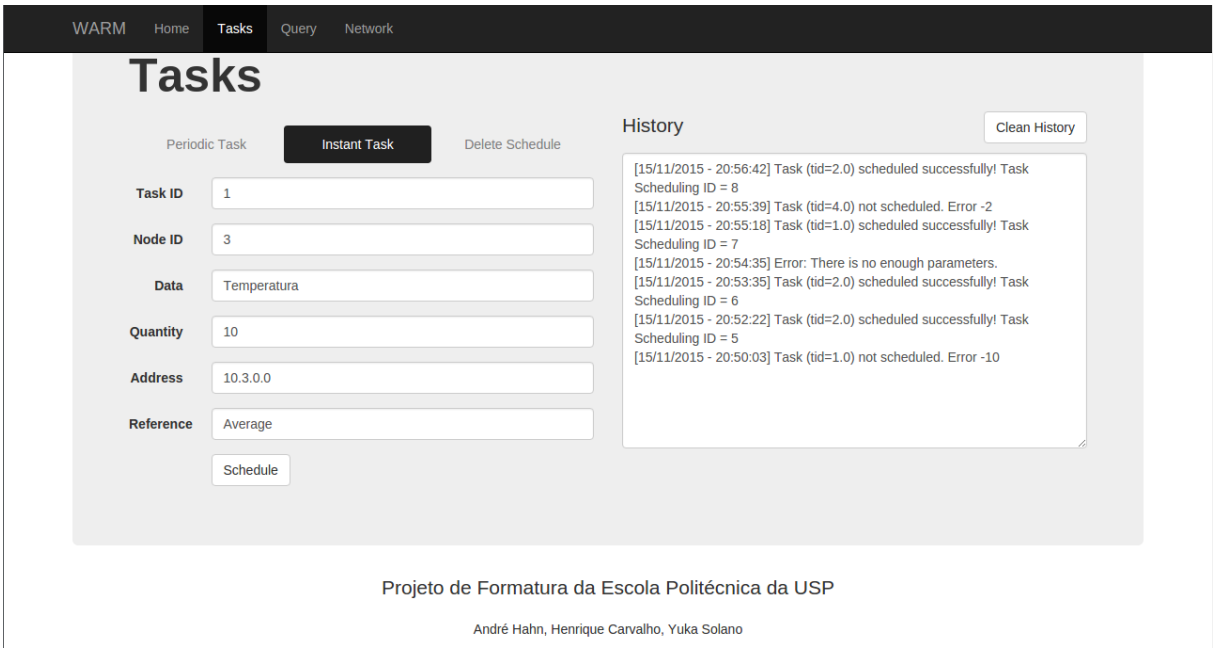
6.4 Demonstração de uso

Um exemplo de uso foi criado com o objetivo de demonstrar como o *framework* *WARM* pode ser utilizado de forma simples. O exemplo foi desenvolvido em *Javascript* e consiste em uma interface amigável ao usuário para realizar solicitações ao *framework* através da API disponibilizada. Há quatro abas nesta interface: 'Home', 'Tasks', 'Query' e 'Network'. Em 'Home', o usuário encontrará uma breve explicação sobre o *WARM* e um tutorial para o uso do sistema.

Em 'Tasks' é possível realizar o agendamento e cancelamento de todas as tarefas disponíveis na rede apenas preenchendo o formulário apresentado. Para agendar uma tarefa periódica é preciso fornecer o identificador a tarefa e do nó, o período, a duração, o endereço de destino dos dados coletados e uma referência, para caso o usuário queira usar esses dados coletados como entrada de outra tarefa, como por exemplo, uma tarefa que calcule a média desses dados. Para agendar uma tarefa instantânea, é preciso do identificador da tarefa e do nó, uma referência para os dados que serão usados como entrada, a quantidade desses dados, o endereço de destino da informação gerada e uma referência para essa informação.

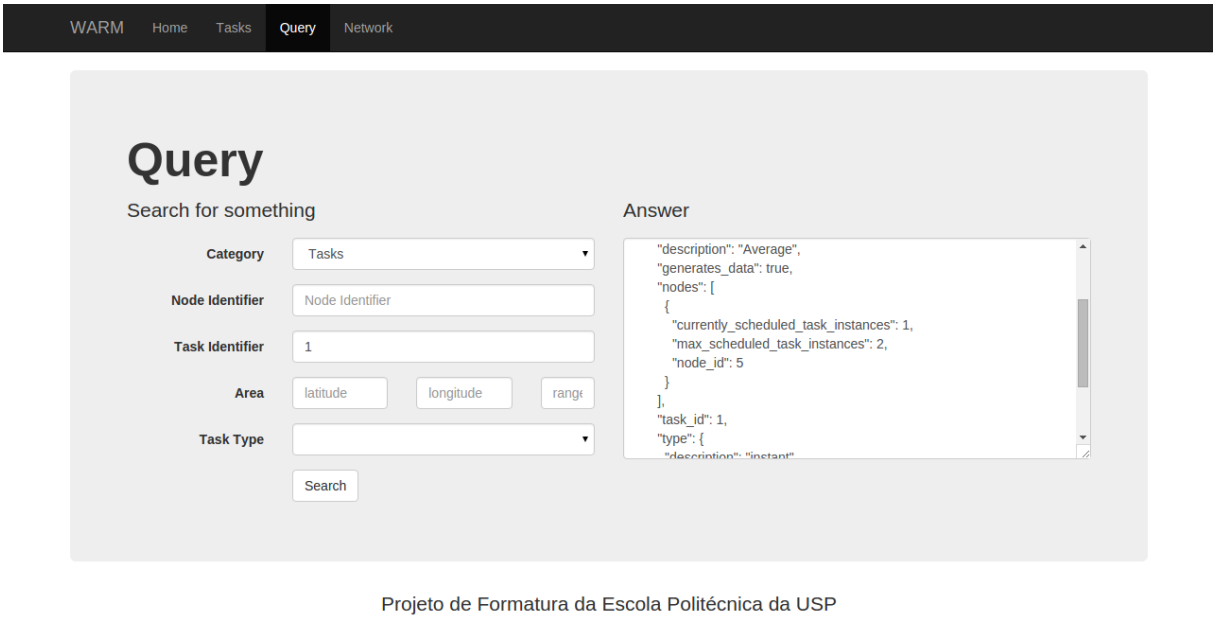
Em 'Query', o usuário pode solicitar informações da rede e receber o resultado em formato JSON. Há cinco categorias de buscas disponíveis: 'Nodes', 'Tasks', 'Schedules', 'Parameters', 'NodesStatistics' e 'TasksStatistics'. É possível restringir a busca preenchendo os campos disponíveis em cada categoria, por exemplo, o usuário pode fazer uma requisição das informações apenas dos nós que se encontram em uma especifica região fornecendo uma determinada área através da latitude e longitude de um ponto central e

Figura 9 – Aba 'Tasks' da interface desenvolvida.



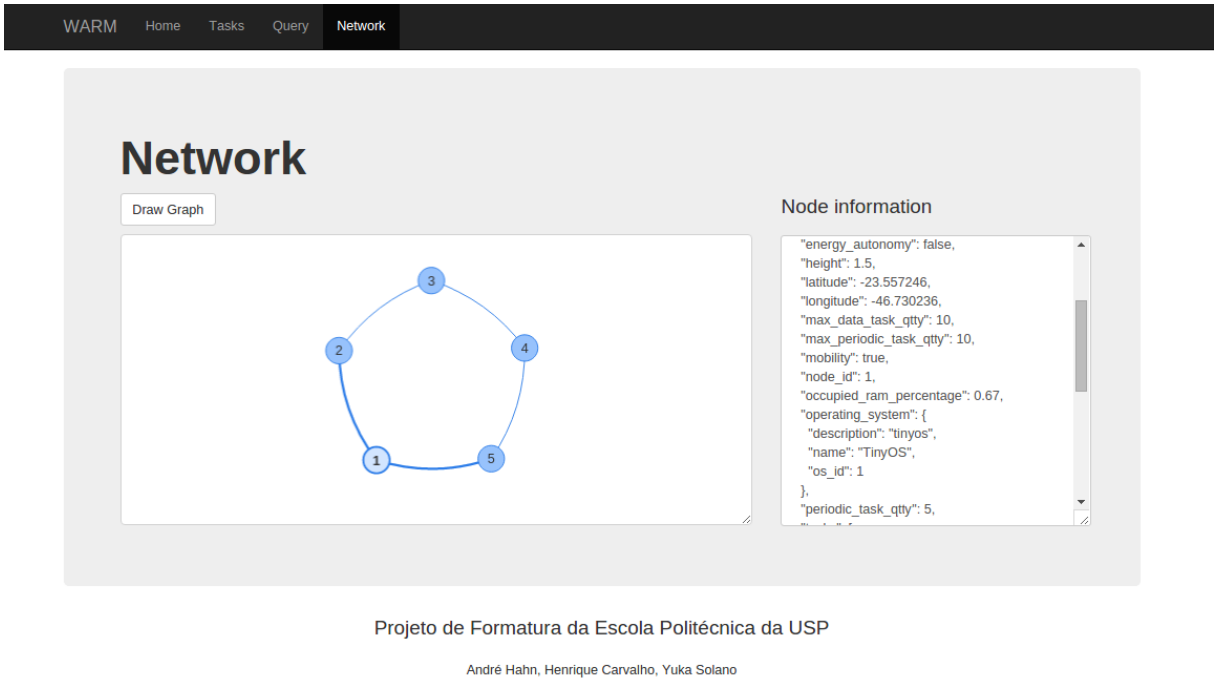
o raio para a busca.

Figura 10 – Aba 'Query' da interface desenvolvida.



Em 'Network' é possível visualizar como os nós estão distribuídos na rede de uma forma interativa para o usuário. Se um nó for clicado as suas informações aparecerão na barra lateral da interface.

Figura 11 – Aba 'Network' da interface desenvolvida.



7 Conclusão

Esse capítulo encerra este trabalho, retomando seus objetivos e discutindo como eles foram alcançados tendo em vista os resultados de seu desenvolvimento. Primeiramente, os resultados alcançados após término do projeto são relatados e discutidos em relação aos requisitos levantados. Em seguida, comenta-se acerca dos possíveis desdobramentos para o trabalho realizado, descrevendo algumas das possibilidades para lhe dar prosseguimento. Finalmente, são realizadas algumas reflexões acerca do que o projeto significou, de uma maneira mais ampla, enquanto trabalho final de graduação.

7.1 Resultados alcançados

Conforme apresentado na seção 1.1, o objetivo principal deste trabalho consiste na concepção, especificação, implementação, validação e análise de desempenho do *WARM*, um *framework open source* que facilita o desenvolvimento e o gerenciamento de aplicações em RSSFs. Com relação à concepção e à especificação desse *framework*, considera-se que o trabalho cumpriu seus objetivos, levantando requisitos e propondo uma solução conceitual bastante completa para o problema levantado, incluindo protocolos de comunicação, organização da arquitetura e armazenamento dos dados necessários para a sua operação.

Cabe comentar aqui que, devido à sua robustez, foram poucas as modificações realizadas sobre a especificação inicialmente proposta durante a fase de implementação, devido a alguma inviabilidade técnica encontrada nessa etapa de projeto. As poucas alterações que foram necessárias diziam respeito sobretudo a detalhes pontuais dos protocolos de comunicação descritos nas seções 3.4 e 3.5.

Com relação aos requisitos funcionais e não funcionais do projeto, levantados na seção 3.2, os testes realizados no capítulo 6 nos permitem afirmar que todos, com exceção do requisito não-funcional RNF6, foram atendidos. Entretanto, nem todos os itens presentes na especificação proposta no capítulo 5 foram implementados, sobretudo por falta de tempo e recursos. Para evidenciar como foi possível atender a maioria dos requisitos mesmo sem cumprir a risca o que estava na especificação, iremos discutir como a implementação realizada atende a esses requisitos e de que maneira os itens não implementados afetam esse atendimento.

O requisito funcional RF1 é atendido pela possibilidade que a API REST dá ao usuário de agendar múltiplas tarefas distintas compondo uma ou mais aplicações para

uma RSSF. O RF2 também é possível através da API REST, mas a localização informada para um nó sensor só será baseada em dados obtidos via GPS caso ele disponha de um sensor apropriado − o que não foi o caso da plataforma *TelosB*, utilizada nos testes, que informava uma localização pré-fixada. A API também atende o RF3, fornecendo informações armazenadas no banco de dados do controlador do *framework* e que são atualizadas em eventos de recepção de pacotes vindos do *middleware*, informando características dos nós, descrições de tarefas e informações de estado atual tanto dos nós quanto das tarefas executadas.

O RF4 foi atendido no que se refere à possibilidade do usuário de agendar tarefas através da API REST. No entanto, foram especificadas tarefas de três tipos: periódicas, instantâneas e *triggers*. Desses três, a implementação realizada só fornece suporte aos dois primeiros. Devido à falta de tempo e recursos, o suporte a tarefas do tipo *trigger* não foi completamente implementado tanto no *middleware* quanto no controlador do *framework*. Entretanto, é possível programar tarefas instantâneas que se comportem de maneira muito parecida com *triggers*, embora o seu mapeamento e agendamento através de rótulos e referências, conforme especificado na seção 3.4.5, não seja possível dessa forma.

A configuração das tarefas por meio da API REST também atende ao RF5, que possibilita a configuração de todos os parâmetros dos dois tipos de tarefas suportados, com exceção de um. O tempo de validade de parâmetros recebidos, via rede, de outras tarefas como entrada para tarefas instantâneas é um parâmetro de configuração que não é suportado pelas implementações atuais do *middleware* ou do controlador do *framework*. Com as mesmas restrições já observadas, o referenciamento e o encadeamento de tarefas também é suportado, satisfazendo o RF6. O último requisito funcional, RF7 é atendido devido à possibilidade de se agendar tarefas instantâneas apropriadas ao encerramento de uma cadeia de tarefas, dando destino aos dados coletados ao longo dela, como o envio dos dados à internet ou o seu registro em disco.

O requisito não-funcional RNF1 é atendido pelo uso do paradigma SDN, mais especificamente através do *TinySDN*, que centraliza o controle da rede de sensores. O *TinySDN* também possibilita o atendimento do RNF2, pois associa novos dispositivos à infraestrutura da rede automaticamente, mas o *middleware* presente nos nós sensores também tem um papel muito importante, fornecendo ao controlador as características e capacidades de um nó associado, de maneira que ele possa ser utilizado pelas aplicações em execução. Essas funcionalidades do *middleware* também são importantes para atender o requisito RNF3, uma vez que um usuário leigo não precisa se preocupar com a programação dos nós sensores que for utilizar − no caso de já estarem pré-programados com o *middleware* e já carregados com uma diversidade apropriada de tarefas.

A API de Tarefas, componente do *middleware*, também permite o fácil desenvolvimento de novas tarefas que utilizem novos recursos presentes em novas plataformas de

hardware, tornando o projeto extensível no que se refere ao requisito RNF4. Essa característica, aliada ao fato de o *middleware* ser baseado no SO *TinyOS*, contribui para a portabilidade do sistema − que pode ser transportado sem grandes dificuldades para outras plataformas de *hardware* suportadas pelo *TinyOS*, embora nenhum teste nesse sentido tenha sido feito −, de acordo com o requisito RNF7. O uso da linguagem *Python*, também altamente portável, na implementação do controlador é mais um fator que contribui para o atendimento desse requisito.

A interoperabilidade do *framework* projetado com outros sistemas é proporcionada pela sua API REST, arquitetura cuja popularidade e facilidade de uso contribuem para a fácil integração do *WARM* a outros sistemas, satisfazendo o RNF5. O mapeamento de tarefas, implementado pelo controlador, faz com que o sistema atenda o requisito RNF8, desde que uma entrada submetida pelo usuário não inclua o uso de uma tarefa de *trigger*, como já foi esclarecido acima. Por fim, apesar de o requisito não-funcional RNF6 não ter sido atendido por falta de recursos e tempo, várias das informações atualmente disponibilizadas pelo *middleware* ao *framework* possibilitam que seja implementada − para o módulo de monitoramento da rede presente no controlador − a funcionalidade de balanceamento automático dos recursos da rede.

Finalmente, além de comentar que a grande maioria dos requisitos do projeto foram atendidos de maneira satisfatória, é importante ressaltar que os resultados obtidos ainda mostram que *WARM* apresenta um tempo de resposta compatível com o que se espera de uma RSSF − considerando características inerentes a esse tipo de rede e seus componentes, como latência de comunicação e baixo poder de processamento − conforme observa-se pelos resultados dos testes no capítulo 6.

7.2 Trabalhos futuros

Na seção 7.1 foram mencionados alguns itens da especificação feita no capítulo 3 que não puderam ser implementados devido à falta de tempo e de recursos. A forma mais imediata de dar continuidade a este trabalho seria complementar a implementação de forma que ela esteja completamente de acordo com a especificação proposta.

Isso significaria incluir, na implementação atual, o suporte a tarefas de *trigger*, a possibilidade de configuração de uma janela de validade para os parâmetros de entrada de tarefas instantâneas e o balanceamento automático, pelo controlador, dos recursos alocados nos nós da rede. Concluída essa fase de melhoria da implementação, caberiam novos testes, para assegurar que os requisitos foram cumpridos e avaliar o desempenho do *framework* após tais modificações.

Completar a implementação de acordo com o especificado é, no entanto, somente uma das possibilidades para dar continuidade a este projeto. Uma etapa seguinte seria

a de otimização do *middleware*, para que ele ocupe um espaço menor em memória nos nós sensores, possibilitando que uma quantidade maior e mais diversificada de tarefas seja carregada em um nó. Além disso, seria importante escrever grande número de tarefas periódicas e instantâneas, que aproveitassem os recursos presentes no *TelosB*. Inclusive, seria interessante testar o sistema em outras plataformas de *hardware* suportadas pelo *TinyOS* e até portá-lo para outros sistemas operacionais e outras implementações de SDN para RSSF, de forma que ele dê suporte a um número ainda maior de nós sensores.

Finalmente, é necessário lembrar que, durante o recorte de escopo, documentado na seção 3.1, várias funcionalidades foram removidas antes de sequer serem incluídas nos requisitos ou na especificação. Dentre elas, merecem destaque:

- A inclusão de uma camada para a comunicação segura de dados entre os nós sensores − um requisito cada vez mais importante para RSSFs, a medida que elas passam a coletar dados de caráter privado, como em aplicações para o monitoramento de pacientes, residências, plantas industriais, entre outros;
- A viabilização da reprogramação remota das tarefas carregadas em um nó sensor através da API REST, que contribuiria para o aumento da flexibilidade da infraestrutura de uma RSSF instalada;
- A melhoria do balanceamento da carga nos nós sensores da rede, permitindo que o controlador seja capaz não somente de redistribuir tarefas quando necessário ou em caso de falhas, mas também carregar novas tarefas onde haja demanda.

A adição de alguma das funcionalidades descritas acima − incluindo sua especificação, implementação e teste − mereceria atenção especial em possíveis dobramentos futuros para este trabalho.

7.3 Considerações finais

A realização desse trabalho possibilitou o exercício de diversos conceitos e práticas com que os alunos de Engenharia Elétrica e Computação têm contato ao longo da graduação. O projeto e o desenvolvimento do sistema proposto, por si só, requisitou a aplicação de diversas técnicas aprendidas nas disciplinas de Engenharia de *Software*, como metodologias de desenvolvimento, levantamento de requisitos, modelagem de sistemas e de bancos de dados, metodologias de teste, técnicas de versionamento, documentação de código e diversos conceitos de arquitetura de *software*.

A execução do projeto demandou também sólidos conceitos de arquitetura de redes e de computadores, bem como conceitos de sistemas operacionais, relacionados a protocolos de comunicação, programação de sistemas, programação paralela e programação de

sistemas embarcados. Além disso, ela possibilitou o emprego de diversas técnicas observadas em laboratórios realizados durante a graduação, como Laboratório de Redes de Computadores, Laboratório de Fundamentos de Engenharia de Computação, Laboratório de Programação e Laboratório de Processadores.

Finalmente, cabe aqui destacar outros aspectos que também foram alcançados com o desenvolvimento desse trabalho, que tem grande importância devido ao seu papel como marco na conclusão da graduação em Engenharia Elétrica com ênfase em Computação. Um desses aspectos é, sem dúvida, a prática do trabalho em grupo, fundamental na carreira da Engenharia. Outro foi a aquisição de vivência e experiência no projeto e execução de um trabalho de grande porte, aplicando de forma relacionada uma ampla gama de conceitos e técnicas vistas durante a graduação. E, por fim, destaca-se o contato que a realização desse trabalho possibilitou com a inovação, com a pesquisa e com a tecnologia em estado da arte, que devem ser parte do cotidiano dos profissionais na área da Engenharia.

Referências

ABERER, K.; HAUSWIRTH, M.; SALEHI, A. A middleware for fast and flexible sensor network deployment. In: *Proceedings of the 32Nd International Conference on Very Large Data Bases*. VLDB Endowment, 2006. (VLDB '06), p. 1199–1202. Disponível em: <http://dl.acm.org/citation.cfm?id=1182635.1164243>.

AZZARA, A. et al. PyoT, a macroprogramming framework for the Internet of Things. In: IEEE. *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*. 2014. p. 96–103. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all-.jsp?arnumber=6871193>.

COSTANZO, S. et al. Software Defined Wireless Networks: Unbridling SDNs. In: *Software Defined Networking (EWSDN), 2012 European Workshop on*. IEEE, 2012. p. 1–6. ISBN 978-1-4673-4554-5. Disponível em: <http://dx.doi.org/10.1109/ewsdn.2012.12>.

CULLER, D.; ESTRIN, D.; SRIVASTAVA, M. Overview of sensor networks. *Computer Magazine*, IEEE Computer Society, v. 37, n. 8, p. 41–49, 2004.

Django Project. *The Web framework for perfectionists with deadlines | Django*. 2015. Disponível em: <https://www.djangoproject.com>. Acesso em: 22.06.2015.

FIELDING, R. T.; TAYLOR, R. N. Principled design of the modern web architecture. In: *Proceedings of the 22Nd International Conference on Software Engineering*. New York, NY, USA: ACM, 2000. (ICSE '00), p. 407–416. ISBN 1-58113-206-9. Disponível em: <http://doi.acm.org/10.1145/337180.337228>.

Flask. *Welcome | Flask (A Python Microframework)*. 2015. Disponível em: <http:/-/flask.pocoo.org/>. Acesso em: 22.06.2015.

GAY, D. et al. The nesC language: A holistic approach to networked embedded systems. In: ACM. *Acm Sigplan Notices*. 2003. v. 38, n. 5, p. 1–11. Disponível em: <http://dl.acm.org/citation.cfm?id=781133>.

Git. *Git*. 2015. Disponível em: <http://www.git-scm.com/>. Acesso em: 11.09.2015.

golang. *The Go programming language*. 2015. Disponível em: <https://golang.org>. Acesso em: 13.06.2015.

HEESCH, D. van. *Doxygen Manual*. 2015. Disponível em: <http://www.doxygen.org/>. Acesso em: 05.11.2015.

KOLDEHOFE, B. et al. The power of software-defined networking: Line-rate content-based routing using openflow. In: *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*. New York, NY, USA: ACM, 2012. (MW4NG '12), p. 3:1–3:6. ISBN 978-1-4503-1607-1. Disponível em: <http://doi.acm.org/10.1145-/2405178.2405181>.

LEVIS, P. et al. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In: ACM. *Proceedings of the 1st international conference on Embedded networked sensor systems*. 2003. p. 126–137. Disponível em: <http://dl.acm.org/citation.cfm?id=958506>.

LEVIS, P. et al. TinyOS: An operating system for sensor networks. In: *Ambient intelligence*. Springer, 2005. p. 115–148. Disponível em: <http://link.springer.com-/chapter/10.1007/3-540-27139-2_7>.

LUO, T.; TAN, H.-P.; QUEK, T. Q. S. Sensor OpenFlow: Enabling Software-Defined Wireless Sensor Networks. *IEEE Communications Letters*, v. 16, n. 11, p. 1896–1899, 2012. Disponível em: <http://dblp.uni-trier.de/db/journals/icl/icl16.htmlLuoTQ12>.

MADDEN, S. R. et al. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, ACM, v. 30, n. 1, p. 122–173, 2005.

MARGI, C. B. *Comunicação, segurança e gerenciamento em redes de sensores sem fio*. 2015. Tese (Livre-Docência) – Universidade de São Paulo.

OLIVEIRA, B. Trevizan de; MARGI, C. B.; GABRIEL, L. B. TinySDN: Enabling multiple controllers for software-defined wireless sensor networks. In: IEEE. *Communications (LATINCOM), 2014 IEEE Latin-America Conference on*. 2014. p. 1–6. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7041885>.

Oracle Corporation. *java.com: Java + You*. 2015. Disponível em: <https://www.java-.com/en>. Acesso em: 14.06.2015.

PIURI, V.; MINERVA, R. Building the internet of things. *Computer Now*, IEEE Computer Society, v. 8, n. 7, 2015.

Python Software Foundation. *Welcome to Python.org*. 2015. Disponível em: <https:/-/www.python.org>. Acesso em: 13.06.2015.

Sphinx. *Sphinx 1.3.1 Documentation*. 2015. Disponível em: <http://sphinx-doc.org/>. Acesso em: 05.11.2015.

SQLite. *SQLite*. 2015. Disponível em: <https://www.sqlite.org/>. Acesso em: 22.06.2015.

TELOSB. *TelosB - Crossbow Technology*. 2015. Disponível em: <http://www.willow.co-.uk/html/telosb_mote_platform.php>. Acesso em: 17.6.2015.

World Wide Web Consortium. *SOAP Specifications*. 2015. Disponível em: <http://www-.w3.org/TR/soap/>. Acesso em: 22.06.2015.