

**UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA**

**ARQUITETURA DE MIDDLEWARE
PARA INTERNET DAS COISAS**

HIRO GABRIEL CERQUEIRA FERREIRA

ORIENTADOR: RAFAEL TIMÓTEO DE SOUSA JÚNIOR

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA ELÉTRICA

PUBLICAÇÃO: PPGEE.DM - 576/2014

BRASÍLIA/DF, AGOSTO – 2014

**UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA**

ARQUITETURA DE MIDDLEWARE PARA INTERNET DAS COISAS

HIRO GABRIEL CERQUEIRA FERREIRA

**DISSERTAÇÃO SUBMETIDA AO DEPARTAMENTO DE
ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA
UNIVERSIDADE DE BRASÍLIA COMO PARTE DOS REQUISITOS
NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM
ENGENHARIA ELÉTRICA.**

APROVADA POR:

**Prof. Rafael Timóteo de Sousa Júnior, PhD (ENE-UnB)
(Orientador)**

**Prof. Ricardo Zelenovsky, PhD (ENE-UnB)
(Examinador Interno)**

**Prof. Mario Antônio Ribeiro Dantas, PhD (INE-UFSC)
(Examinador Externo)**

BRASÍLIA/DF, 28 DE AGOSTO DE 2014

FICHA CATALOGRÁFICA

FERREIRA, HIRO GABRIEL CERQUEIRA

Arquitetura de Middleware para Internet das Coisas [Distrito Federal] 2014.

xvii, 125p., 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2014). Dissertação de Mestrado – Universidade de Brasília. Faculdade de Tecnologia. Departamento de Engenharia Elétrica.

1. Sistemas Distribuídos

3. Internet das Coisas

I. ENE/FT/UnB II. Título (série)

2. Computação Ubíqua

4. Arquitetura de middleware

REFERÊNCIA BIBLIOGRÁFICA

FERREIRA, HIRO G. C. (2014). Arquitetura de Middleware para Internet das Coisas. Dissertação de Mestrado em Engenharia Elétrica, Publicação PPGEE.DM - 576/2014, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 125p.

CESSÃO DE DIREITOS

AUTOR: Hiro Gabriel Cerqueira Ferreira

TÍTULO: Arquitetura de Middleware para Internet das Coisas

GRAU: Mestre

ANO: 2014

É concedida à Universidade de Brasília a permissão para reproduzir cópias desta dissertação de mestrado e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. O autor reserva a si outros direitos de publicação, e nenhuma parte desta dissertação de mestrado pode ser reproduzida sem autorização por escrito dele.

Hiro Gabriel Cerqueira Ferreira
Condomínio Ouro Vermelho 2, quadra 10, lote 4, fase 1, Lago Sul
Brasília-DF – Brasil – 71.680-385
Celular: (61) 9111-8168 / e-mail: hiro.ferreira@gmail.com

AGRADECIMENTOS

Agradeço à minha mãe, Dione Mary, o suporte incondicional oferecido durante toda a minha vida, que me possibilitou chegar ao patamar onde estou e concluir este trabalho.

Agradeço à minha namorada, Ana Luisa, a companhia e os incentivos constantes durante todas as etapas de meu mestrado.

Agradeço ao meu orientador, Professor-Doutor Rafael Timóteo, todos os ensinamentos, conselhos e oportunidades oferecidos.

Agradeço à Professora-Doutora Edna Canedo a presteza e o auxílio em como escrever artigos científicos e como escolher congressos para publicá-los.

Agradeço aos meus colegas Armia Wagdy, Michael Abd-Elsayed e Nader Nohad, do departamento de Engenharia Elétrica da Universidade de Alexandria no Egito, o auxílio na codificação e testes do *Middleware*.

Agradeço ao meu pai, Nílson Ferreira, pelo auxílio na revisão de todo o texto deste trabalho e pelo incentivo em todas as demais áreas de minha vida.

Por último, e não menos importante, agradeço a meus irmãos e irmãs, avós, primos, tias, padrasto e amigos, pela presença e alegria que trazem para minha vida.

RESUMO

ARQUITETURA DE MIDDLEWARE PARA INTERNET DAS COISAS

Autor: Hiro Gabriel Cerqueira Ferreira

Orientador: Professor-Doutor Rafael Timóteo de Sousa Júnior

Programa de Pós-graduação em Engenharia Elétrica

Brasília, 28 de agosto de 2014

Com o advindo da terceira fase computacional, a era da computação ubíqua, a quantidade e o propósito específico de dispositivos dotados de poder computacional vêm crescendo aceleradamente. A incorporação desses dispositivos em redes de comunicação, como a rede de computadores mundial (*internet*), tem sido denominada como a Internet das Coisas (IoT). Tal paradigma procura prover inteligência para objetos, com ou sem poder computacional, de forma a possibilitar seu controle e a notificação de alterações em seu estado. Para isso, é demandado um modelo arquitetural de comunicação factível e capaz de abranger a maior gama possível de dispositivos, habilitar a comunicação dispositivos- -aplicações ou dispositivos-dispositivos, mesmo quando em geolocalizações distintas. Além disso, é necessário que tal arquitetura permita escalar para qualquer quantidade de objetos e prover interoperabilidade para dispositivos distintos e/ou de diferentes fabricantes.

Esta pesquisa visa propor uma arquitetura de middleware capaz de controlar e notificar o estado atual de dispositivos genéricos. Foram definidos os componentes físicos e lógicos dessa arquitetura e foram criadas as respectivas *application programming interfaces* (APIs) uniformes e transparentes, tanto para que aplicações pudessem controlar dispositivos englobados pela arquitetura, quanto para disponibilizar *interfaces* de comunicação analógicas e digitais, para que novos dispositivos controláveis pudessem ser adicionados nas APIs. As tecnologias utilizadas e propostas são escaláveis e extensíveis para dispositivos e aplicações e permitem a entrega de pervasividade para humanos. O desacoplamento físico entre os componentes do *middleware* e o uso da *internet* para troca de dados com aplicações trouxe mobilidade para as entidades envolvidas na comunicação.

Foi implementado um protótipo funcional do modelo arquitetural proposto, e foi possível remotamente controlar e monitorar o estado de dispositivos por meio do uso de tecnologias difundidas como o UPnP, REST e ZigBee sobre redes IP. O ambiente experimental mostrou velocidade satisfatória para atender demandas de controle, funcionou adequadamente, utilizando poucos recursos computacionais do *middleware* e das

aplicações envolvidas, e utilizou dispositivos que não eram dotados de poder computacional.

Com o modelo teórico definido e com sua validação empírica, foi constatado que o *middleware* proposto para IoT é capaz de controlar e monitorar estado de dispositivos trazendo a devida interoperabilidade, escalabilidade e mobilidade para as partes envolvidas.

ABSTRACT

PERVASIVE, MOBILE, TRANSPARENT, EXPANDABLE AND SCALABLE MIDDLEWARE ARCHITECTURE FOR INTERNET OF THINGS

Author: Hiro Gabriel Cerqueira Ferreira

Supervisor: Professor-Doctor Rafael Timóteo de Sousa Júnior

Electrical Engineering's Postgraduate Program

Brasilia, August 28th 2014

With the arising of the third computational era, ubiquitous computing, the amount and specific purpose of devices with computational power is growing rapidly. Such devices exchanging data through networks, like the Internet, have been named as the Internet of Things (IoT). Such paradigm aims to provide intelligence to objects, with or without computational power, in order to allow its control and notification of state changing. To be truly enabled, the IoT requests a feasible architectural model capable of covering the widest possible range of devices, capable of enabling communication between devices and to application, at the same place or not, capable of scaling to any number of objects and capable of providing interoperability for different kinds of devices and/or from different manufacturers.

This research proposed a middleware architecture capable of controlling and reporting the current status of generic devices. Transparent and uniform APIs were created so that applications could control devices encompassed by the architecture, and analog and digital communication interfaces were created so that new controllable devices could be added under the APIs. The proposed and used technologies are scalable and extensible for devices and applications and enables one to provide pervasiveness to humans. The physical decoupling between components of the middleware and the usage of the Internet to exchange data with applications brought mobility to involved entities.

A functional prototype of the proposed architectural model was deployed using widespread technologies like UPnP, ZigBee and REST over IP networks and allowed to remotely control and monitor the state of devices. The experimental environment showed satisfactory speed for control requests, worked flawlessly using few computational resources of middleware and applications involved, and was use with devices without computational power.

Through the architectural model and its empirical validation, it could be concluded that the proposed middleware for internet of things was able to control and monitor the state of devices bringing proper interoperability, scalability and mobility to involved parties.

SUMÁRIO

1.	INTRODUÇÃO	1
1.1	. MOTIVAÇÃO	3
1.2	. OBJETIVOS DO TRABALHO	3
1.3	. METODOLOGIA DE PESQUISA	4
1.4	. CONTRIBUIÇÕES DO TRABALHO	4
1.5	. ORGANIZAÇÃO DO TRABALHO	5
2.	ESTADO DA ARTE E REVISÃO BIBLIOGRÁFICA	6
2.1	. INTERNET DAS COISAS	6
2.1.1	COMPUTAÇÃO UBÍQUA.....	7
2.1.2	PERSPECTIVAS DA IoT.....	8
2.1.2.1	. VISÃO ORIENTADA A CONEXÃO	9
2.1.2.2	. VISÃO ORIENTADA A SEMÂNTICA	10
2.1.2.3	. VISÃO ORIENTADA A DISPOSITIVOS.....	11
2.2	. TECNOLOGIAS RELACIONADAS.....	11
2.2.1	. TECNOLOGIAS PARA PROTOTIPAGEM FÍSICA	12
2.2.1.1	. ARDUINO	12
2.2.1.2	. RASPBERRY Pi.....	15
2.2.1.3	. INTEL GALILEO E INTEL EDISON	16
2.2.2	. TECNOLOGIAS PARA COMUNICAÇÃO ENTRE DISPOSITIVOS.....	17
2.2.2.1	. TCP/IP.....	18
2.2.2.2	. IEEE 802.15.4 – ZigBee.....	19
2.2.3	. TECNOLOGIAS PARA ESCRITA SEMÂNTICA DE DADOS	22
2.2.3.1	. XML	22
2.2.3.2	. JSON	23
2.2.4	. TECNOLOGIAS PARA APIs E ABSTRAÇÃO LÓGICA DE OBJETOS	24
2.2.4.1	. APIs REST	25
2.2.4.2	. PILHA DE PROTOCOLOS UPnP	27
2.2.4.3	. 6LoWPAN e CoAP	30
2.2.5	– Tecnologias para Hospedagem e Acesso a Middlewares e seus Dados	31
2.2.5.1	. CLOUD COMPUTING	32
2.2.5.2	. NoSQL.....	34
2.2.6	. TECNOLOGIAS PARA IDENTIFICAÇÃO DE OBJETOS	36
2.3	. ARQUITETURAS EXISTENTES DE IoT	36
2.3.1	. Hydra – LynkSmart.....	37
2.3.2	. IoT-A.....	38
2.3.3	– iCORE.....	39
2.4	. PROBLEMAS EM ABERTO	40
2.5	. SÍNTESE DO CAPÍTULO	41
3.	PROPOSTA DE MIDDLEWARE PARA INTERNET DAS COISAS	42
3.1	. ARQUITETURA PROPOSTA	43
3.1.1	. ESTRUTURA FÍSICA	46
3.1.1.1	. CONTROLADOR-MESTRE.....	47
3.1.1.2	. CONTROLADOR-ESCRAVO	48
3.1.2	. ABSTRAÇÃO DE DISPOSITIVOS	50
3.1.2.1	. CONTROLADORES-ESCRAVO	51
3.1.2.2	. DISPOSITIVOS	51
3.1.2.3	. SERVIÇOS.....	52
3.1.2.4	. AÇÕES	52
3.1.2.5	. VARIÁVEIS DE ESTADO	52
3.1.2.6	. ARGUMENTOS DE ENTRADA E DE SAÍDA.....	53

3.1.3 . ESTRUTURA LÓGICA.....	53
3.1.3.1 . CAMADA DE SERVIÇOS	54
3.1.3.2 . CAMADA DE CONTROLE.....	58
3.1.3.3 . CAMADA DE COMUNICAÇÃO.....	61
3.1.3.4 . CAMADA DE EXECUÇÃO	66
3.1.4 . EXEMPLOS DE SISTEMAS	68
3.1.4.1 . MONITOR DE HUMIDADE	68
3.1.4.2 . CONTROLE DE ILUMINAÇÃO	69
3.1.4.3 . DETECÇÃO PARA INCÊNDIOS	70
3.1.4.4 . IRRIGAÇÃO	70
3.1.4.5 . CONTROLE DE TEMPERATURA	71
3.1.4.6 . ESTACIONAMENTO	72
3.1.4.7 . CONTROLE DE DISPOSITIVOS DOMICILIARES.....	73
3.1.5 . EXEMPLO DE CENÁRIO DE APLICAÇÃO.....	75
3.1.6 . SEGURANÇA.....	77
3.2 . SÍNTESE DO CAPÍTULO	80
4. PROTÓTIPO FUNCIONAL.....	81
4.1 . MODELO RELACIONAL DOS DADOS	81
4.2 . ADMINISTRAÇÃO DE ABSTRAÇÕES	83
4.3 . CONTROLADOR-MESTRE	86
4.4 . COMUNICADOR XBEE	88
4.5 . CONTROLADOR-ESCRAVO	89
4.6 . CENÁRIO DE TESTES E RESULTADOS OBTIDOS.....	90
4.7 . SÍNTESE DO CAPÍTULO	95
5. CONCLUSÕES.....	97
5.1 . TRABALHOS FUTUROS.....	98
5.2 . PUBLICAÇÕES RELACIONADOS A ESTE TRABALHO	98
REFERÊNCIAS BIBLIOGRÁFICAS.....	100
ANEXOS	104
ANEXO 1 – CÓDIGO ARDUINO PARA FAZER UM LED PISCAR	105
ANEXO 2 - XML UPnP DE DESCRIÇÃO DE DISPOSITIVOS	106
ANEXO 3 - XML UPnP DE DESCRIÇÃO DE SERVIÇOS	107
ANEXO 4 – JSONS TROCADOS ENTRE AS CAMADAS DE SERVIÇOS E DE CONTROLE.....	109
ANEXO 5 – JSONS TROCADOS ENTRE AS CAMADAS DE COMUNICAÇÃO E DE CONTROLE... ..	111

LISTA DE TABELAS

TABELA 1 - TIPOS JSON	24
TABELA 2 - RECURSOS E REQUISIÇÕES REST	26
TABELA 3 - PARÂMETROS DE UMA URL REST	26
TABELA 4 - CAMPOS DO JSON DE PEDIDO DE AÇÃO PARA CAMADA DE CONTROLE.....	55
TABELA 5 - CAMPOS DO JSON DE NOTIFICAÇÃO DE DISPONIBILIDADE, OU NÃO, DE UM DISPOSITIVO.....	55
TABELA 6 - CAMPOS DO JSON DE NOTIFICAÇÃO DE ALTERAÇÃO DE VARIÁVEL DE ESTADO	56
TABELA 7 - REQUISIÇÕES GET SUPORTADAS PELA API REST.....	57
TABELA 8 - REQUISIÇÕES PUT SUPORTADAS PELA API REST.....	58
TABELA 9 - CAMPOS DA INFORMAÇÃO CARREGADA POR PACOTES ZIGBEE DE PEDIDO DE ESCRITA DE VALOR EM PINO.....	62
TABELA 10 - CAMPOS DA INFORMAÇÃO CARREGADA POR PACOTES ZIGBEE DE MENSAGEM DE CONFIRMAÇÃO DE RECEBIMENTO	64
TABELA 11 - CAMPOS DA INFORMAÇÃO CARREGADA POR PACOTES ZIGBEE DE NOTIFICAÇÃO DE NOVO VALOR LIDO	65
TABELA 12 - ABSTRAÇÃO DO SISTEMA MONITOR DE HUMIDADE.....	69
TABELA 13 - ABSTRAÇÃO DO SISTEMA DE ILUMINAÇÃO	69
TABELA 14 - ABSTRAÇÃO DO SISTEMA PARA INCÊNDIOS.....	70
TABELA 15 - ABSTRAÇÃO LÓGICA DO SISTEMA DE IRRIGAÇÃO	71
TABELA 16 - ABSTRAÇÃO LÓGICA DO SISTEMA DE CONTROLE DE TEMPERATURA.....	71
TABELA 17 - ABSTRAÇÃO LÓGICA DO SISTEMA DE ESTACIONAMENTO.....	73
TABELA 18 - ABSTRAÇÃO DE UM CONTROLADOR DE DISPOSITIVOS DOMÉSTICO	73
TABELA 19 - TELAS DO SISTEMA DE GERÊNCIA DE ABSTRAÇÃO DE DISPOSITIVOS	83

LISTA DE FIGURAS

FIGURA 2.1 - RELAÇÃO ENTRE COMPUTAÇÃO UBÍQUA, PERVASIVA E MÓVEL (ARAÚJO, 2003)	7
FIGURA 2.2 - PARADIGMA DA INTERNET DAS COISAS COMO UM RESULTADO DE DIFERENTES VISÕES	9
FIGURA 2.3 - DUTY CYCLE PWM (ADAPTADO DE ARDUINO, 2014)	14
FIGURA 2.4 - ENCAIXE DE ESCUDOS ARDUINO	15
FIGURA 2.5 - RAPSBERRY PI MODELO B (RASPERRY PI, 2014)	16
FIGURA 2.6 - PLACAS INTEL GALILEO E EDISON (ADAPTADO DE INTEL CORPORATION, 2014)	17
FIGURA 2.7 - FUNCIONAMENTO DO TCP/IP E SUAS CAMADAS (KUROSE <i>ET AL.</i> , 2001)	19
FIGURA 2.8 - TOPOLOGIAS E NÓS ZIGBEE (ADAPTADO DE BARONTI <i>ET AL.</i> , 2007)	20
FIGURA 2.9 - TAGS	23
FIGURA 2.10 - BLOCOS DE UMA REDE UPNP (ADAPTADO DE UNIVERSAL PLUG AND PLAY FORUM, 2000)	28
FIGURA 2.11 - PILHA DE PROTOCOLOS UPNP (ADAPTADO DE UNIVERSAL PLUG AND PLAY FORUM, 2000)	30
FIGURA 2.12 - ARQUITETURA LINKSMART (LINKSMART WIKI, 2014)	38
FIGURA 2.13 - ÁRVORE DA IOT-A (IOT-A, 2014)	39
FIGURA 3.1 - PROPOSTA DE MIDDLEWARE	42
FIGURA 3.2 - VISÃO FÍSICA DA ARQUITETURA PROPOSTA	46
FIGURA 3.3 - FLUXO SIMPLIFICADO DE DADOS ENTRE ENTIDADES FÍSICAS	49
FIGURA 3.4 - RELACIONAMENTO DA ABSTRAÇÃO LÓGICA DE DISPOSITIVOS	50
FIGURA 3.5 - ARQUITETURA EM CAMADAS	54
FIGURA 3.6 - PROCEDIMENTO DA CAMADA DE CONTROLE PARA ALTERAR ESTADO DE UM DISPOSITIVO	59
FIGURA 3.7 - PROCEDIMENTO DA CAMADA DE CONTROLE PARA NOTIFICAÇÕES DE ALTERAÇÃO DE ESTADO	61
FIGURA 3.8 - EXEMPLO DE CIRCUITO DE ESCRITA	68
FIGURA 3.9 - DISPOSIÇÃO FÍSICA DE OBJETOS PARA O SISTEMA DE ESTACIONAMENTO	72
FIGURA 3.10 - SISTEMAS POR RECINTO DE UMA CASA	75
FIGURA 3.11 - DISTRIBUIÇÃO FÍSICA DE COMPONENTES PARA CENÁRIO DE APLICAÇÃO	76
FIGURA 3.12 - TLS NA COMUNICAÇÃO EXTERNA	77
FIGURA 3.13 - AES NA COMUNICAÇÃO INTERNA	78
FIGURA 3.14 - MÉTODOS DE AUTENTICAÇÃO OAUTH	79
FIGURA 4.1 - MODELO RELACIONAL DO PROTÓTIPO	82
FIGURA 4.2 - COMPONENTES DO CONTROLADOR-MESTRE	87
FIGURA 4.3 - TELAS DO APLICATIVO DE TESTES	88
FIGURA 4.4 - COMPONENTES DO CONTROLADOR-ESCRAVO	90
FIGURA 4.5 - COMPONENTES FÍSICOS DO PROTÓTIPO	90
FIGURA 4.6 - CONFIGURAÇÃO DE CONTROLADORES-ESCRAVO	91
FIGURA 4.7 - CONFIGURAÇÃO DE DISPOSITIVOS	91
FIGURA 4.8 - CONFIGURAÇÃO DE SERVIÇO	91
FIGURA 4.9 - CONFIGURAÇÃO DE AÇÕES	91
FIGURA 4.10 - CONFIGURAÇÃO DE VARIÁVEL DE ESTADO	91
FIGURA 4.11 - CONFIGURAÇÃO DE ARGUMENTO DE ENTRADA E SAÍDA	92

FIGURA 4.12 – TEMPO PARA ATENDER REQUISIÇÕES DE CONTROLE	93
FIGURA 4.13 - UTILIZAÇÃO DE RAM E CPU POR CÓDIGOS QT DO CONTROLADOR-MESTRE.....	93
FIGURA 4.14 - UTILIZAÇÃO DE RAM E CPU POR SERVIDOR MYSQL DO CONTROLADOR-MESTRE	94
FIGURA 4.15 - UTILIZAÇÃO DE RAM E CPU POR SERVIDOR APACHE DO CONTROLADOR-MESTRE	94

LISTA DE ACRÔNIMOS

6LoWPAN	<i>IPv6 over Low power Wireless Personal Area Networks</i>
AES	<i>Advanced Encryption Standard</i>
API	<i>Application Programming Interface</i>
CoAP	<i>Constrained Application Protocol</i>
CRUD	<i>Create, Read, Update and Delete</i>
DTMF	<i>Dual Tone Multi Frequency</i>
GENA	<i>Generic Event Notification Architecture</i>
HTML	<i>Hypertext Markup Language</i>
HTTP	<i>Hypertext Transfer Protocol</i>
IaaS	<i>Infrastructure as a Service</i>
IDE	<i>Integrated Development Environment</i>
IoT	<i>Internet of Things</i>
ISM	<i>Industrial Scientific and Medical Radio Bands</i>
IP	<i>Internet Protocol Version 4</i>
IPv6	<i>Internet Protocol Version 6</i>
IR	<i>Infrared</i>
JSON	<i>JavaScript Object Notation</i>
LED	<i>Light-Emitting Diode</i>
M2M	<i>Machine to Machine</i>
MAC	<i>Medium Access Control</i>
MAC*	<i>Message Authentication Code</i>
MIC	<i>Message Integrity Code</i>
NoSQL	<i>Not Only SQL</i>
PaaS	<i>Platform as a Service</i>
PWM	<i>Pulse Width Modulation</i>
PC	<i>Personal Computer</i>
REST	<i>Representational State Transfer</i>
RF	<i>Radio Frequency</i>
RFC	<i>Request for Comments</i>
RFID	<i>Radio Frequency identification</i>
SaaS	<i>Software as a Service</i>
SDK	<i>Software Development Kit</i>
SOAP	<i>Simple Object Access Protocol</i>
SQL	<i>Structured Query Language</i>
SSDP	<i>Simple Service Discovery Protocol</i>
TCP	<i>Transmission Control Protocol</i>
TLS	<i>Transport Layer Security</i>
UDN	<i>Unique Device Name</i>
UDP	<i>User Datagram Protocol</i>
UPnP	<i>Universal Plug and Play</i>
URL	<i>Uniform Resource Locator</i>
XML	<i>Extensible Markup Language</i>
W3C	<i>World Wide Web Consortium</i>
WSAN	<i>Wireless Sensors and Actuators Networks</i>

1. INTRODUÇÃO

Nas últimas décadas, ocorreu significativo aumento no poder computacional, na quantidade de dispositivos eletrônicos e nas tecnologias de comunicação de computadores disponíveis no mercado. Associado às reduções no custo de aquisição e no consumo de energia de dispositivos eletrônicos, foi habilitado um cenário nunca antes possível, o da computação ubíqua.

Inicialmente vislumbrado por Weiser (1991), o paradigma da ubiquidade é baseado em três pilares: Diversidade, Conectividade e Descentralização. A Diversidade vincula-se ao conceito de que cada dispositivo deve ter seu fim único, ou seja, deve atender à sua funcionalidade específica e prestar bem seu serviço único, sem a necessidade de realizar outras tarefas. A Conectividade traz a ideia de que cada dispositivo deve ser capaz de comunicar-se com os outros, independente de sua localização geográfica. A Descentralização trata de garantir que um dispositivo seja capaz de realizar suas funções independente de outros dispositivos, ou seja, ele deve conseguir existir e funcionar, sem depender de outras entidades para operar. A partir desses três pilares, a argumentação de Weiser consiste em que os dispositivos deveriam ser de tecnologia calma, pervasiva e móvel, ou seja, deveriam conseguir operar em segundo plano sem intervenção humana, de forma transparente para as pessoas e independente de onde estivessem.

Já a Lei de Goordon E. Moore (1965) sugere, de forma simplificada, que a capacidade de integração (por consequência, a capacidade computacional) de processadores dobra a cada dois anos. De acordo com a NPD Display Search (2014), a quantidade de dispositivos eletrônicos distintos quadruplicou nos últimos quatro anos. Isso traz, como consequência, o surgimento de dispositivos cada vez mais poderosos que, integrando capacidade de comunicação de baixo consumo provida por protocolos como o IEEE802.15.4, geram o cenário de dispositivos conectados aptos a prestar serviços para o homem em larga escala, como sugeriu Weiser.

Em 1999, Ashton propôs o termo *Internet das Coisas*, do inglês *Internet of Things* (IoT), para tratar do uso, na área de logística, de identificação de objetos por meio de radiofrequência, do inglês *Radio frequency Identification* (RFID). O significado do termo rapidamente ampliou-se e passou a abranger a área de sensores e atuadores sem fio, de objetos conectáveis a redes que utilizam o protocolo de interconexão *Internet Protocol*

(IP), bem como as tecnologias de semântica de dados, gerando, respectivamente, uma visão orientada às coisas, uma visão orientada à *internet* e uma visão orientada à semântica.

De acordo com Atzori *et al.* (2010), a junção dessas três visões gera o cenário real da IoT, que tem sinergia com a computação ubíqua, diferenciando apenas e principalmente no fato de os objetos da IoT serem orientados a conexão utilizando a *internet* como canal principal da troca final de dados.

Em 2013, existiam mais de dez bilhões de dispositivos capazes de comunicar-se utilizando a *internet* e, de acordo com o *T Sensor Summit Forum*, a projeção é que esse valor chegue a um trilhão em 2022. Essa grande quantidade de equipamentos traz desafios na área de infraestrutura pois será preciso administrar o tráfego e o armazenamento dos dados gerados por esse grande volume de dispositivos.

A interoperabilidade entre dispositivos de diferentes fabricantes, o reaproveitamento de dispositivos já existentes (estes sem capacidade de comunicação), a escalabilidade, a adaptabilidade, a confiabilidade, a velocidade e a segurança de estruturas para comportar esses dispositivos também apresentam desafios que demandam esforços acadêmicos e da indústria para habilitar o real potencial da IoT e da ubiquidade.

Considerados tais desafios, as Arquiteturas de *middleware* para habilitar a IoT são, portanto, estruturas físicas e lógicas complexas. Tais plataformas devem ser modularizadas e utilizar o máximo de tecnologias já existentes para assim permitir seu fácil aperfeiçoamento e reposição por métodos mais atuais e/ou adequados.

Objetivando apresentar uma resposta a tais desafios e requisitos, arquitetura proposta nesta dissertação é baseada em tecnologias difundidas nas áreas de:

- Semântica, como a *Extensible Markup Language* (XML) e a *JavaScript Object Notation* (JSON);
- Tecnologias de abstração de objetos por meio de APIs, como o *Universal Plug and Play* (UPnP) e o *Representational State Transfer* (REST);
- Plataformas de prototipagem de *hardware*, como Arduino e Raspberry Pi;
- Tecnologias consolidadas nas áreas de comunicação, como o TCP/IP e o IEEE 802.15.4;

- Tecnologias consolidadas e emergentes na área de escalabilidade de hospedagem e de aplicações de larga escala, como *Cloud Computing* e *Not Only SQL* (NoSQL).

Com isso, o intuito do presente trabalho é prover uma maneira flexível, descentralizada, escalável e expansível de permitir o controle e a notificação sobre estado da máxima quantidade e gama possível de dispositivos. É almejado que eles sejam transparentemente utilizados pelo máximo de tipos distintos de aplicações para proverem seus serviços pervasivos específicos sem consumir recursos desnecessários.

1.1. MOTIVAÇÃO

Apesar de já existirem *middlewares* que permitem o controle remoto de dispositivos, a maioria não possui APIs padronizadas, nem formas intuitivas e simplificadas de adicionar mais dispositivos. Também, não são escaláveis ou ainda estão em fase de levantamento de requisitos. Por tais motivos, há indicações de que muitos dos problemas que virão com a IoT ainda não foram sequer descobertos.

Além de ser importante prover uma revisão detalhada e atual sobre tecnologias e assuntos relacionados à ubiquidade e à IoT, a elaboração de uma pesquisa detalhada e a criação de um modelo real para *middleware* de IoT permitem identificar tópicos de discussão e definem parâmetros para comparações futuras. Modelos diferentes ou melhorias na proposição deste trabalho poderão assim ser feitos.

1.2. OBJETIVOS DO TRABALHO

Mais precisamente, o presente trabalho visa simplificar e padronizar o controle e a notificação de estado de dispositivos na IoT. Nesse contexto, esta pesquisa almeja:

- a. Criar um *middleware* para transformar objetos simples (do inglês *Plain Objects*) em objetos inteligentes (do inglês *Smart Objects*). O objetivo é habilitar objetos a informar seu estado atual e a permitir serem controlados por outros dispositivos ou aplicações;
- b. Criar uma plataforma capaz de comunicar-se com o máximo de *Smart Objects* já existentes e distintos, por meio do envio e do recebimento de comandos, utilizando tecnologias como *Powerline*, DTMF, CoAP ou UPnP;
- c. Criar APIs para que programadores possam utilizar dispositivos por meio de *interfaces* com semântica padronizada, mesmo para objetos de fabricantes e tecnologias distintas, ou *Plain Objects* convertidos em *Smart Objects* pela própria arquitetura;

- d. Na concepção do *middleware* proposto, integrar tecnologias já existentes para que possa ser posteriormente reaproveitado e melhorado por interessados;
- e. Como resultado adicional do desenvolvimento da plataforma, configurar um primeiro passo no estudo real de consequências geradas pelo uso de *Smart Objects* em áreas, como tráfego e segurança de dados.

Em síntese, o principal objetivo desta pesquisa é a elaboração de uma arquitetura de *middleware* para IoT que utilize tecnologias já existentes, que seja flexível tanto para dispositivos quanto para aplicações e que possa ser escalável para englobar a maior quantidade possível de elementos em sua estrutura, em especial no que se refere a dispositivos usados nas implementações.

1.3. METODOLOGIA DE PESQUISA

Nesta dissertação, a proposta de pesquisa foi dividida em três etapas para facilitar o entendimento do trabalho. Essa metodologia visa aprofundar o estudo relacionado ao tema e aos problemas relacionados, identificando os assuntos abordados pela comunidade acadêmica e os desafios na construção de uma arquitetura para IoT:

- Fase 1:** Realizar pesquisa bibliográfica para identificar e analisar artigos e principais problemas relevantes sobre o tema;
- Fase 2:** Obter informações sobre tecnologias relacionáveis ao tema e como elas podem auxiliar na resolução de problemas encontrados na Fase 1;
- Fase 3:** Projetar e implementar uma arquitetura capaz de solucionar o máximo de problemas encontrados na Fase 1, utilizando as tecnologias encontradas na Fase 2 para obter resultados reais, efetuar conclusões e identificar contribuições.

1.4. CONTRIBUIÇÕES DO TRABALHO

O presente trabalho busca trazer as seguintes contribuições:

- Apresentação do estado da arte de tecnologias para IoT, principalmente daquelas orientadas à produção de *middlewares*;
- Apresentação de um modelo genérico de arquitetura de *middleware* para IoT capaz de comportar uma grande variedade de dispositivos;
- Implementação funcional da proposta de arquitetura de *middleware* para IoT com posterior disponibilização dos códigos e dos modelos de *hardware* e de *software*.

1.5. ORGANIZAÇÃO DO TRABALHO

Este trabalho foi ordenado em cinco capítulos, sendo este primeiro o de Introdução. Para facilitar o entendimento desta pesquisa, os demais capítulos estão organizados como descrito a seguir:

O Capítulo 2 oferece a revisão atual das principais tecnologias englobadas pela IoT ou que podem ser utilizadas diretamente por ela. Além disso, são apresentados os trabalhos correlatos, outras arquiteturas de *middlewares* disponíveis e os principais problemas em aberto na área de infraestrutura de *middlewares*.

O Capítulo 3 apresenta os principais desafios encontrados para a realização de arquiteturas de *middleware* para IoT e, com efeito, a proposta final gerada por esta pesquisa e casos de uso para esse modelo final.

O Capítulo 4 apresenta um protótipo real, composto por *softwares* e *hardwares*, e os resultados obtidos.

O Capítulo 5 conclui este trabalho. Nele, sintetizam-se os resultados encontrados e são sinalizados caminhos futuros que podem ser seguidos para a sequência deste trabalho.

2. ESTADO DA ARTE E REVISÃO BIBLIOGRÁFICA

Este capítulo contém a revisão dos principais conceitos de Ubiquidade, IoT e de tecnologias relacionadas aos dois temas. Para facilitar o entendimento, o presente capítulo está organizado em três tópicos maiores. Na seção 2.1, é tratada a Teoria de Ubiquidade e de IoT. Na seção 2.2, são revisadas as principais tecnologias atuais que permitem a criação de ambientes da IoT. Na Seção 2.3, são mostradas as mais conhecidas arquiteturas de IoT disponíveis, mesmo aquelas que ainda estão em fase de levantamento de requisitos.

2.1. INTERNET DAS COISAS

O termo *Internet das Coisas*, nomeado por Ashton (1999), tornou-se popular principalmente por seu sucesso no uso da tecnologia RFID em áreas de rastreamento de objetos, pessoas e animais (Kortuem *et al.*, 2010).

Trazendo a ideia de que cada objeto deveria ser unicamente endereçável, o conceito de Internet das Coisas foi ampliado, adicionando a comunicação de objetos por meio da *internet*. Com isso, é esperado que a *internet* passe a comunicar não somente pessoas com pessoas mas também objetos com pessoas e objetos com objetos, do inglês *Machine to Machine* (M2M).

Interações entre objetos permitirá que prestem serviços complexos para as pessoas sem necessariamente requerer intervenção humana (Tan *et al.*, 2010). Dessa forma, os objetos inteligente sairão do foco do cotidiano do homem, ficando em execução de segundo plano, e, por isso, tornar-se-ão pervasivos.

Outra característica importante trazida pelo uso da *internet* na comunicação de objetos é a mobilidade. A *internet*, também conhecida como a *rede mundial de computadores* ou a *rede de redes*, possui alcance global e permite que dispositivos se conectem a ela no mundo inteiro. Tal benefício também é trazido para os objetos pois poderão trocar sua geolocalização sem necessariamente perder sua conectividade.

A junção de características de pervasividade e de mobilidade retomam ao conceito de ubiquidade trazido por Mark Weiser, no começo da década de 90.

A seguir são explicados os conceitos da computação ubíqua, era computacional que habilita o surgimento de cenários reais em que objetos são capazes de comunicarem-se para prestar serviços para o homem, como almeja a IoT. Em seguida, será retomado o tema da IoT, abordando suas principais visões e características.

2.1.1 COMPUTAÇÃO UBÍQUA

O paradigma da computação ubíqua, vislumbrado inicialmente por Weiser (1991), é caracterizada por alto grau de mobilidade, transparência de uso para usuários e reação ao contexto. Como sugerido por Lyytinen (2002), ela pode ser caracterizada pela junção da computação móvel (alto grau de mobilidade) com a computação pervasiva (alto grau de transparência e reação ao contexto), como mostrado na Figura 2.1. Com essa união, a ubiquidade atinge seus três princípios: Diversidade, Conectividade e Descentralização.

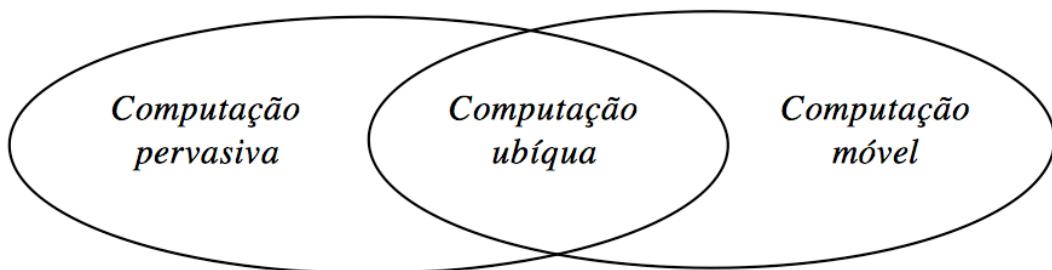


Figura 2.1 - Relação entre Computação Ubíqua, Pervasiva e Móvel (Araújo, 2003)

A computação pervasiva é aquela que traz, para um sistema computacional, a facilidade e a transparência no uso, e a capacidade de entendimento do ambiente e do contexto em que está inserido (Merk, 2001). Ela objetiva prover o melhor serviço possível para o homem de forma calma e pertinente.

A computação móvel é aquela que traz mobilidade e portabilidade a sistemas computacionais (Chen *et al.*, 2000). Ela objetiva permitir que um sistema possa mudar sua geolocalização ou as máquinas físicas em que estão hospedadas, sem perder suas características de funcionamento e de conexão.

O Princípio da Diversidade é aquele que demanda que um dispositivo ubíquo tenha seu propósito específico de existência (Hansmann, 2003). Isso objetiva o uso de suas

funcionalidades para o fim mais adequado para ela. Um dispositivo não necessita de ser apto a fazer nada além de sua funcionalidade específica e, preferencialmente, deve ser utilizado para ela sempre que estiver disponível.

O Princípio da Conectividade é aquele que demanda que um dispositivo mantenha sua conexão com outros para estar apto a prover serviços (Hansmann, 2003). É importante frisar que, caso seja o mais adequado entre os disponíveis na rede, ele deverá executar uma tarefa mesmo que não seja a de seu propósito específico.

O Princípio da Descentralização é aquele que demanda que um dispositivo consiga realizar seu propósito específico mesmo quando não estiver com outros para auxiliá-lo (Hansmann, 2003). Ele deve ser capaz de, sozinho, executar sua função específica que deve independe de sua localização e de quem estiver ao seu redor.

Da união dos três princípios e das computações móveis e pervasivas, nasce a computação ubíqua. Esse paradigma herda e prega a mobilidade, a portabilidade, a transparência de comunicação, a trivialidade de uso, a pertinência, a reação ao contexto, o propósito específico, a conexão constante e a independência de existência.

2.1.2 PERSPECTIVAS DA IoT

Os conceitos da computação ubíqua são utilizados na IoT em suas três principais óticas. É com esses conceitos que novas tecnologias surgem para dar vida à IoT.

Para o entendimento do que a IoT abrange, primeiro é preciso analisar quais são suas principais visões e o que elas englobam. Um resumo dessas visões com alguns itens que fazem parte delas estão exibidos na Figura 2.2 e estão descritos a seguir.

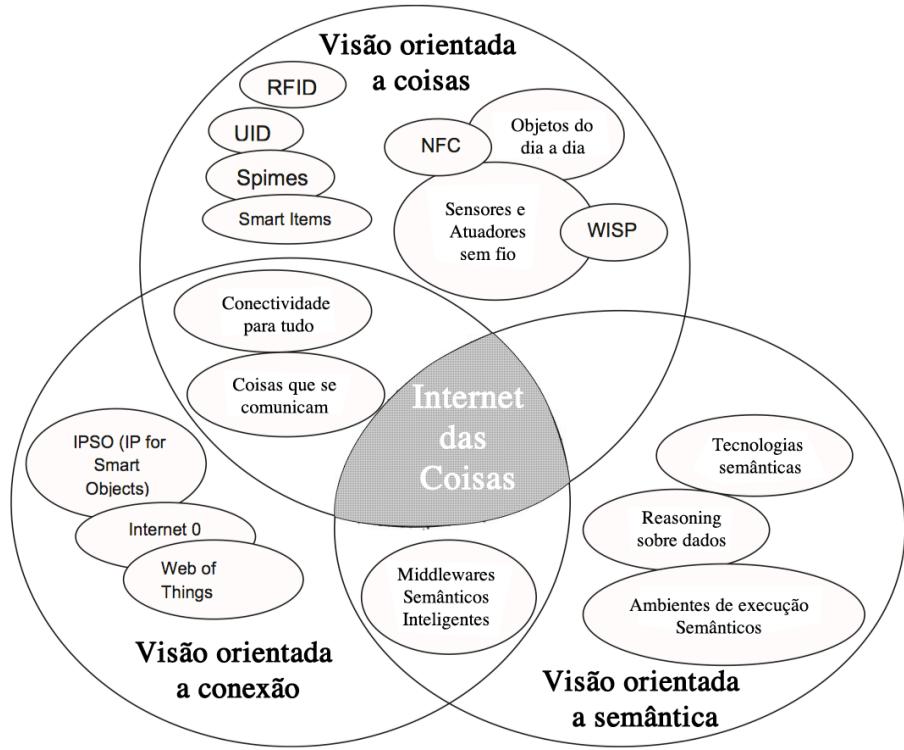


Figura 2.2 - Paradigma da Internet das Coisas como um resultado de diferentes visões
 (adaptado de Atzori *et al.*, 2010)

2.1.2.1 . VISÃO ORIENTADA A CONEXÃO

A comunicação entre computadores é estudada desde a década de 50, como mostrado por Kleinrock (2010). Tais estudos permitiram o surgimento de tecnologias como o TCP/IP, que padronizaram a maioria das comunicações entre computadores utilizadas na *internet*. A necessidade de dispositivos mais baratos, cada vez menores e, consequentemente, menos poderosos que os computadores tradicionais, exigiu atenção dobrada ao tamanho de componentes físicos e ao consumo de energia do sistema.

A necessidade de operação em ambientes e em condições mais restritas impactou diretamente na capacidade e na forma de comunicação desses dispositivos que, em muitos casos, passaram a utilizar comunicações sem fio baseadas em protocolos e outras tecnologias proprietárias. Logo veio a necessidade de padronizar a comunicação entre dispositivos para ambientes restritos e, com isso, surgiram padrões de mercado e de consórcios como a Internet 0, o IPSO e a *Rede das Coisa*, do inglês *Web of Things* (WoT).

A Internet 0 é uma proposta do centro para *bits* e átomos do MIT que pregam o IP para qualquer objeto. Essa tecnologia é uma modulação ponto a ponto para redes de dispositivos embarcados, baseada em IP para prover comunicação simplificada, barata e lenta entre dispositivos (Gershenfeld *et al.*, 2006). A Internet 0 se assemelha ao Código Morse e é utilizada por meio de RF, infravermelho, ultrassom, *powerLine* e códigos de barra.

A aliança do protocolo da Internet para Objetos inteligentes, do inglês *Internet Protocol for Smart Objects Aliance* (IPSO Aliance), é um esforço conjunto de organizações, com e sem fins lucrativos, para promover e padronizar a comunicação e acesso a dispositivos inteligentes, utilizando o IP como tecnologia base (IPSO 2014). Instituições como o MIT, Nokia, Google, Cisco, Bosch, Toshiba, Oracle e Atmel são membros ativos do IPSO.

A WoT é o conceito que traz e prega a tendência de objetos a conectarem-se à *web* por meio do IP.

Tecnologias de comunicação mais difundidas – como o *ZigBee* (IEEE802.15.4), *bluetooth* (IEEE802.15.1), *RFID* e *6LoWPAN* – são influenciadas, auxiliadas, evoluídas e incorporadas na IoT por alianças, tecnologias e conceitos como, respectivamente, o IPSO, a Internet 0 e a WoT. A maioria das soluções de comunicação de dispositivos tende a usar a *internet* (IP) e esse protocolo tem-se tornado o principal foco de estudo, na visão de conectividade da IoT para aproveitar sua infraestrutura, maturidade e difusão através da *world wide web*.

2.1.2.2 . VISÃO ORIENTADA A SEMÂNTICA

A IoT também se preocupa em como abstrair, padronizar e exibir dados relativos a dispositivos. Essa consideração vem para permitir a interoperabilidade e a usabilidade de dispositivos que, de alguma forma, sejam capazes de comunicarem-se.

Tecnologias, como o UPnP, e práticas, como *reasoning* e buscas ontológicas, utilizam-se tecnologias de representação como o XML e o JSON para padronizar a interoperabilidade e permitir a reutilização e a inteligência do uso de dados (Atzori *et al.*, 2010). Um conceito que exemplifica a visão semântica da IoT é o da Web Semântica (Berners-Lee *et al.*, 2001), que prega a evolução da *web* como sendo aquela em que os dados possuem significados

bem definidos por meio de interoperabilidade estrutural, sintática e semântica (Oliveira, 2011).

2.1.2.3 . VISÃO ORIENTADA A DISPOSITIVOS

Além de englobar a comunicação com o mundo externo e em como os dados devem ser gerenciados, escritos e interpretados, a IoT também abrange a amplificação da inteligência dos objetos para que eles passem de um objeto simples para um objeto inteligente.

A visão orientada a dispositivo foca em como traduzir dados entre o mundo real e o virtual, em como identificar um objeto unicamente, em como aplicar ações e ler estados de dispositivos físicos. É nessa área que é adicionada a sensibilidade ao contexto em que o dispositivo está inserido.

Tecnologias, como RFID, NFC e Códigos de Barra, auxiliam na identificação de dispositivos. Tecnologias de sensores e atuadores, *spimes* e WISP auxiliam na indução e na leitura de estados de equipamentos, e microcontroladores, como Arduino, Raspberry pi e Galileo, orquestram as demais tecnologias de dispositivos para criar a inteligência do contexto e para interagir como canal de comunicação por meio da semântica devida.

2.2. TECNOLOGIAS RELACIONADAS

Na seção 2.1, foram apresentados os principais conceitos da IoT e como eles se relacionam. Esta seção é focada nas principais tecnologias que permitem que a IoT seja aplicável na vida real. Esta seção aborda as tecnologias práticas para a realização da IoT.

Na subseção 2.2.1, passaremos pelas tecnologias que, quando acopladas a um dispositivo, permitem que ele passe a ser um objeto inteligente. A subseção 2.2.2 trata das principais tecnologias para comunicação com outros dispositivos e sistemas que objetos inteligentes utilizam e para os quais prestam serviços. A seção 2.2.3 trata das abstrações linguísticas em que dados de um dispositivo devem ser escritos para que possam ser entendidos por outras entidades. A seção 2.2.4 trata das tecnologias para abstrair um dispositivo, convertendo suas características físicas em características virtuais que podem ser absorvidas, interpretadas e transmitidas em conjunto com as tecnologias abordadas nas subseções 2.2.1, 2.2.2 e 2.2.3. A subseção 2.2.5 trata de ambientes e modelos de dados

contemporâneos e avançados para a hospedagem e a execução da parte virtual de dispositivos. A seção 2.2.6 finaliza, tratando das principais tecnologias para a identificação única de dispositivos.

2.2.1. TECNOLOGIAS PARA PROTOTIPAGEM FÍSICA

A prototipagem física aqui abordada consiste na construção de *hardwares* que permitam a interação entre máquina e objeto simples. A proveniência de tal interação permite que dados analógicos ou digitais sejam transformados em estímulos interpretáveis por um objeto. Um exemplo simples dessa interação seria um comando digital de ligar ou desligar uma lâmpada que seria convertido em ação equivalente a ligar e a desligar um interruptor por meio de, por exemplo, um relé conectado a um microcontrolador. O protótipo físico recebe o comando digital ou analógico, interpreta-o e repassa-o a um componente especializado capaz de executar a ação correta.

Aqui, abordaremos microcontroladores que são dotados de processadores e de memórias mais limitadas e que são capazes de receber uma série de comandos e os converter em estímulos analógicos ou digitais para excitar um objeto, fazendo com que ele altere seu estado.

As principais plataformas de *hardware* e de *software* disponíveis atualmente são o microcontrolador Arduino (seção 2.2.1.1), a Raspberry Pi (seção 2.2.1.2) e os controladores Galileo (seção 2.2.1.3) e Edison (seção 2.2.1.3).

2.2.1.1 . ARDUINO

Lançada em 2005, a Arduino é uma plataforma aberta de *hardware* e de *software*, construída em placa única que possui o intuito de simplificar a aplicação de objetos ou de ambientes interativos. Esta plataforma de baixo custo possui entradas e saídas digitais e analógicas e possui muitos componentes especializados em interagir com objetos simples e inteligentes (Arduino, 2014).

Em geral, os modelos da placa utilizam a família de processadores megaAVR, da corporação Atmel (2014), e possuem voltagem de operação de 5V, corrente de 40mA por pino (alcance de 150m em fio de 20AWG, sem amplificação), 32KB de memória *flash*, SRAM de 2KB, EEPROM de 1KB e relógio de 16MHz. As placas são programadas,

utilizando as bibliotecas Arduino, com a linguagem C sob uma abstração simplificada de escrita. A programação da placa foi abstraída, para que pessoas sem familiaridade com microcontroladores ou com a linguagens de programação, consigam escrever aplicações funcionais e que possam ser facilmente transmitidas para placa utilizando o ambiente de desenvolvimento integrado, do inglês *Integrated Development Environment* (IDE). O IDE Arduino, que inclui a biblioteca Arduino automaticamente em códigos escritos nele, se responsabiliza por fazer a transmissão e a implantação do programa na placa automaticamente.

O Anexo 1 contém um código de exemplo que mostra como ligar e desligar um diodo emissor de luz, do inglês *Light-emitting diode* (LED). Caso seja conectado um relé a um interruptor de lâmpada e ao pino do LED, tal código faria uma lâmpada ligar e desligar.

A placa possui entradas e saídas digitais e analógicas. A saída, ou escrita, digital em seus pinos consiste em converter códigos do programa HIGH (1) ou LOW (0), em sinais constantes de 5V ou 0V, em seus pinos, respectivamente. A leitura, ou entrada, digital em seus pinos consiste em interpretar se a voltagem em seus pinos é de 5V ou 0V e converter a leitura para código nos valores HIGH (1) ou LOW (0), respectivamente. A leitura, ou entrada, analógica consiste em ler a voltagem em seus pinos que pode variar entre 0V e 5V, e quantizar o valor, por meio de um conversor analógico digital (ADC), em escala de 0 a 1023, a ser repassado para o código como um inteiro, e que, pela quantização utilizada, possui sensibilidade/erro de 4,9mV. A escrita, ou saída, analógica consiste em pegar valor de código, inteiro variante de 0 a 255, e repassar para seus pinos, utilizando a técnica de modulação de largura de pulsos, do inglês *Pulse Width Modulation* (PWM). Tal técnica simula resultados analógicos em dispositivos analógicos, utilizando meios digitais com a alteração do tempo em que o sinal permanece em 5V e 0V. O tempo que o pulso possui para variar seu valor entre 0V e 5V é de 2 milissegundos e esse intervalo de repetição é chamado de *duty cycle*.

Se o programador envia 0 de saída analógica em seu código, um pulso gerado no pino permanece 0% de um *duty cycle* em 5V, um dispositivo analógico seria estimulado da mesma maneira que se recebesse 0V durante todo o tempo. Se o programador envia 64 de saída analógica em seu código, um pulso gerado no pino permanece 25% de um *duty cycle* em 5V e o restante em 0V, um dispositivo analógico seria estimulado da mesma maneira

que se recebesse 1,25V, durante todo o tempo. Se o programador envia 127 de saída analógica em seu código, um pulso gerado no pino permanece 50% de um *duty cycle* em 5V e o restante em 0V, um dispositivo analógico seria estimulado da mesma maneira que se recebesse 2,5V, durante todo o tempo. Se o programador envia 191 de saída analógica em seu código, um pulso gerado no pino permanece 75% de um *duty cycle* em 5V e o restante em 0V, um dispositivo analógico seria estimulado da mesma maneira que se recebesse 3,75V, durante todo o tempo. Se o programador envia 255 de saída analógica em seu código, um pulso gerado no pino permanece 100% de um *duty cycle* em 5V, um dispositivo analógico seria estimulado da mesma maneira que se recebesse 5V, durante todo o tempo. Essas relações estão apresentadas na Figura 2.3 - Duty Cycle PWM.

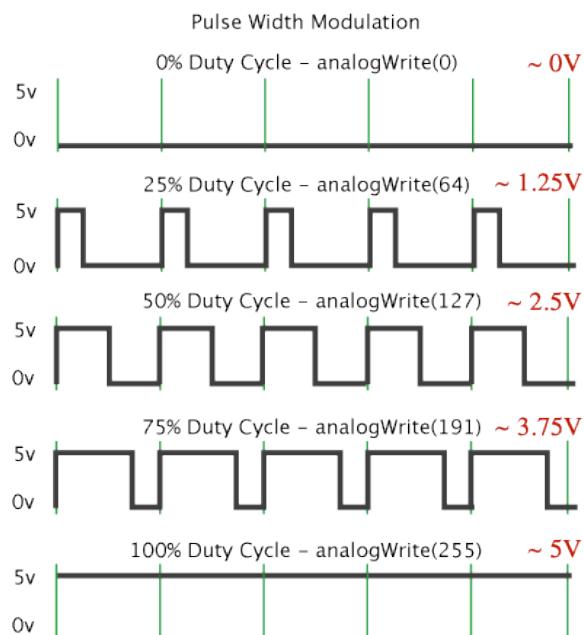


Figura 2.3 - Duty Cycle PWM (adaptado de Arduino, 2014)

Outra característica da Arduino é a de possuir bibliotecas de *software* para comunicação cabeadas e sem fio com outros dispositivos, como computadores e outros microcontroladores. Combinado ao *software*, a placa é prototipada para conectar-se, com o encaixe superior, com placas de expansão, denominadas *escudos*, que são capazes de transmitir os dados por meio de radiofrequência ou comunicação cabeadas, como mostrado na Figura 2.4 - Encaixe de escudos Arduino. Os escudos costumam manter disponíveis os pinos analógicos e digitais de entrada e de saída para que outros escudos possam ser colocados em cima deles. Além de comunicação WiFi, existem escudos para outras aplicações como Ethernet, LCD, GSM, GPS, USB e XBEE, que também são de encaixe.



Figura 2.4 - Encaixe de escudos Arduino

Além dos escudos existem vários componentes de baixíssimo custo desenvolvidos por vários fabricantes que possuem *hardware* e *softwares* feitos especificamente para compatibilidade com as entradas e as saídas digitais e analógicas da Arduino. Entre os componentes mais comuns, estão medidores de temperatura do ar, umidade do ar, umidade do solo, pressão, distância, luminosidade, fumaça, toque humano, leitores RFID, detector de incêndio, giroscópio, infravermelho, acelerômetro, detector de álcool no ar, detector de som, GPS, detector de proximidade de metais, detector de locomoção, identificação de cor e luz visível. Na maioria dos casos, basta conectar o componente físico à placa Arduino, importar a biblioteca do componente no código Arduino e utilizar a abstração dos componentes tão naturalmente como se usa a biblioteca Arduino padrão. A comunidade de usuários da placa é extremamente ativa, e a documentação e a ajuda *online* são claras e fáceis de serem encontradas, o que garantiu – e garante – a continuidade e a evolução dessa plataforma de *hardware* e *software* livre e aberta.

2.2.1.2 . RASPBERRY Pi

Lançada em 2006, a Raspberry Pi é um microcontrolador do tamanho de um cartão de crédito que tem como intuito auxiliar no ensino de programação e em projetos de eletrônica como automação residencial, experimentos científicos ou músicos. Ela é baseada em sistemas operacionais livres, como o Debian ou Fedora, que ficam hospedados e rodam a partir de um cartão SD. A placa é alimentada por uma porta USB comum e, ao ser conectada a um teclado, mouse e monitor, pode funcionar como um computador Linux regular. A Raspberry Pi pode ter suas aplicações desenvolvidas em linguagens como Python, C, C++, Java e Ruby, e permite a instalação de *softwares* adicionais que possam ser compilados para plataforma ARMv6 (RaspberryPi, 2014).

O modelo B, considerado o modelo padrão da placa, vem com 512MB de memória RAM, duas portas USB, uma porta Ethernet de 100Mb, saída de vídeo RCA e HDMI de vídeo, saída de áudio P2 e HDMI, suporta cartões SD de até 32GB, GPU e CPU com relógio de 700MHz da Broadcom (2014), 26 pinos de entrada e saída digital (GPIO), com até 16mA de corrente por pino, e trabalhando entre 0V e 3.3V ou entre 0Ve 5V. A placa realiza escrita PWM mas não dá suporte à leitura analógica. O esquemático da placa é exibido na Figura 2.5 - Raspberry Pi Modelo B.

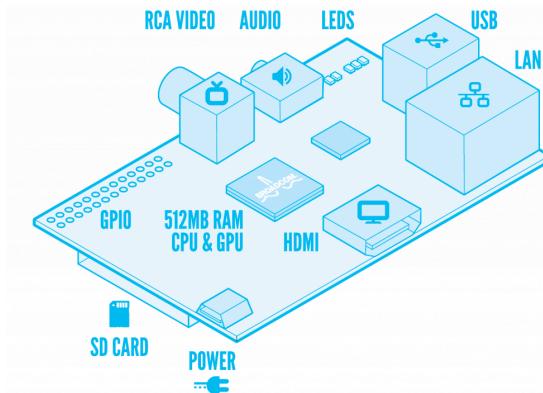


Figura 2.5 - Raspberry Pi Modelo B (Raspberry Pi, 2014)

Assim como a Arduino, a Raspberry Pi também pode conectar os GPIOs a uma variedade de circuitos desenvolvidos por outros fabricantes que, em geral, permitem controlar e sentir o ambiente. Ambas as placas são pequenas e de baixo custo, sendo a Arduino a menor e mais barata. O principal diferencial da Raspberry Pi está em sua capacidade de processamento, armazenamento e, consequentemente, gama de aplicações que são muito superiores à da Arduino.

2.2.1.3 . INTEL GALILEO E INTEL EDISON

Em 2013, a Corporação Intel lançou um microcontrolador chamado Galileo. Tal placa foi feita em proximidade com a Arduino e possui compatibilidade com os escudos e códigos Arduino. Baseada na arquitetura x86, a Galileo roda um sistema Linux em cartão SD que possui as bibliotecas da Arduino e pode ser programada usando o IDE Arduino. O microcontrolador da Intel possui processador de 32bits com um núcleo de *thread* única e relógio de 400MHz. A Galileo possui 256MB de RAM, portas USB, Ethernet, microSD e RS-232, relógio em tempo real e faz escrita e leitura digitais e analógicas (PWM, ADC) com pinagem igual à da Arduino (Intel Corporation, 2014).

Em 2014, a Corporação Intel introduziu o microcontrolador Edison. Tal placa possui o tamanho de um cartão SD e vem com memória, *bluetooth* e *WiFi* integrados, possui processador de dois núcleos x86, com relógio de 400MHz, Linux embarcado e suporta códigos Arduino, Python, Node.js e Wolfram (Intel Corporation, 2014).

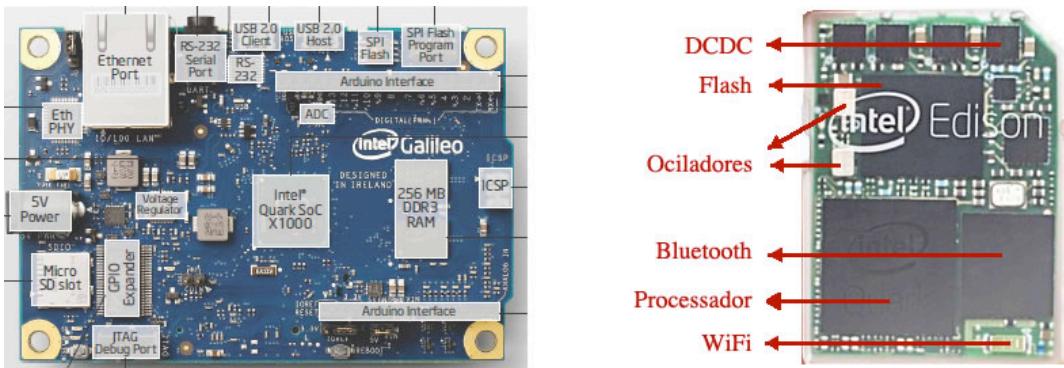


Figura 2.6 - Placas Intel Galileo e Edison (adaptado de Intel Corporation, 2014)

Edison e Galileo fazem parte de um esforço da Intel de prover uma solução para a IoT e para tecnologias vestíveis, do inglês *wearable technology*. Tais microcontroladores possuem como diferencial sua alta compatibilidade com diversas linguagens de programação, com tecnologias Arduino, elevado poder de processamento e tudo a custo acessível. O esquemático das placas é mostrado na Figura 2.6 - Placas Intel Galileo e Edison (adaptado de Intel Corporation, 2014)

2.2.2 . TECNOLOGIAS PARA COMUNICAÇÃO ENTRE DISPOSITIVOS

Um microcontrolador, conectado a um objeto, podendo assim controlar e monitorar o estado desse objeto simples, é considerado um objeto inteligente. Para habilitar a IoT é preciso que o objeto inteligente possa prestar serviços a outras entidades e, para isso, é preciso haver conectividade. A subseção 2.2.2 vem para falar das tecnologias mais difundidas na intercomunicação de dispositivos e que podem ser utilizadas na IoT.

A subseção 2.2.2.1 trata da pilha de protocolos TCP/IP, base das comunicações da *internet*; e a subseção 2.2.2.2 trata do padrão IEEE 802.15.4, ZigBee, que é muito difundido em redes sem fio de sensores e atuadores, do inglês *wireless sensors and actuators networks* (WSAN), e comunicação M2M.

2.2.2.1 . TCP/IP

A pilha de protocolos TCP/IP surgiu da ARPANET, projeto do governo dos Estados Unidos para trocar dados entre sistemas computacionais em diferentes geolocalizações com tolerância a falha de enlaces, e seu uso inicial era para comunicar computadores militares e universitários. A versão 4 da pilha de protocolos é utilizada até hoje e possui cinco camadas no modelo OSI: aplicação, transporte, rede, enlace e física (Kurose *et al*, 2001).

A camada de aplicação é onde ficam os programas de computador que desejam utilizar a rede para trocar informação e onde ficam os protocolos de comunicação entre aplicações como o HTTP, o SMTP, o FTP, o SSH, o SSL e o DNS. Por meio de *sockets*, eles decidem se vão enviar e receber seus dados da camada de transporte, com ou sem orientação à conexão. Grupos de dados da camada de aplicação são denominados *pacotes*.

A camada de transporte gerencia a forma como os dados vão ser enviados na rede, sendo o TCP e o UDP seus principais protocolos. O TCP troca dados com orientação à conexão, ou seja, faz o controle de fluxo, ajustando a velocidade de transmissão entre os dois fins, faz controle de congestionamento e garante a entrega de seus pacotes. Já o UDP não dá nenhuma dessas garantias e, por isso, é mais rápido que o TCP, adequando-se para atividades de *streaming* e em tempo real, em que a perda de pacote é tolerável. O pacote dessa camada é chamado de *segmento*.

A camada de rede é a responsável por tirar um *datagrama*, pacote dessa camada, de um computador, rotear ele entre os nós da rede até que chegue ao computador final. Esta camada utiliza o protocolo IP para atingir seus objetivos.

A camada de enlace é responsável por gerenciar a transmissão entre dois nós consecutivos da rede. Seu pacote se chama *quadro* e, entre seus principais protocolos, estão o *Ethernet* (IEEE 802.3), WiFi (IEEE 802.11), *bluetooth* (IEEE802.15.1), o ATM e o PPP. É importante frisar que cada enlace da rede pode funcionar com um protocolo diferente pois a camada de enlace se preocupa apenas com a transmissão entre dois pontos, formando assim um enlace de dados. A decisão de quais enlaces são tomados para transmitir dados entre dois pontos finais é de decisão da camada de rede.

A camada física é por onde passam os *bits* que compõem um *quadro*. Seus protocolos dependem dos protocolos de enlace e de qual meio físico (ar, água, fio de cobre *etc*) e em que forma esse *bit* é transmitido de um lado para o outro. Essa camada é, de fato, a transmissão física da informação entre dois nós.

A cada camada que uma mensagem do TCP/IP passa é adicionado um cabeçalho contendo informações de controle para que os dados sejam passados corretamente em cada elemento de rede e a cada camada do protocolo. Um exemplo de transmissão TCP/IP é mostrado na Figura 2.7 - Funcionamento do TCP/IP e suas camadas.

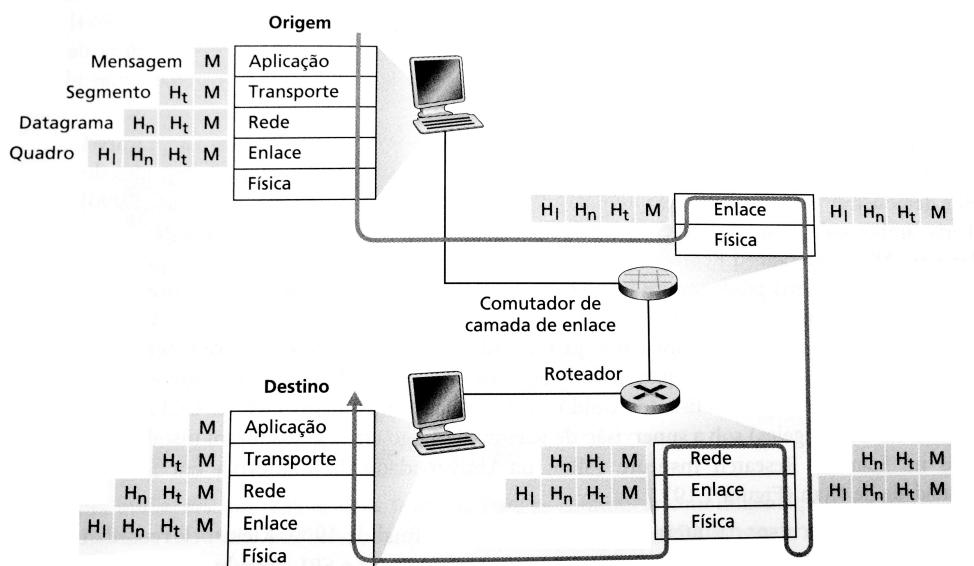


Figura 2.7 - Funcionamento do TCP/IP e suas camadas (Kurose *et al.*, 2001)

A maioria das redes locais do mundo requerem que seus nós conectem-se, utilizando TCP/IP para que possam se comunicar com as redes externas, com os outros nós disponíveis na *internet*.

Por ser o protocolo de comunicação mais difundido no mundo, o TPC/IP tem servido como base, inspiração ou padrão definitivo da maioria das aplicações de IoT.

2.2.2.2 . IEEE 802.15.4 – ZigBee

Os padrões IEEE 802.15.4 e ZigBee são esforços acadêmico e privado para prover solução de baixíssima complexidade, baixíssimo custo, baixíssimo consumo de energia e baixa taxa de dados em comunicação sem fio entre dispositivos. Enquanto o IEEE 802.15.4 provê especificação da camada física e da subcamada de enlace de acesso ao meio – do inglês *Medium Access Control* (MAC), IEEE Standard (2014) – o ZigBee cuida das

camadas acima e estende o âmbito do padrão até a camada de aplicação. De acordo com a ZigBee Alliance (2014), mais de 50% dos *chipsets* IEEE802.15.4 são produzidos utilizando as camadas do padrão ZigBee.

Essa tecnologia permite até 65.536 nós por sub-rede, pode possuir sub-redes em até 16 canais de 3MHz, trabalha em bandas de rádio livres como a banda industrial, científica e médica (do inglês *industrial, scientific and medial – ISM*), provê taxas de transmissão de até 250kbps, criptografia AES-CCM, com chave pré-compartilhada e pode trabalhar em topologias de malha, árvore e estrela.

Os microcontroladores Arduino, Raspberry Pi e Intel Galileo possuem adaptadores para trabalhar com o módulo XBee, que traz a funcionalidade de comunicação ZigBee de forma simplificada e natural para *hardware* e *software*. Uma vez configurado, com comandos seriais UART, e conectado a uma fonte de energia de 3.3V, como os microcontroladores, o módulo XBee se conecta automaticamente à rede ZigBee e envia e repassa informações, do microcontrolador para outros nós da rede, como uma porta serial UART comum, sendo o endereço da porta o endereço do nó destino.

Os protocolos ZigBee se responsabilizam por levar a informação de um nó até outro provendo segurança, garantindo a entrega e procurando sempre o melhor caminho, a melhor eficiência, e economizando energia. As três topologias de rede e os três tipos de nó de uma rede ZigBee estão representados na Figura 2.8 - Topologias e nós ZigBee.

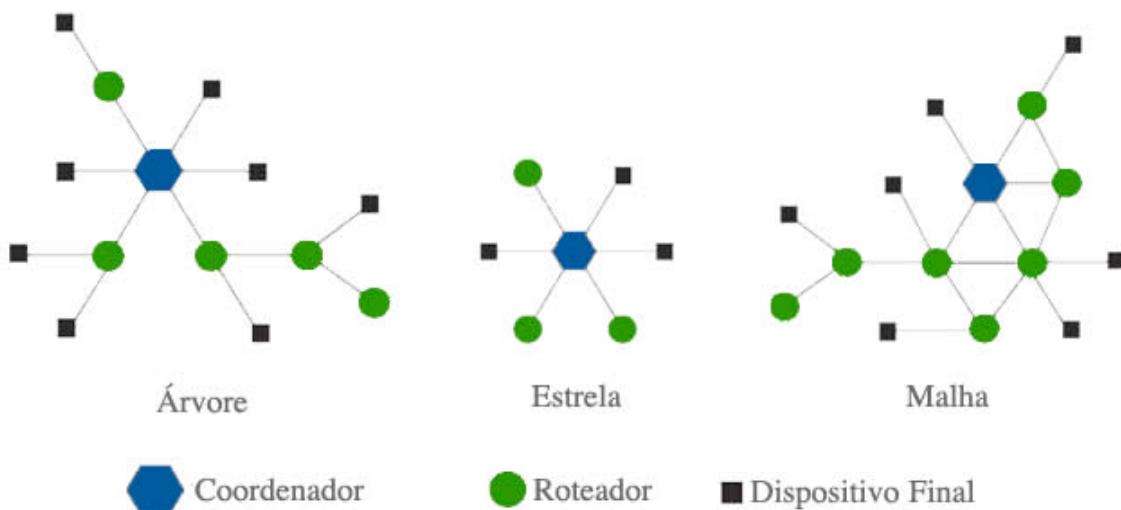


Figura 2.8 - Topologias e nós ZigBee (adaptado de Baronti *et al.*, 2007)

Os três tipos de nós são coordenador, roteador e dispositivo final. Cada rede ZigBee deve possuir apenas um nó coordenador, pode possuir vários roteadores e vários dispositivos finais. Caso um nó ZigBee perca a conexão com o nó que se emparelhou inicialmente, ele procura outro nó para se conectar automaticamente e retorna à rede.

O coordenador é responsável por iniciar uma rede em um determinado canal e pode rotear pacotes entre nós conectados diretamente a ele. Esse tipo de nó permite que outros nós façam parte da rede, conectando-se inicialmente nele. Coordenadores ficam ativos 100% do tempo e podem, além de rotear mensagens de outros, rotear mensagens geradas destinadas ou originadas por ele mesmo.

Nós do tipo dispositivo final sempre estão localizados nas pontas das redes e não precisam ficar ativos 100% do tempo, apenas quando estão transmitindo ou recebendo informações, e assim são os nós que gastam menos energia. Esses nós não podem conectar outros dispositivos à rede e não roteiam dados; apenas mandam e recebem do nó coordenador ou roteador ao qual estão diretamente conectados.

Nós do tipo roteador são capazes de adicionar novos nós à rede do tipo roteador ou dispositivo final e retransmitem mensagens entre nós diretamente conectados a ele. Por esse motivo, tal nó precisa ficar ativo 100% do tempo. Eles podem estar localizado nas extremidades ou interior da rede sem, necessariamente, terem nós conectados a ele. Caso a rede seja do tipo estrela, esse nó se comporta igual a um nó do tipo dispositivo final.

As três topologias de rede ZigBee são estrela, árvore e malha. Na topologia de estrela, os nós dispositivo final e roteador só podem enviar dados por meio do coordenador, que está diretamente conectado com todos os nós da rede. Na topologia de árvore, cada nó só pode se comunicar com seus nós-filho ou com seu pai direto. Uma mensagem enviada sobe a árvore até atingir um pai que possua o nó destino como filho, que fará a mensagem descer na rede até atingir tal filho. Na topologia de malha, os nós se comunicam diretamente com todos os nós que estão ao alcance de seu rádio. A comunicação é mais flexível sendo redirecionada para o destino pela distância física mais próxima possível.

Devido a seu custo, padronização, flexibilidade, adaptabilidade e difusão, os padrões IEEE802.15.4 e ZigBee, com suas topologias e nós, têm sido muito utilizados para aplicações de WSANs e IoT e têm-se mostrado cada vez mais como um padrão que vai perdurar em aplicações de baixa taxa de transmissão e baixo consumo de energia.

2.2.3 . TECNOLOGIAS PARA ESCRITA SEMÂNTICA DE DADOS

Objetos inteligentes com capacidade de comunicação passam a ser capazes de prestar serviços para outras entidades permitindo que elas o controlem e fiquem sabendo de seus estados. Esta seção vem para apresentar tecnologias difundidas que descrevem como a informação deve ser formatada, escrita e interpretada, para que dispositivos consigam padronizar a forma de troca de dados na camada de aplicação, permitindo assim a fácil tradução dos dados para o padrão interno das aplicações, mantendo a inteligibilidade para outros sistemas.

Tecnologias flexíveis para formatação de dados são extremamente importantes para trazer a transparência para a IoT pois são independentes do sistema operacional, da linguagem de programação, do *hardware* e do *software*. Desde que o destino e a origem sejam capazes de escrever e interpretar textos e sigam as regras de formatação do padrão, as mensagens poderão ser trocadas sem a necessidade de *frameworks* ou bibliotecas adicionais.

A subseção 2.2.3.1 traz o padrão XML do consórcio da rede de âmbito mundial, do inglês *World Wide Web Consortium* (W3C), que tem sido utilizado em larga escala na *internet* para troca de dados. A subseção 2.2.3.2 traz o formato JSON, mantido pela Associação Europeia para Padronização da Informação e de Sistemas de Comunicação, do inglês *European Association for Standardizing Information and Communication Systems* (ECMA International), também utilizado largamente por aplicações WEB.

2.2.3.1 . XML

O XML descreve uma classe de objeto de dados chamada de documento XML. Este perfil para aplicações é derivado, compatível e reduzido da ISO8879: Linguagem de Marcação Padrão e Generalizada, do inglês *Standard Generalized Markup Language* (SGML). O intuito dos Documentos XML é prover para a *web* uma formatação de dados padrão e mais abrangente que a Linguagem de Marcação de Hipertexto, do inglês *Hypertext Markup Language* (HTML), e que seja servida, recebida e processada tão amplamente quanto os documentos HTML (Bray *et al.*, 1998). Ambas as linguagens servem para estruturar os dados porém o intuito do HTML é exibir a informação, com foco em como ela deve ser

apresentada para o usuário. O XML vem para auxiliar no transporte e no armazenamento da informação, focando no que os dados representam.

Documentos XML são baseados em estruturas de linguagem chamadas *tags*. Tais estruturas são compostas por marcação de início, informação e marcação de fim. Atributos opcionais e ilimitados, contendo metadados relativos à informação, podem vir no escopo da marcação de início. Exemplos de *tags* são exibidos na Figura 2.9 - Tags.

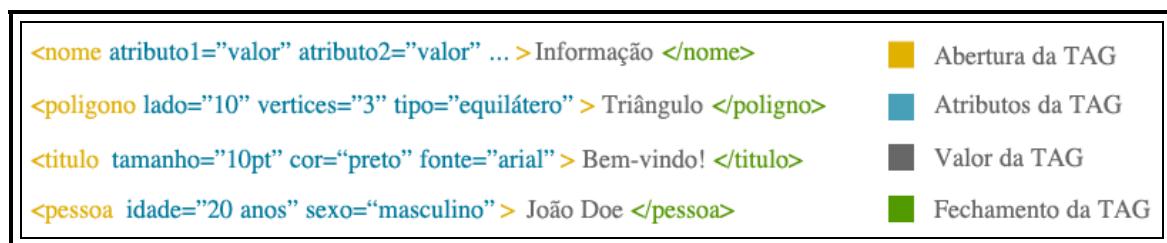


Figura 2.9 - Tags

A informação de uma *tag* pode ser um conjunto de outras *tags*, possibilitando assim o aninhamento. Essa característica permite que a abstração de estruturas complexas, como listas, árvores e registros, caibam em um documento XML. Os anexos 2 e 3 trazem, respectivamente, exemplos de XMLs utilizados pela pilha de protocolos UPnP para descrição e serviços de um dispositivo.

Documentos XML proveem marcações de fácil interpretação por máquinas e por seres humanos, podem ser escritos por aplicações simples e são flexíveis o suficiente para permitir que documentos criados por padrões de *tags* de terceiros não saiam do padrão XML. Por esses motivos, essa tecnologia tem sido aplicada em diversos modelos de serviços, em sua maioria, *web* e em protocolos orientados à prestação de serviços como o SOAP.

2.2.3.2 . JSON

O JSON é uma notação de dados leve, baseada em texto e transparente à linguagem de programação e de sistema operacional. Padrão da ECMA é utilizado em larga escala por programadores de serviços *web* assíncronos que utilizam, principalmente, JavaScript para interação de aplicação do lado do cliente. Sua vantagem está no fato de possuir tipos primitivos e estruturados (*strings*, números, *arrays*, objetos, booleanos e nulos) e uma

notação mais reduzida que o XML, causando, respectivamente, menos esforço de programação para convertê-la em um objeto de aplicação e menor uso de banda para transmissão de dados.

Tabela 1 - Tipos JSON

Tipo	Representação JSON	Exemplo JSON
Primitivo string	“chave” : “valor textual”	“nome” : “João Doe”
Primitivo número	“chave” : valor_numérico	“idade” : 20
Primitivo booleano	“chave” : valor_booleano	“erros” : false
Primitivo nulo	“chave” : valor_nulo	“mensagem” : null
Estrutura array	“chave” : [valor1, valor2, ... valorN]	“estruturas”:[“João Doe”, 32, false, null, { “longitude”: 3.2093, “latitude”: -3.094 }]
Estrutura objeto	“chave” : { “chave1”:valor1, “chave2”:valor2, ... “chaveN”:valorN }	“pessoa”:{ “nome”: “João”, “aposentado”: false, “frases”:[“oi”, “olá”], “posição”:{ “longitude”: 3.2093, “latitude”: -3.094 } }

Os tipos estruturados de um JSON são *array* e *objeto*. Os tipos primitivos são *strings*, números, booleans e nulos. Todo documento JSON tem seu escopo definido como sendo o valor de um tipo estruturado ou primitivo. Cada valor de item estruturado pode possuir outros itens estruturados ou primitivos dentro dele. Os tipos JSON, suas representações textuais e um exemplo de aplicação estão exemplificados na Tabela 1 - Tipos JSON. Assim como nos XMLs, o aninhamento é possível.

2.2.4 . TECNOLOGIAS PARA APIS E ABSTRAÇÃO LÓGICA DE OBJETOS

Objetos inteligentes capazes de comunicar-se utilizando uma linguagem que outras entidades possam entender permitem que fabricantes criem seus próprios protocolos e aplicações para gerenciar seus dispositivos. Essa seção vem para apresentar as tecnologias mais comuns para padronizar como e para onde as mensagens devem ser enviadas a fim de

obter os resultados esperados. Aqui são abordadas APIs de comunicação para permitir que aplicações tenham acesso aos recursos de um dispositivo de forma padronizada, permitindo, inclusive, a utilização de dispositivos desconhecidos pois o padrão de regras se aplica a todos os dispositivos, trazendo a interoperabilidade e a adaptabilidade para a IoT.

A subseção 2.2.4.1 apresenta o modelo de API REST, indiretamente utilizado por requisições de *websites* da *internet* e extremamente difundido em aplicações WEB. A subseção 2.2.4.2 traz a pilha de protocolos UPnP, que provê abstrações e APIs muito utilizadas para a área de acesso, fornecimento e execução de mídias entre dispositivos distintos. Na subseção 2.2.4.3, é tratado do padrão 6LoWPAN do IETF que é uma adaptação do protocolo IP para dispositivos menores, utilizando o IPv6 e o CoAP para gerência de dispositivos na camada de aplicação.

2.2.4.1 . APIs REST

REST é um conjunto coordenado de restrições arquiteturais que tenta minimizar a latência e a comunicação em rede e, ao mesmo tempo, maximizar a independência e a escalabilidade das implementações de componentes *web*. Diferente de outros estilos, o REST permite o acesso à informação e à execução de comandos em componentes, por meio da semântica direta de acesso ao serviço. A URL de acesso e o método de requisição a um serviço REST já contêm todas as ações e dados do componente a executar a ação (Fielding *et al.*, 2002).

Os serviços REST são do tipo cliente servidor, sem estado, passíveis de *cache*, feitos em camadas de execução transparentes ao cliente e que podem ser executados em entidades diferentes, e possuem *interface* uniforme e desacoplada.

Aplicado a serviços *web*, têm-se as APIs REST. O acesso aos serviços é dado por uma URL base como: <http://meuendereco.com.br/api/v2/>

O acesso às URLs do serviço pode ser feito por quatro tipos de requisições que alteram a entidade localizada: GET, PUT, POST e DELETE. O método GET lê e devolve as informações da entidade, o método PUT cria uma entidade na URL, o método POST atualiza a entidade, e o método DELETE remove a entidade. Podem-se ver exemplos de acesso a recursos, utilizando as requisições na Tabela 2 - Recursos e Requisições REST.

Tabela 2 - Recursos e Requisições REST

Recurso (Após URL base)	POST (Criar)	GET (Ler)	PUT (Atualizar)	DELETE (Remover)
/controladores	Cria novo controlador	Traz a lista de controladores	Atualiza controladores em massa.	Remove todos os controladores
/controladores/21	erro	Traz dados do controlador 21	Se existir o controlador 21, atualize-o. Caso contrário: erro!	Remove controlador 21
/controladores/21/dispositivos	Cria novo dispositivo vinculado ao controlador 21	Traz lista de dispositivos do controlador 21.	Atualiza dispositivos do controlador 21 em massa.	Remove todos os dispositivos do controlador 21
/dispositivos/7	erro	Traz dados do dispositivo 7.	Se existir o dispositivo 7, atualize-o. Caso contrário: erro!	Remove dispositivo 7.

Além da URL base, recurso e método de requisição, têm-se também os parâmetros adicionais que podem ser adicionados à URL. Em geral, é adicionado o caractere ? após o endereço do recurso, e os parâmetros são adicionados um após o outro, separados pelo caractere &, como mostrado na Tabela 3 - Parâmetros de uma URL REST.

Tabela 3 - Parâmetros de uma URL REST

Parâmetro	URL	Descrição
nome	?nome=lâmpada	Acessa recursos com nome <i>lampada</i>
estado	?estado=ligado	Acessa recursos com estado <i>ligado</i>
Estado e nome	?estado=ligado&nome=lampada	Acessa recursos com estado <i>ligado</i> e nome <i>lampada</i>

Ao final de toda a URL de acesso a um recurso, é adicionado um ponto final seguido do formato em que o conteúdo da mensagem está sendo enviado e/ou em que a mensagem deve ser respondida, por exemplo, *.json* ou *.xml*. Abaixo está um exemplo de uma URL complexa de acesso a um recurso:

<http://meuendereco.com.br/api/v2/controladores/21/dispositivos/?estado=ligado.json>

Tal URL, sendo acessada via GET, traria, em formato JSON, todos os dispositivos do controlador 21 que estão com estado ligado.

O REST traz uma maneira de acessar componentes e recursos muito simples e, em geral, utilizando apenas o protocolo HTTP, que roda em cima da pilha TCP/IP. Isso faz com que a maioria das aplicações possa acessar recursos de serviços de maneira transparente e clara e, mesmo assim, o REST é flexível o suficiente para que os serviços tenham a liberdade de tomar qualquer forma.

2.2.4.2 . PILHA DE PROTOCOLOS UPnP

O UPnP é mantido pelo fórum UPnP, conglomerado de mais de 1000 empresas que conta com a participação de corporações como a Dolby, CableLabs, Cisco, Intel, Lenovo, LG, Microsoft, Nokia, Panasonic, Samsung, Seagate, Sony, Toshiba, Verizon e Western Digital. O fórum foi criado para trazer a interoperabilidade entre dispositivos e facilitar a comunicação entre eles em uma rede local.

A pilha de protocolo trabalha em cima do TCP/IP e utiliza outros protocolos conhecidos, como o SOAP e o GENA, para abstrair dispositivos e a forma como eles interagem uns com os outros. Um dos principais pontos fortes do UPnP é o selo DLNA. Produtos com esse selo, que inclusive já estão disponíveis no mercado, são aprovados e aptos a utilizar o UPnP ou serem utilizados por ele.

O UPnP define uma arquitetura para intercomunicação de dispositivos que habilita uma rede com características pervasivas e de comunicação do tipo Peer-to-Peer, P2P (Miller *et al.*, 2001). Ele usa protocolos-padrão da *internet* (Universal Plug and Play Forum, 2000), tais como IP, TCP, UDP, SOAP, SSDP, GENA XML e HTTP, para permitir a comunicação transparente entre suas partes. Por meio da pilha UPnP, qualquer entidade capaz de ler e escrever mensagens XML para outra entidade estará pronta para controlar, ser controlada, notificar de alterações de seu estado para seus assinantes e ser notificada de alteração de estado de outras entidades das quais é assinante.

Os três blocos básicos de construção de uma rede UPnP são dispositivos, serviços e pontos de controle (Universal Plug and Play Forum, 2000). Estes blocos estão representados na Figura 2.10 - Blocos de uma rede UPnP.

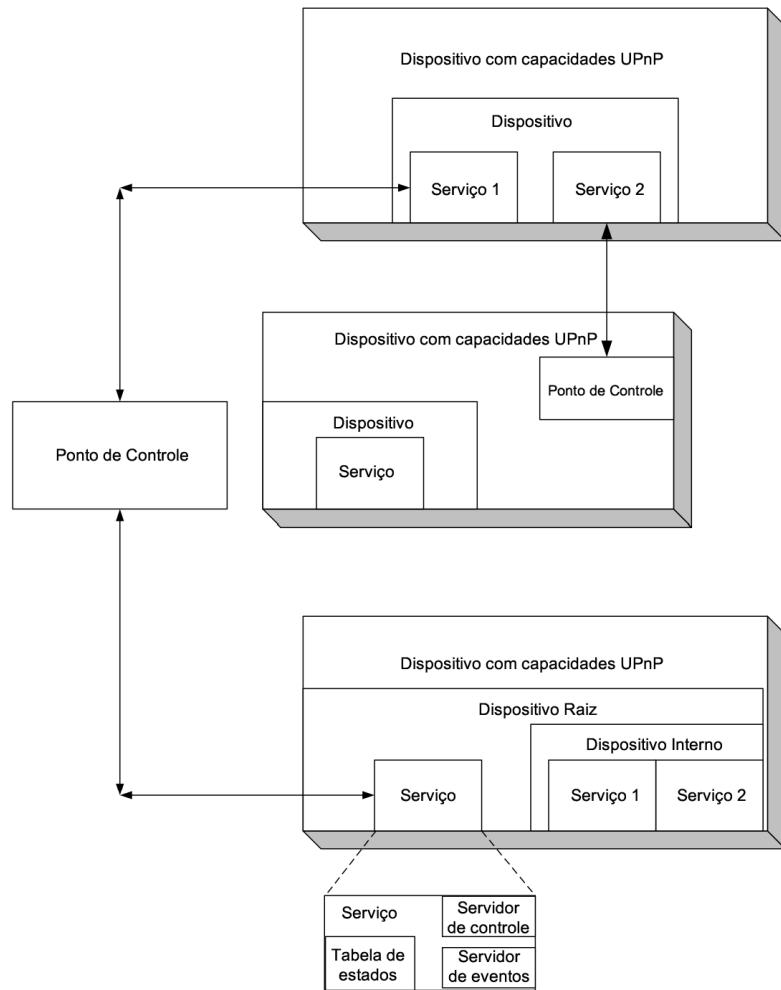


Figura 2.10 - Blocos de uma rede UPnP (adaptado de Universal Plug and Play Forum, 2000)

Dispositivos possuem serviços e podem opcionalmente conter dispositivos internos e/ou pontos de controle. Um dispositivo pode ter mais do que um serviço e mais do que um dispositivo interno. Cada dispositivo interno pode ter seus próprios serviços e seus próprios dispositivos internos.

Serviços são unidades de controle compostas por uma tabela de estado, um servidor de controle e um servidor de eventos. Serviços possuem ações que podem ser invocadas por meio de SOAP no seu servidor de controle. Uma vez que uma ação é tomada e variáveis da tabela de estado são modificadas, o servidor de eventos do serviço é responsável por notificar sobre a alteração a todos os pontos de controle que são assinantes do serviço pela arquitetura geral para notificação de eventos, do inglês *General Event Notification Architecture* (GENA). O Servidor de eventos também é responsável por gerenciar a lista de assinantes do serviço em cima de requisições HTTP (Cohen *et al.*, 2000).

Os pontos de controle são entidades que podem descobrir e controlar outros dispositivos. Eles são capazes de obter descrições de dispositivos e de seus serviços, invocar ações de controle nos serviços, e subscrever as fontes de eventos de serviços. Quando incorporado a dispositivos regulares, pontos de controles permitem a construção de redes UPnP pervasivas e capazes de reagir ao contexto, podendo assim trocar dados pertinentes e servir adequadamente a seu propósito.

Em redes locais IP com dispositivos UPnP, 6 principais operações ocorrem. São elas:

1. **Endereçamento:** Dispositivos obtêm um endereço na rede local IP, pelo DHCP ou Auto-IP.
2. **Descoberta:** Quando um dispositivo se conecta à rede, ele envia uma mensagem SSDP Broadcast, notificando a todos de sua chegada. Um ponto de controle também pode enviar uma mensagem, a qualquer momento, perguntando quem está conectado na rede.
3. **Descrição:** Pontos de controle podem pedir dados adicionais de um dispositivo ou de um de seus serviços. Para isso, ele pede o XML de descrição de um dispositivo que contém seus dados básicos, lista de dispositivos internos e lista de serviços. O ponto de controle também pode pedir informações do serviço, pela *url* da lista de serviços do XML de descrição. Caso o faça, ele receberá as ações e as variáveis daquele serviço via XML de descrição de Serviço. Um exemplo dos dois XML estão disponíveis nos ANEXO 2 - XML UPnP DE DESCRIÇÃO DE DISPOSITIVOSe 3.
4. **Controle:** Em posse dos dados de um dispositivo e de seus serviços, um ponto de controle pode pedir a um serviço que execute determinada ação, pelo XML SOAP.
5. **Eventos:** Dispositivos utilizam o GENA, através de servidores de evento de seus serviços, para gerenciar assinantes e notificá-los de suas alterações em variáveis da tabela de estado.
6. **Apresentação:** Dispositivos podem mostrar uma página *web* para os pontos de controle e/ou usuários finais, apresentando suas capacidades e permitindo seu controle através de formulários *web*.

O UPnP está passando por um enorme impulso na indústria (Universal Plug and Play Forum, 2000) e está-se tornando popular entre produtos com a certificação DLNA,

utilizada principalmente para troca de multimídia (DLNA, 2014). Como descrito acima, esta tecnologia permite qualquer tipo de dispositivo a ter comunicação externa de forma padronizada com campos bem definidos, retomando assim ao conceito de diversidade da ubiquidade e mantendo a padronização semântica. O UPnP permite o controle de dispositivo e o processamento de serviço de forma descentralizada, e permite a interoperabilidade entre pontos de controle. Além disso, a tecnologia UPnP inclui o conceito de descoberta e de descrição de dispositivos e serviços, permitindo assim que um dispositivo possa ser dinamicamente encontrado e totalmente compreendido, em termos de funcionalidade, por descrições XML.

Outro fator interessante do UPnP é o de permitir que fabricantes coloquem regras em cima da arquitetura básica que já roda em cima de protocolos padronizados, como SOAP, SSDP, GENA, HTTP, TCP, UDP e IP. As camadas e protocolos da pilha UPnP estão representados na Figura 2.11 - Pilha de Protocolos UPnP.

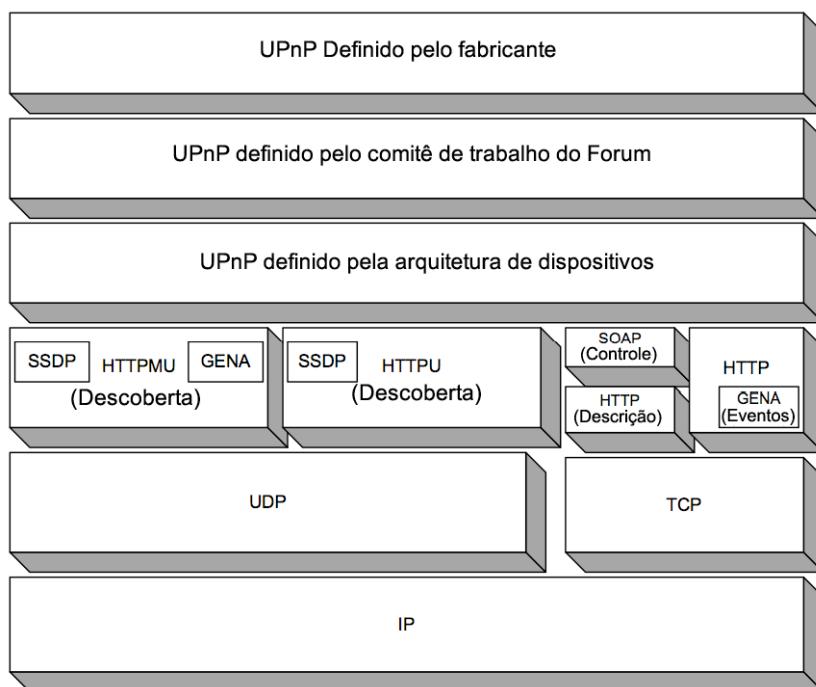


Figura 2.11 - Pilha de Protocolos UPnP (adaptado de Universal Plug and Play Forum, 2000)

2.2.4.3 . 6LoWPAN e CoAP

6LoWPAN é um protocolo do IETF para integrar o IPv6 em redes IEEE802.15.4. O grupo desenvolveu redução da pilha de protocolos para adequá-la a dispositivos de baixa capacidade computacional e permitir que eles participem da Internet das Coisas (Kushalnagar *et al.*, 2007).

As alterações foram feitas no IPv6 para que ele pudesse ser de fato utilizado em traduções no IEEE802.15.4 (Hui *et al.*, 2011). Entre as alterações estão a adaptação do tamanho dos pacotes entre as duas redes, a resolução de endereços entre as redes, a adaptação de gasto de energia e o processamento para dispositivos limitados, a camada de adaptação para os formatos de interoperabilidade e de pacotes, as considerações de roteamento e protocolos para topologias em malha, a adição de descoberta de dispositivos e serviço, e a adição de camadas de segurança como ACL.

Como se pode ver, nenhuma forma de acesso é disponibilizada pelo 6LoWPAN. Para isso, ele é utilizado em conjunto com o protocolo para aplicações limitadas, do inglês *Constrained Application Protocol* (CoAP), também regulado pelo IETF.

O CoAP provê um esquema de requisições e respostas, estilo cliente servidor para comunicação M2M, baseada em URIs para acesso de recursos que podem ser descobertos na rede por meio de notificações (Shelby *et al.*, 2012). Ele pode facilmente interagir com HTTP e mantém a simplicidade para ser utilizado em ambientes de dispositivos limitados. Se assemelha à API REST porém trabalha com tamanho de pacotes e cabeçalhos reduzidos para realizar a adaptação aos ambientes de destino. Além dessa melhoria, ele também possui a característica de descoberta de dispositivos e serviços, *broadcast* de mensagens e criptografia TLS.

Diferentemente do UPnP, o combo 6LoWPAN e CoAP não provê nenhum tipo de abstração direta de um dispositivo, o que o dá maior flexibilidade para protocolos proprietários da camada de aplicação e faz com que perca a padronização e necessite de documentação para uso de dispositivos de fabricantes diferentes. Sua vantagem reside em ser voltado especificamente para ambientes restritos, como o da Internet das Coisas, e por ser apoiado e regulado pelo IETF.

2.2.5 – Tecnologias para Hospedagem e Acesso a Middlewares e seus Dados

Além de comunicação funcional e semântica com campos bem definidos e execução garantida por microcontroladores, soluções de como hospedar inteligência e dados fora dos objetos inteligentes pode permitir que eles sejam ainda mais simples e, consequentemente, reduzam sua necessidade de processamento, armazenamento e gasto de energia.

Aplicações mais poderosas em servidores mais poderosos podem reter a inteligência, a segurança e a abstração de vários objetos inteligentes simultaneamente e servir de *interface* entre entidades que desejem utilizar um serviço e os dispositivos finais.

Esta seção vem para apresentar as tecnologias mais recentes de servidores e bancos de dados. Na subseção 2.2.5.1, será apresentada a computação em nuvem, novo paradigma computacional que se adapta perfeitamente a qualquer tamanho de aplicação. Na subseção 2.2.5.2, faremos uma rápida passagem pelos modelos de dados não relacionais, que tem mostrado performance muito superior a modelos relacionais quando trata de armazenamento de um volume muito grande de dados.

2.2.5.1 . CLOUD COMPUTING

A computação em nuvem, do inglês *Cloud Computing*, é um modelo computacional que habilita o acesso ubíquo, sob demanda e conveniente, a uma gama configurável de recursos computacionais como redes, servidores, armazenamento, aplicações e serviços. Tais recursos podem ser rapidamente aumentados ou diminuídos com um mínimo de esforço de gerenciamento ou interação do provedor do serviço para atender às necessidades de uma ou mais aplicações que estejam consumindo da nuvem. Diferente de modelos tradicionais, nos quais a computação era vista como um produto, na *cloud computing* a computação é tratada como um serviço (Mell *et al.*, 2011).

O modelo em nuvem é composto por, essencialmente, cinco características, três modelos de serviços e quatro modelos de implementação.

As cinco características essenciais da computação em nuvem são o autoserviço sob demanda, acesso amplo à rede, agrupamento e alocação de recursos entre inquilinos, elasticidade rápida e o medição do serviço prestado.

As cinco características garantem o aumento e a diminuição rápidos, com o uso de rede, de recursos compartilhados e de uso medido, de serviços que são gerenciados pelo próprio consumidor, automática ou manualmente, não obstante precisar envolver o provedor do serviço. Eles são a base para os modelos de serviço e de implementação da *cloud computing*.

Os três principais modelos de serviço da computação em nuvem são: Software como serviço (SaaS); Plataforma como serviço (PaaS); e Infraestrutura como serviço (IaaS).

O modelo de *software* como um serviço, do inglês *Software as a Service* (SaaS), é aquele em que o consumidor vai utilizar as aplicações do provedor que rodam em uma estrutura de nuvem. Um exemplo seria o GoogleDocs, *software web* para criação e edição de textos e planilhas eletrônicas.

O modelo de plataforma como um serviço, do inglês *Platform as a Service* (PaaS), é aquele no qual o consumidor desenvolve sua própria aplicação utilizando plataforma, linguagem de programação, biblioteca, serviço e/ou ferramentas oferecidos pelo provedor em seu ambiente em nuvem. Basta subir a aplicação no servidor que ela rodará. O provedor PaaS cuidará dos detalhes de implantação, provisionamento de capacidade, balanceamento de carga, autoescalamiento e do monitoramento de saúde do aplicativo.

O modelo de infraestrutura como um serviço, do inglês *Infrastructure as a Service* (IaaS), é aquele no qual o consumidor utiliza recursos básicos de *hardware*, tais quais armazenamento, processamento e rede, providos pela estrutura em nuvem do provedor. Neste modelo, o consumidor tem liberdade de fazer como bem entender sob os *hardwares*, sem ferir o SLA. No IaaS, o desenvolvedor é responsável por configurar o sistema operacional, seus pacotes e extensões, e por monitorar todos os *softwares*. Nesse caso o provedor IaaS cuidará de prover o *hardware* bem como sua autoescala e balanceamento de carga, necessários para que a aplicação, o sistema operacional e suas dependências possam funcionar para qualquer tipo de demanda de uso.

Os quatro modelos de implementação são: nuvem privada, nuvem comunitária, nuvem pública e nuvem híbrida.

Uma nuvem privada é aquela de apenas uma empresa ou organização. Ela hospeda aplicações privadas e pode receber segurança como se fosse um perímetro computacional tradicional privado.

Uma nuvem comunitária é uma na qual o uso é exclusivo de organizações seletas que compartilham os recursos entre si. Da mesma forma que a nuvem privada, seu perímetro é privado entre as organizações participantes.

Uma nuvem pública é aquela na qual qualquer um tem acesso aos recursos compartilhados. É o tipo mais comum e utilizado atualmente pela maioria dos grandes aplicativos *web*. Sua segurança é dificultada pela sua publicidade pois, por ser uma tecnologia nova e em ascensão, a computação em nuvem ainda possui muitos desafios e problemas a serem descobertos e tratados e, que no caso de rede pública, recai sobre os provedores do serviço. Já a nuvem híbrida é a cuja estrutura é composta por mais de um tipo de nuvem (privada, comunitária ou pública) simultaneamente. Sua segurança é dada pelos tipos de nuvem que estão sendo misturadas.

Um dos fatores mais interessantes da computação em nuvem para a internet das coisas é o da rápida evolução computacional e de métodos para compartilhamento de recursos e autoescala. Ao utilizar-se das características da *cloud computing*, a IoT poderá ter servidores em *cluster* compartilhados com outras aplicações que poderão autogerenciar os recursos alocados para a IoT sem deixar nada a desejar para as entidades que estão tentando acessar serviços de dispositivos. Um *middleware* que fique entre os dispositivos e as entidades utilizadoras de serviços poderá ter sua capacidade computacional, largura de banda e capacidade de armazenamento, crescendo ou diminuindo, de acordo com a demanda por serviços de dispositivo, podendo assim retirar a maior parte da inteligência dos dispositivos. Aliado à tecnologia ZigBee para repassar dados a um microcontrolador que executará a informação no objeto final, o *middleware* poderá guardar os estado e as abstrações lógicas e prover a segurança necessária para os dispositivos, que poderão ser mais vulneráveis e menos potentes.

2.2.5.2 . NoSQL

Bancos de dados relacionais que utilizam a linguagem de consulta estruturada, do inglês *Structured Query Language* (SQL), foram utilizados para a maioria dos modelos de armazenamento de dados no passado. Tal acontecimento tem gerado problemas de complexidade crescente, principalmente para aplicações cuja latência deva ser a menor possível, quando se trata de grande quantidade de informação a ser armazenada e grande

volume de leitura e escrita massiva de informações no disco rígido. Os modelos de dados normalizados e com suportes completos a Atomicidade, Consistência, Isolamento e Durabilidade (ACID) não se encaixam mais em sistemas que necessitem ser distribuídos, ter alta performance e ter alta disponibilidade (Hecht *et al.*, 2011).

Devido a essas demandas, muitas organizações desenvolveram seus próprios sistemas de armazenamento de dados, classificados como Não Somente SQL, do inglês *Not Only SQL* (NoSQL).

Atualmente, os modelos de dados mais utilizados são os bancos de dados relacionais SQL e quatro principais grupos de NoSQL: Chave-valor, Documentos, Famílias de Colunas e Banco de Dados de Grafos.

O modelo de dados relacional SQL é o mais famoso e utilizado no mundo. SGDBs como PostGree, MySQL e SQLServer o utilizam e dão suporte total à linguagem de consulta estruturada. São baseados em modelo de dados relacional, normalizados, e dão suporte completo à ACID, o que faz com que não sejam escaláveis horizontalmente. Este modelo permite a busca completa dos dados, dando suporte a relacionamento em suas consultas por meio do SQL.

O modelo de Chave-valor endereça os dados por chave única. Dados são inseridos e retirados do armazenamento por uma chave. Por esse motivo, o modelo Chave-valor é independente de esquemas de dados e não possui dados indexados para busca. Este modelo foi criado pela Amazon para ser utilizado em seu *site* internacional de compra e venda de produtos, especificamente no armazenamento e no gerenciamento de carrinhos de compra. Os armazenadores mais famosos desta categoria são o DynamoDB, o RIAK e o Redis.

O modelo de Documentos encapsula pares chave-valor em documentos JSON e, por esse motivo, se diferenciam da Chave-valor por possuírem dados buscáveis e um relacionamento fraco entre seus dados. Os armazenadores mais famosos são o MongoDB e o CouchDB.

O modelo Famílias de Colunas foi inspirado no BigTable, inventado pela Google, e possui armazenadores famosos como o Cassandra e o HBase, feito pela equipe do Apache através do Hadoop. Este modelo guarda um numero arbitrário de pares chave-valor em linhas, formando um conjunto de colunas que, quando agrupadas, formam uma família de coluna.

Essas famílias são utilizadas para fazer o particionamento e a organização inteligente da informação. Valores também são armazenados com informações de data e de hora da inserção, para auxiliar no versionamento, performance e consistência dos dados.

Bancos de dados de Grafos são utilizados para armazenar informações altamente conectadas, com muitos relacionamentos, pois substituem JOINS por transversais eficientes. Seu principal representante é o NEO4J.

Devido a seu modelo mais simplificado e facilidade de distribuição e replicação, modelos NoSQL costumam ser muito mais velozes e tolerantes a falhas que modelos relacionais tradicionais (Leavitt, 2010).

2.2.6. TECNOLOGIAS PARA IDENTIFICAÇÃO DE OBJETOS

Além da transformação de um objeto simples em inteligente, existem sistemas que servem para identificar um objeto e, dependendo do contexto em que estão inseridos, para eles é dado um estado.

As principais tecnologias de identificação são o código de barras, QRCode e o RFID. O código de barras é o mais antigo e utiliza uma leitura linear de cores para identificar os dados. O QRCode faz uma leitura em duas dimensões para identificar os dados. O RFID funciona através de envio e leitura de rádio frequência para identificar os dados.

O início do termo IoT foi dado pelo uso de sistemas tecnológico baseados em RFID. Tecnologias de identificação são muito utilizadas em logística para saber onde um objeto está e em que estágio do transporte em um determinado ambiente ele está. Cada objeto possui identificador único que é escaneado para o sistema, utilizando leitores códigos de barras, de QRCode ou RFID. Uma vez escaneado no sistema, o usuário diz em que estágio aquele objeto se encontra e assim o sistema toma todas as medidas para notificação de troca de estado e gestão de histórico e abstração do objeto. Esses foram os primeiros sistemas implementados de IoT e, certamente, os mais difundidos até hoje.

2.3. ARQUITETURAS EXISTENTES DE IoT

Para permitir que a Internet das Coisas se torne realidade, é preciso que seja padronizada a forma como os objetos devem funcionar para que possam prestar serviços para outras

entidades. É preciso definir qual será a forma de abstração lógica de dispositivos, comunicação com o mundo exterior, como o acesso aos serviços deverá acontecer e todos os protocolos necessários para tornar um objeto simples em um objeto inteligente completo. Para tal problema existem modelos teóricos e práticos que englobam todos os conceitos e códigos necessários para que um ambiente inteligente seja gerado.

Esta seção vem para apresentar as três principais arquiteturas de Internet das Coisas. A seção 2.3.1 trata da implementação em código aberto *LinkSmart*, que foi feita para controle de dispositivos pelos *webservices*. A seção 2.3.2 trata do modelo arquitetural IoT-A, que foca na interoperabilidade de dispositivos. A seção 2.3.3 trata do modelo iCORE, que foca na IoT pela perspectiva de tecnologias cognitivas e semânticas.

2.3.1. Hydra – LynkSmart

Inicialmente chamado de Hydra Middleware, o LinkSmart é uma plataforma escrita em Java para interoperabilidade de dispositivos. Ela objetiva a construção de sistemas de alto desempenho e baixo custo para construção de objetos inteligentes e Internet das Coisas. O LinkSmart é um *middleware* baseado no OSGi para a criação de redes de heterogêneas através de *webservices* de baixo acoplamento (Hydra Middleware, 2014).

O LinkSmart utiliza protocolos proprietários para fazer a comunicação entre seus dispositivos conectados e pede que um dispositivo utilize o *framework* ou que seja colocado atrás de um *proxy* para funcionar. Uma vez que a configuração esteja feita, qualquer dispositivo LinkSmart, via *proxy* ou não, pode ser controlado por de *webservices* em sua arquitetura SOA.

No caso do *proxy*, é preciso ter uma máquina hospedeira com, no mínimo, 1GB de memória RAM livre e será preciso escrever um *driver* de comunicação em Java. Pelo *driver*, o hospedeiro LinkSys se torna capaz de trocar dados com o dispositivo final e passa a colocá-lo em sua rede. É suportada a conexão com dispositivos com USB, ZigBee e *bluetooth*.

No caso sem *proxy*, o dispositivo deve possuir, no mínimo, 256MB de RAM livre. Tal dispositivo deve hospedar os códigos Java e bibliotecas LinkSys e OSGi e assim será considerado um dispositivo nativo LinkSys.

Para seus dispositivos, o Middleware LinkSys dá suporte a rede e eventos, fornece segurança dos dados, realiza descoberta de *proxys* e dispositivos, e permite a configuração de funcionamento de cada componente separadamente. Uma imagem com uma rede típica LinkSys está representada na Figura 2.12 - Arquitetura LinkSmart.

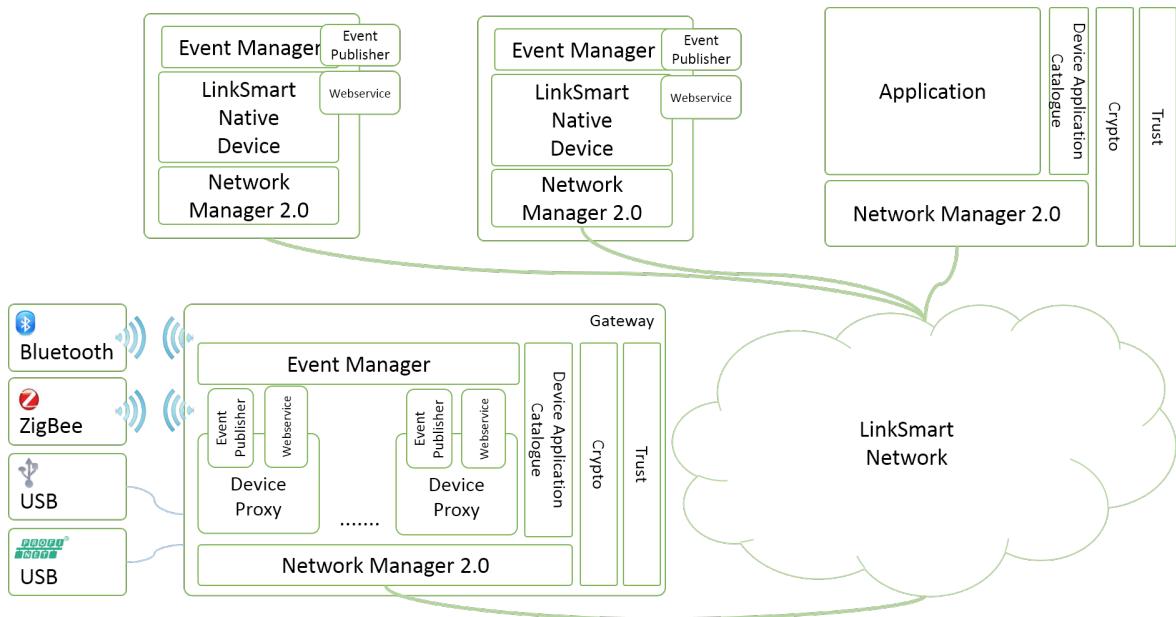


Figura 2.12 - Arquitetura LinkSmart (LinkSmart Wiki, 2014)

Entre suas vantagens, estão a facilidade de codificação e a atividade recente e constante em sua melhoria. O fato de possuir padrões internos pode prejudicar sua evolução futura por outras pessoas, uma vez que tem difícil entendimento e pouca documentação de funcionamento interno. Outro ponto a levantar é o de requisitos de memória RAM que podem acabar impactando para dispositivos limitados provenientes da Internet das Coisas.

2.3.2. IoT-A

Iniciado em setembro de 2010 e finalizado em novembro de 2013, o projeto europeu IoT-A foi destinado a criar um modelo de referência de arquitetura para a Internet das Coisas. O grupo, formado por membros do setor privado e de universidades europeias, recebeu apoio do sétimo programa-quadro para pesquisa (FP7), da Comissão Europeia, e focou no desenvolvimento de uma arquitetura que fosse capaz de trazer interoperabilidade entre dispositivos, entidades e sistemas distintos com a redução em componentes menores que entregassem dados em uma gramática comum (IoT-A, 2014).

Tal arquitetura visou proteger os dispositivos e os abstrair de forma atômica para que possa ser aplicada a qualquer tipo de objeto, tendo assim a maior heterogeneidade possível. Na IoT-A, estão previstos o uso de muitas funcionalidades presentes em APIs REST e UPnP e protocolos como o 6LoWPAN como serviços *web*, por meio de URIs a serem providos por dispositivos para seu controle, gerenciamento de estado, assinantes de alterações de estado de dispositivos e adaptação para dispositivos limitados em *hardware* e em *software*.



Figura 2.13 - Árvore da IoT-A (IoT-A, 2014)

A IoT-A tem o intuito de transformar as tecnologias existentes em serviços para os consumidores finais, deixando as tecnologias base escondidas de quem está utilizando os serviços, apenas tendo que interfacear, via *middleware*. Este modelo é exemplificado pelo grupo através da Figura 2.13 - Árvore da IoT-A.

2.3.3 – iCORE

Também contemplado pelo FP7, o iCORE foi iniciado em outubro de 2011 e tem previsão de finalização em outubro de 2014. Este modelo pretende gerar uma arquitetura modelo que parte do princípio de Objetos Virtuais (VOs), que são equivalentes a Objetos

inteligentes, e Conjunto de Objetos Virtuais (CVOs). Seu diferencial em relação à IoT-A está no fato de focar na cognição de modelos de VOs para formar CVOs que sejam semanticamente compatíveis e possam esconder a complexidade tecnológica que reside nos VOs (iCore, 2014).

Os sete objetivos do iCORE são: prover um *framework* para gerência de cognição de VOs interconectados; prover modelos de VOs; prover modelos de CVOs; levantamento de requisitos das principais demandas de usuários e outras partes interessadas; prover um protocolo de segurança para o *framework* inteiro; validar em ambientes reais os modelos gerados; criar um padrão e disseminar na união europeia para aumentar a competência local sobre o tema de IoT.

Atualmente, o projeto se encontra em fase de finalização e já possui um modelo de referência de arquitetura que estende e melhora o modelo proposto pela IoT-A.

2.4. PROBLEMAS EM ABERTO

Os três principais problemas para arquitetura de Internet das Coisas são a interoperabilidade, a escalabilidade e a segurança (Atzori *et al.*, 2010). Uma arquitetura de sucesso deve endereçá-los e fazer isso utilizando o máximo de tecnologias e padrões já existentes para que possa ser evoluída por partes e por grupos diferentes.

A interoperabilidade é fundamental para que sistemas inteligentes possam utilizar dispositivos distintos sem muito esforço de aprendizado ou, para a maioria dos casos, automaticamente. Como parte da interoperabilidade estão a agilidade e a adaptabilidade do sistema. A agilidade é importante quando se tratar do uso de dispositivos em tempo real. A arquitetura precisa ser capaz de realizar os pedidos das entidades em tempo hábil. A adaptabilidade faz com que a arquitetura seja capaz de englobar qualquer tipo de dispositivo embaixo de sua alcada para prover um serviço igual para aplicações que usem os dispositivos por meio dela (Miorandi *et al.*, 2012).

A escalabilidade é necessária para que o *middleware* possa englobar redes de dispositivos e demandas por serviço de qualquer magnitude. É preciso que a arquitetura se adapte a cenários de poucos, ou muitos, dispositivos, poder computacional e entidades requisitantes de serviço. É necessária a adequação física a esses modelos, o uso inteligente dos recursos e o arranjo de componentes para sempre servir ao usuário final satisfatoriamente. Como

parte da escalabilidade, encontra-se o uso inteligente dos recursos. Para que objetos possam ser utilizadas com o mínimo de recurso possível, assim adentrando os cenários mais limitados e estudados na IoT, é preciso ter isso como premissa (Gubbi *et al.*, 2013)

A segurança faz com que aplicações e usuários finais possam adotar serviços de dispositivos por meio da arquitetura (Atzori *et al.*, 2010). Para garantir a continuidade da arquitetura é preciso adequar-se às premissas da confidencialidade, privacidade, autenticidade e integridade da informação.

As arquiteturas de *middleware* aqui exibidas não possuem modelos de fácil implementação ou escaláveis para qualquer cenário. Todas oferecem um certo grau de interoperabilidade, segurança e escalabilidade, mas não endereçam problemas de implementação ou agilidade ou adaptabilidade e, muitas vezes, quando o fazem, utilizam protocolos e tecnologias proprietárias que impedem seu desenvolvimento por terceiros. Portanto, é preciso que um modelo de arquitetura, que seja implementável e que utilize tecnologias padronizadas e difundidas, seja proposto, endereçando os problemas de interoperabilidade, segurança e escalabilidade.

2.5. SÍNTESE DO CAPÍTULO

O objetivo deste capítulo foi apresentar os principais conceitos e as principais tecnologias que se relacionam com a Internet das Coisas. Foram tratados conceitos de computação ubíqua, Internet das Coisas e suas visões, tecnologias habilitadoras para a produção física e lógica de objetos inteligentes e como dar a eles infraestrutura para a comunicação adequada com outras entidades.

Aqui, foram trazidas as principais arquiteturas de *middleware* existentes para Internet das Coisas e os principais desafios a serem enfrentados nessa área que ainda não foram totalmente contemplados por soluções existentes.

3. PROPOSTA DE *MIDDLEWARE* PARA INTERNET DAS COISAS

Este capítulo apresenta a arquitetura de *middleware* proposta para a Internet das Coisas. Descreve-se como utilizá-la para construir sistemas reais e como aplicá-la em um cenário-modelo, englobando um conjunto de componentes representativo dos dispositivos, componentes, protocolos e sistemas discutidos no capítulo anterior.

A Figura 3.1 - Proposta de Middleware traz uma visão de como o *middleware* proposto integra dispositivos e aplicações. As duas rotinas fundamentais são a de controle e a de monitoramento de estado de dispositivos, apresentadas inicialmente a seguir pois dão um entendimento geral da estrutura e funcionamento da proposta desta dissertação.

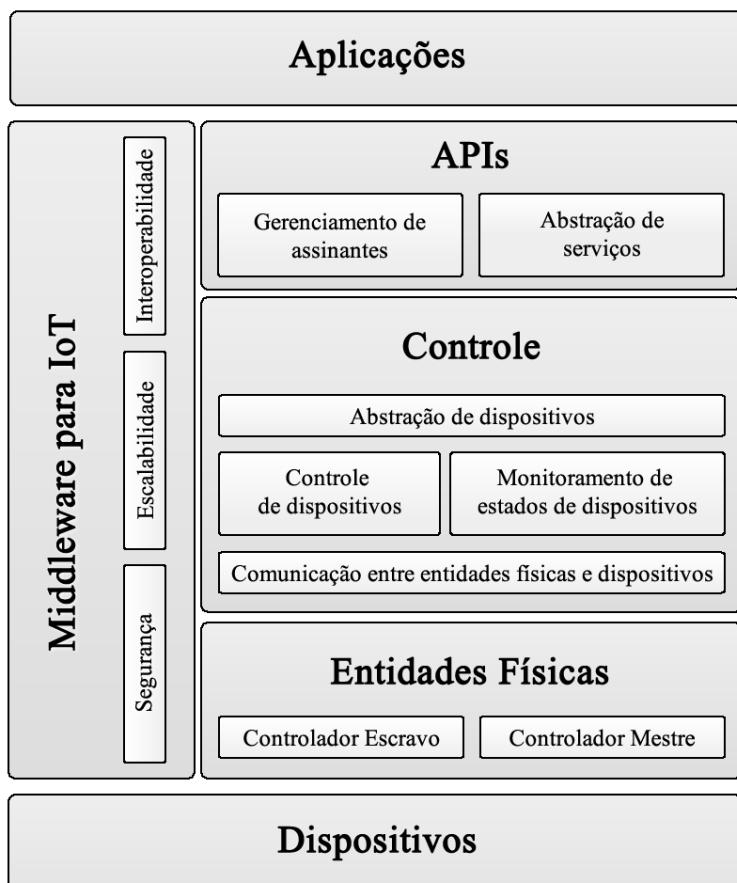


Figura 3.1 - Proposta de Middleware

A primeira rotina consiste em enviar requisições de controle de uma aplicação para um dispositivo. Aplicações trocam dados com o *middleware* através de abstrações de serviços providas por APIs. O middleware traduz as requisições de aplicações em comandos executáveis e transmite os comandos corretos para os dispositivos alvo. Esse repasse de dados ocorre de um controlador mestre, entidade física que conversa com aplicações, para

um controlador escravo, entidade física que conversa com dispositivos. Para isso, o middleware utiliza suas abstrações lógicas de dispositivos e repassa os comandos entre as entidades físicas, que executarão os comandos no dispositivo alvo. Uma vez finalizado esse processo, o middleware retorna uma resposta para o dispositivo notificando a conclusão da operação.

A segunda rotina consiste em monitorar o estado de um dispositivo e, quando perceber uma alteração, notificar as aplicações desse novo estado. Para isso, as entidades físicas monitoram o estado de um dispositivo e, quando percebem a modificação, comunicam o novo estado de um controlador escravo para um controlador mestre, que notificará aplicações. Isso é feito utilizando a comunicação entre entidades físicas, abstração e dispositivos e o gerenciamento de assinantes. Este último bloco é responsável por saber quais aplicações, dentre as que se comunicam através de sua API, estão interessadas em saber o estado de quais dispositivos e, quando houver alterações, notificar tais dispositivos do novo estado.

A seção 3.1 traz uma visão detalhada de como a arquitetura de middleware funciona, de um ponto de vista físico e lógico, traz o modelo de abstração de dispositivos e exemplos de sistemas e abstrações implementáveis que utilizam a solução proposta.

3.1. ARQUITETURA PROPOSTA

De acordo com a revisão apresentada em Weiser (1991), Lyytinen (2002), e Atzori *et al.*, (2010) e com as arquiteturas relacionados Hydra Middleware (2014), IoT-A (2014) e iCore (2014), a IoT demanda por uma arquitetura de *middleware* que traga pervasividade e mobilidade para dispositivos, que seja transparente e segura para dispositivo e aplicações, que seja extensível a qualquer tipo de dispositivo e que seja escalável para atender a qualquer demanda de dispositivos e aplicações.

A extensibilidade e a transparência para qualquer dispositivo fará com o *middleware* abranja grande variedade de objetos abaixo de si, como mostrado na Figura 2.13 - Árvore da IoT-A (IoT-A, 2014), tornando assim o serviço prestado para aplicações externas, acima do *middleware*, uniforme e padronizado. Adicionando APIs transparentes ao *middleware*, aplicações poderão ser escritas, independente de sua plataforma, para utilizar os

dispositivos para entender o contexto em que estão inseridos e executar as ações devidas, trazendo assim pervasividade ao sistema.

A pervasividade unida às tecnologias de comunicação móvel, como o ZigBee (ZigBee Alliance, 2014), para comunicar as partes dentro do *middleware*, fará com que dispositivos se tornem ubíquos (Lyytinen, 2002) e tenham interoperabilidade pois passarão a individualmente ou em conjunto entender o contexto em que estão inseridos, a comunicar-se uniformemente e a prestar serviços adequados e convenientes aos usuários finais, onde quer que estejam.

Neste trabalho, propõe-se uma arquitetura lógica, física, pervasiva e móvel, para controle e notificação de estado de dispositivos de maneira transparente, extensível e escalável. O presente trabalho tem por finalidade permitir que aplicações, em tempo real, controlem e sejam notificadas do estado de dispositivos. Com a utilização da arquitetura aqui proposta, dispositivos, mesmo os dotados de pouca ou nenhuma capacidade de processamento ou fonte de energia, tornar-se-ão pervasivos e móveis de maneira transparente, extensível e escalável.

A escalabilidade, a extensibilidade e a transparências são devidas à forma como se dá a comunicação com aplicações externas e dispositivos. A arquitetura possui APIs e *interfaces seriais* analógicas e digitais para comunicação com *hardwares*. As APIs REST e UPnP utilizadas por este trabalho são independentes de sistema operacional ou linguagem de programação, são transparentes e, por serem feitas por tecnologias da *web*, podem usufruir de técnicas de escalabilidade da computação em nuvem. As *interfaces* de *hardware* permitem que novos dispositivos possam ser acoplados à arquitetura, estendendo assim suas funcionalidades e abrangência.

A arquitetura física foi pensada para dar flexibilidade ao sistema. Possuir apenas um controlador poderia trazer a necessidade de espalhamento de muitos fios para alcançar os dispositivos que fossem adicionados à arquitetura. Outro problema de se possuir apenas um controlador seria o de realizar o processamento completo por uma entidade. Por esse motivo, a arquitetura aqui proposta é compostas por dois tipos de entidades controladoras: controlador-mestre e controladores-escravo. O controlador-mestre retém as APIs e a abstração lógica dos dispositivos. É nele onde ocorre a maior parte do processamento e onde se localiza a maior parte da inteligência da arquitetura. Os controladores-escravo se

conectam fisicamente a um ou vários dispositivos e são responsáveis exclusivamente por ler estados de dispositivos e executar ações neles. Controladores-mestre e escravo podem se comunicar via rede sem fio para gerenciar todos os dispositivos e prestar serviços. Tal característica traz a mobilidade pois os controladores-escravo e os mestres podem trocar sua posições geográficas sem necessariamente perder a conectividade. A portabilidade dos códigos das entidades da arquitetura faz com que ela possa ser incorporada tanto em sistemas simples, como um microcontrolador, como em sistemas complexos, como um *cluster* de servidores, utilizando tecnologias de replicação e balanceamento de carga sob demanda.

A abstração lógica dos dispositivos conectados à arquitetura é feita utilizando o padrão UPnP. Nessa arquitetura, cada dispositivo possui serviços a oferecer, que podem ser executados por meio de ações que possuem argumentos/parâmetros, a serem recebidos e enviados, que alteram suas variáveis de estado. Essa abstração permite que qualquer dispositivo seja escrito em linguagem de máquina e seja utilizado por aplicações. Dispositivos controlados por inteligência de aplicações podem funcionar sem intervenção humana, fazendo então com que se tornem pervasivos para os humanos.

Uma premissa para a construção da proposta de arquitetura aqui apresentada é a de utilizar o máximo de tecnologias já existentes e difundidas. Com isso, pretende-se ter um *middleware* implementável, com maiores chances de sobrevivência a médio prazo, que possa se utilizar da documentação e pesquisa dessas tecnologia e que seja beneficiado sempre que qualquer uma delas evoluir.

Foi feito um esforço de padronização de funcionamento e implementação para que novos desafios relativos à Internet das Coisas sejam descobertos pois não existe nenhuma arquitetura implementada que seja capaz de ser totalmente entendida, melhorada e utilizada para desvendar esses novos desafios, como o modelo de tráfego gerado e problemas relativos à segurança.

Na seção 3.1.1, será apresentada a forma física e como os componentes se conectam na proposta de *middleware*. Na seção 3.1.2, será apresentado como um dispositivo físico é abstraído para sua forma lógica. Na seção 3.1.3, serão apresentadas todas as partes lógicas do sistema e como elas se integram para possibilitar a execução de serviços. Na seção 3.1.4, serão dados exemplos de como sistemas podem utilizar a arquitetura. Na seção 3.1.5,

será mostrado um cenário de aplicação-modelo, contemplando todos os sistemas apresentados na seção 3.1.4. Na seção 3.1.6, será feita uma introdução de como adicionar segurança básica à arquitetura.

3.1.1. ESTRUTURA FÍSICA

A separação física dos controladores da arquitetura se fez necessária para a economia e melhor utilização dos recursos físicos, descentralização do gerenciamento e do processamento, e para dar mobilidade para os dispositivos.

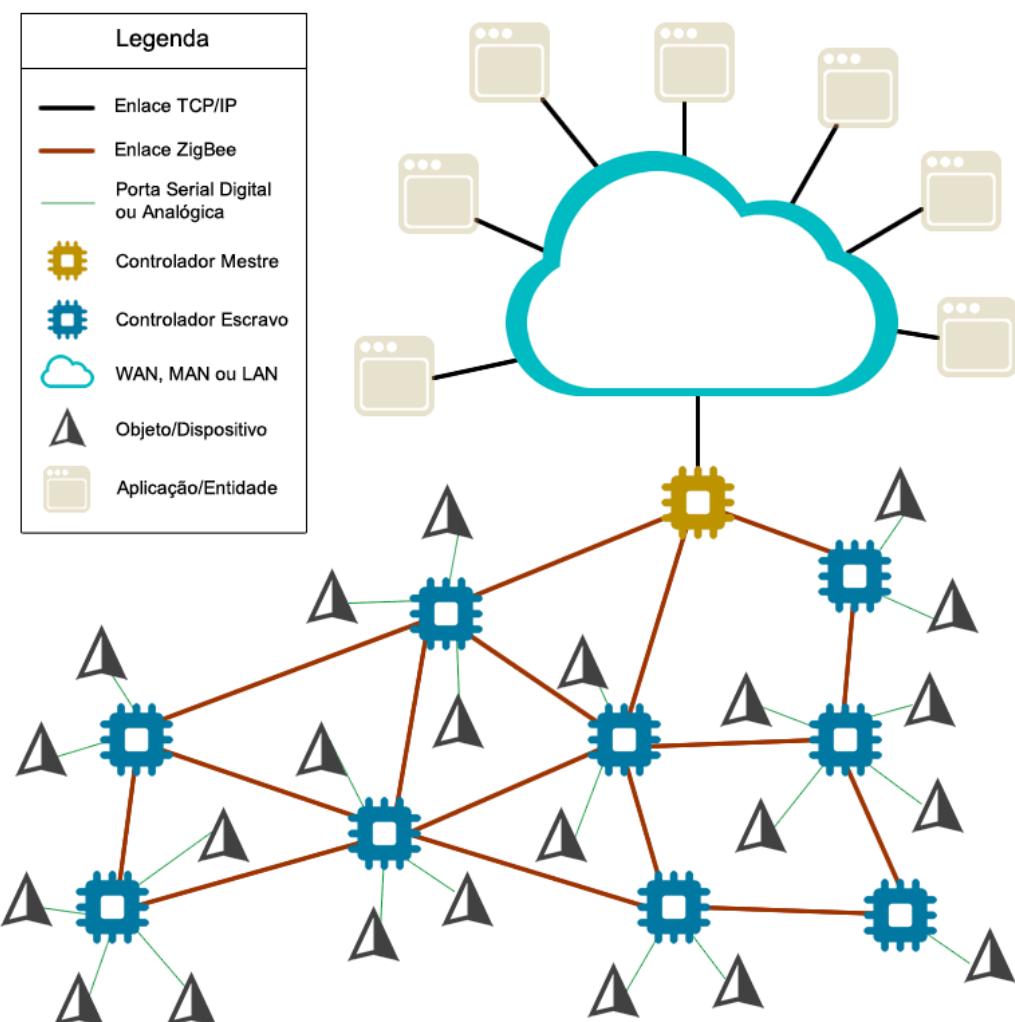


Figura 3.2 - Visão Física da Arquitetura Proposta

Manter todo o processamento e conectar todos dispositivos diretamente a um controlador único faria com que tecnologias de *cluster* de computadores não pudessem ser aplicadas à arquitetura pois as portas físicas para comunicação digital e analógica com dispositivos não

podem ser replicadas como os códigos da arquitetura. Isso traria a necessidade de multiplexadores e de multiplexadores complexos que utilizariam algoritmos proprietários para atingir os dispositivos finais corretamente.

Por esse motivo, foram utilizados vários controladores que se conectam utilizando a tecnologia ZigBee em redes no formato de malha. Com isso, a arquitetura se beneficia de toda a produção de *hardwares*, toda a documentação e a comunidade ativa para tal tecnologia. Qualquer evolução nesse padrão beneficiará diretamente a comunicação interna entre os controladores da arquitetura, tornando-a mais eficiente e passível de evolução.

Controladores diretamente conectados a dispositivos, denominados controladores-escravo, trocam informações com um controlador-mestre que, além de conectar-se com a rede ZigBee, também se conecta com redes IP, de forma semelhante ao 6LoWPAN, porém sem fazer a tradução direta de pacotes entre as duas redes.

Com a separação dos controladores, as partes internas da arquitetura podem gerenciar atividades de controle diferentes, distribuindo parte do processamento. Com isso, a mobilidade também é dada dispositivos que, levando o controlador-escravo junto, podem locomover-se fisicamente sem perder a conectividade à rede ZigBee, bastando apenas estar ao alcance do rádio de qualquer outro controlador.

A Figura 3.2 traz uma visão de como os componentes físicos da rede se conectam. Aplicações, por redes IP, podem requisitar serviços para o controlador-mestre, que repassará para os comandos devidos para o controlador-escravo correto, via rede em malha, que executará a ação no dispositivo correto.

Na seção 3.1.1.1, as funcionalidades e as responsabilidades do controlador-mestre são descritas e, na seção 3.1.1.2, as dos controladores-escravo.

3.1.1.1 . CONTROLADOR-MESTRE

Para arquitetura de *middleware* proposta, o controlador-mestre é o equivalente à cabeça de uma pessoa. Ele se comunica com o resto do corpo e com o mundo exterior. Esta entidade é responsável por:

- Trocar informações com qualquer aplicação que consiga comunicar-se, utilizando API REST ou UPnP, via redes TCP/IP;

- Criar, manter e atualizar a abstração lógica de dispositivos (representação no ambiente virtual que será utilizada nas APIs);
- Comunicar-se com os controladores-escravo;
- Tratar informações de requisições de aplicações, criar os comando corretos para o controlador-escravo e por enviar os dados finais ao controlador-escravo correto do dispositivo;
- Retornar o último estado lido de um dispositivo para uma aplicação que requisite tal dado;
- Registrar aplicações que desejem receber atualizações de estados de dispositivos;
- Notificar alterações nos estados de um dispositivo para todas as aplicações registradas para receber tais notificações.

Sua forma física pode variar de acordo com o cenário de aplicação desejado. Para um cenário pequeno, como uma casa, o controlador-escravo pode ser um microcontrolador como a Raspberry Pi. Para cenários grandes, como uma cidade, ele pode assumir a forma de um *cluster* de servidores e utilizar técnicas da computação em nuvem como平衡adores de carga, servidores de arquivos estáticos, servidores de cache de arquivos estáticos, servidores de cache de dados, servidores de banco de dados e servidores de aplicação. Para a armazenagem de dados de dispositivos e serviços, podem ser utilizados desde arquivos XML a bancos de dados NoSQL, que também permitem replicação e são tolerantes a falhas.

A demanda do *hardware* para um controlador-mestre é que seja capaz de operar interface com redes TCP/IP e com redes ZigBee. Demandas de espaço em disco, processamento e memória RAM são variáveis de acordo com a demanda e podem utilizar as técnicas descritas no parágrafo anterior.

3.1.1.2 . CONTROLADOR-ESCRAVO

Para arquitetura de *middleware* proposta, o controlador-escravo é o equivalente aos membros de uma pessoa. Ele se comunica com os objetos e executa, de fato, os comandos enviados pelo controlador-mestre, mesmo sem plenamente entender o que ele está fazendo. Esta entidade é responsável por:

- Possuir uma *interface*, ou várias, de leitura ou escrita digital, leitura ou escrita analógica, e escrita PWM para que possa comunicar-se com sensores, atuadores ou diretamente com dispositivos inteligentes conectados diretamente a ele;
- Rotear, quando requisitado, dados entre outros controladores-escravo e o controlador-mestre;
- Receber comandos do controlador-mestre mesmo que antes passem por outros controladores-escravo, e executar os comandos nas *interfaces* corretas, para assim aplicar a ação no dispositivo correto;
- Monitorar suas *interfaces* com dispositivos e, quando perceber alterações, notificar o controlador-mestre sobre os novos valores lidos.

O objetivo do controlador-escravo é que seja o mais simples possível, consuma o mínimo possível de energia e faça o mínimo de processamento. Sua forma física deve ser de um microcontrolador simples e barato, como a Arduino, ou mesmo circuitos analógicos, quando cabível. Sendo capaz de comunicar-se serialmente, por meio de uma rede ZigBee, monitorar, escrever e ler em suas *interfaces* digitais e/ou analógicas, pode tomar qualquer forma. Essa entidade não deve armazenar nem o estado nem dados de dispositivos conectados a suas *interfaces*, ela deve apenas escrever ou ler dados em suas *interfaces* de acordo com o requisitado pelo controlador-mestre.

Dois conceitos utilizados para essa entidade são o de circuito de escrita e o de circuito de leitura. São considerados circuitos de escrita qualquer circuito que seja conectado a um controlador-escravo com o intuito de alterar um estado de um objeto. Exemplos seriam atuadores como relés, LED infravermelho ou controles de rádio. Circuitos de leitura são qualquer circuito conectado a um controlador-escravo que tenha o intuito de interpretar o estado de um dispositivo. Exemplos seriam sensores de temperatura, luminosidade ou distância. Com esses dois tipos de circuitos, um controlador-escravo é capaz de alterar e gerenciar o estado de dispositivos.

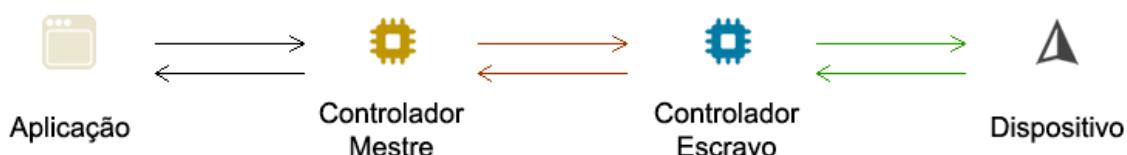


Figura 3.3 - Fluxo simplificado de dados entre entidades físicas

A Figura 3.3 traz um resumo de como as entidades físicas trocam dados entre si. Aplicações trocam dados com controladores-mestre. Controladores-mestre trocam dados com controladores-escravo e aplicações. Controladores-escravo trocam dados com dispositivos e controlador-mestre. E dispositivo troca dados com controlador-escravo.

3.1.2. ABSTRAÇÃO DE DISPOSITIVOS

Para entender como os serviços requisitam informações e como o controlador-mestre transforma essas requisições em comandos a serem executados pelo controlador-escravo, primeiro é preciso entender como a arquitetura faz a abstração dos dispositivos.

Foi utilizado o esquema de abstração do UPnP em que cada dispositivo possui serviços, que possuem ações a serem tomadas baseadas em argumentos enviados junto dela. Quando uma ação acontece, ela altera o estado real do dispositivo, que é traduzido em alterações lógicas nas variáveis de estado do serviço. Quando uma ação é finalizada, podem ser retornados argumentos de saída para o requisitante como o resultado.

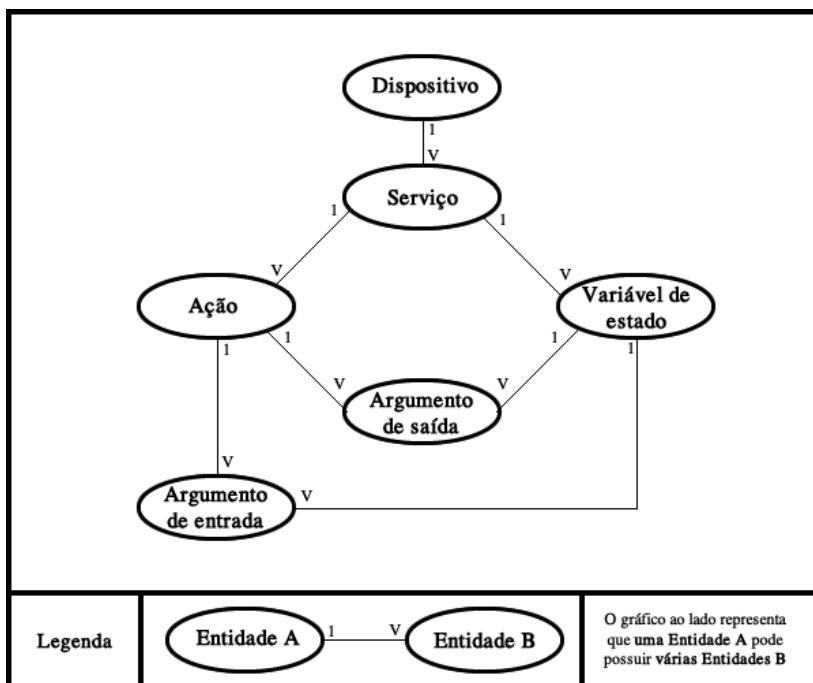


Figura 3.4 - Relacionamento da abstração lógica de dispositivos

Um exemplo dessa abstração seria um televisor conectado a um microcontrolador em uma sala de estar por meio de um LED infravermelho. O controlador-escravo seria o microcontrolador, o dispositivo seria o televisor, um de seus serviços seria o “gerenciar volume”. As ações desse serviço seriam “silenciar” e “ajustar volume”. A ação “ajustar

“volume” teria o argumento de entrada “ajustar volume para”, que se relacionaria com uma variável de estado chamada “volume”, e o argumento de saída “volume ajustado para”, que também se relacionaria com a mesma variável de estado chamada “volume”.

Cada controlador-escravo pode ter vários dispositivos. Cada dispositivo deve ter um ou vários serviços. Cada serviço deve ter uma ou várias ações, e uma ou mais variáveis de estado. Cada ação pode ter vários argumentos de entrada e/ou de saída. Cada argumento de entrada ou saída deve estar relacionado com apenas uma variável de estado. O relacionamento é feito por um identificador numérico único para cada entidade, podendo repetir-se em tipos de entidade distintas. O diagrama de relacionamento está exibido na Figura 3.4.

3.1.2.1 . CONTROLADORES-ESCRAVO

A abstração lógica de um controlador-escravo deve possuir os campos “nome único” e “endereço ZigBee”. O campo “nome único” serve para que aplicações identifiquem unicamente o controlador quando forem requisitar serviços de dispositivos acoplados a ele. O Endereço ZigBee serve para que o controlador-mestre saiba em qual nó da rede ZigBee o controlador-escravo se encontra e assim possa passar os comando para o controlador correto.

3.1.2.2 . DISPOSITIVOS

Para ser compatível com as regras UPnP, a abstração lógica de um dispositivo deve possuir os campos “nome único”, “controlador ao qual está conectado”, “fabricante”, “URL do fabricante”, “descrição do modelo do dispositivo”, “nome do modelo do dispositivo”, “número do modelo do dispositivo”, “URL do modelo do dispositivo”, “número de série” e “Código Universal do Produto (UPC)”. Esses dados são utilizados para gerar o XML de descrição exibido no ANEXO 2 - XML UPnP DE DESCRIÇÃO DE DISPOSITIVOS. Os campos UPnP de dispositivo faltantes devem ser gerados automaticamente pela própria arquitetura no ato de cadastro do dispositivo.

Em especial, o campo “nome único” é utilizado para a API REST. É por meio dele que uma aplicação pode requisitar os serviços de um dispositivo. Esse campo não pode ser

repetido para dispositivos dentro do mesmo controlador-escravo a fim de manter a unicidade hierárquica para as APIs.

3.1.2.3 . SERVIÇOS

Um serviço deve possuir os campos “nome único”, “Descrição” e “dispositivo a que pertence”. Através do nome único, uma aplicação poderá identificar quais ações e variáveis de estado o serviço possui. Esse campo não pode ser repetido para serviços dentro do mesmo dispositivo a fim de manter a unicidade hierárquica para as APIs.

3.1.2.4 . AÇÕES

Ações possuem os campos “nome único” e “serviço a que pertence”. Pelo nome único, aplicações poderão saber quais os argumentos de entrada precisam fornecer e quais argumentos de saída receberão quando utilizarem aquela ação. Esse campo não pode ser repetido para ações dentro do mesmo serviço a fim de manter a unicidade hierárquica para as APIs.

3.1.2.5 . VARIÁVEIS DE ESTADO

Variáveis de estado possuem os campos “nome único”, “serviço ao qual pertence”, “tipo do dado”, “enviar eventos”, “*multicast*”, “pino do circuito de leitura”, “forma de operação do pino do circuito de leitura”, “taxa de transmissão do circuito de leitura”, “pino do circuito de escrita”, “forma de operação do pino do circuito de escrita” e “taxa de transmissão do circuito de escrita”.

O campo “tipo do dado” diz se os valores que a variável de estado pode assumir são números, booleanos ou textuais. O campo “enviar eventos” diz se, ao trocar de estado, a arquitetura deve notificar assinantes desse serviço da alteração nessa variável de estado. O campo “*multicast*” diz se as mensagens de eventos para essa variável de estado serão enviadas para todos os assinantes, por meio de *multicast*, UDP, ou *unicast*, TCP. Os campos “pino do circuito” de escrita e leitura dizem, respectivamente, em que pino está o circuito para alteração ou monitoramento do estado do dispositivo equivalente à variável de estado. Os campos “forma de operação do pino” para circuitos de escrita e leitura dizem se no “pino do circuito” deve-se, respectivamente, escrever e ler o estado do dispositivo de

forma digital ou analógica. O campo “nome único” não pode ser repetido para variáveis de estado dentro do mesmo serviço a fim de manter a unicidade hierárquica para as APIs.

Cada variável e estado pode possuir uma gama de valores permitidos, assim como manda o UPnP, e, para melhorar a inteligibilidade, cada valor enviado por uma aplicação pode possuir um valor de tradução para ser enviado para a camada de execução. Um exemplo disso seria permitir que uma variável de estado chamada *energia* só pudesse ter os valores ligado e desligado e que deveriam ser traduzidos, respectivamente, para 1 e 0, na hora de passar para a camada de execução.

3.1.2.6 . ARGUMENTOS DE ENTRADA E DE SAÍDA

Os argumentos de uma ação, sejam eles de entrada ou saída, devem conter os seguintes campos: “nome único”, “ação a que pertence” e “variável de estado relacionada”. A arquitetura utiliza a variável de estado relacionada a um argumento de entrada recebido de uma aplicação para aplicar no circuito de escrita dessa variável o valor recebido nesse argumento. No caso do argumento de saída, é enviado o último valor, lido no circuito de leitura da variável de estado relacionada, na resposta de um pedido de ação realizado com êxito para uma aplicação.

3.1.3. ESTRUTURA LÓGICA

Outra forma de observar a arquitetura proposta é por seus componentes lógicos, que residem em suas partes físicas e utilizam a abstração do dispositivo para prestar serviços adequadamente.

Baseada na arquitetura de *middleware* orientada a serviços proposta em Atzori *et al.*, (2010), a estrutura lógica aqui proposta foi separada em quatro camadas que funcionam independentes umas das outras. São elas: camada de serviço, camada de controle, camada de comunicação e camada de execução.

A camada de serviço é responsável por trocar dados com as aplicações. A camada de controle é responsável por fazer o gerenciamento de todos os dispositivos e suas abstrações e por traduzir requisições da camada de serviço nos comandos certos para os controladores-escravo executarem. A camada de comunicação é responsável por levar os

comandos do controlador-mestre até o controlador-escravo correto e vice-versa. A camada de execução é responsável por executar ações e ler estados de dispositivos.

O controlador-mestre trabalha nas camadas de serviço, controle e comunicação; o controlador-escravo, nas de comunicação e execução.

A Figura 3.5 - Arquitetura em camadas, traz uma representação dos objetos físicos, com as camadas que eles utilizam e com as entidades lógicas dentro de cada camada que eles utilizam e como elas se relacionam.

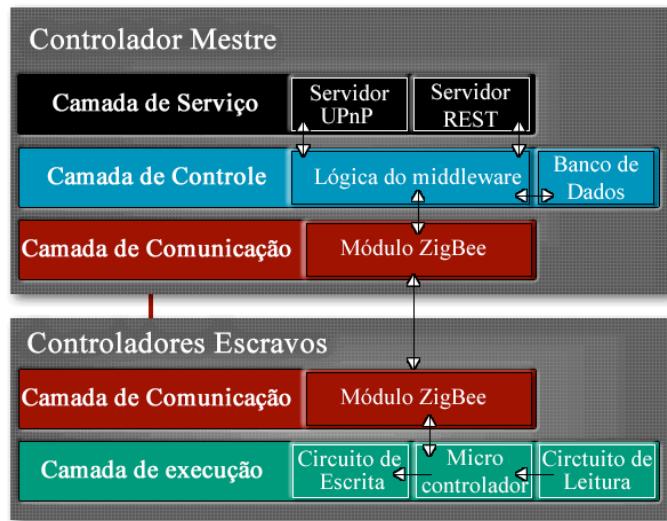


Figura 3.5 - Arquitetura em camadas

3.1.3.1 . CAMADA DE SERVIÇOS

A camada de serviços é responsável por trocar informações diretamente com aplicações, via APIs. Sua função é receber as requisições, identificar o que foi pedido e repassar para a camada de controle o pedido de forma uniforme, independente de qual API a requisição veio. No modelo aqui proposto, serão abordadas apenas APIs UPnP e REST. Caso futuramente novas APIs sejam adicionadas, devem seguir o mesmo procedimento adotado pelas API REST e UPnP.

3.1.3.1.1 – INTERFACE UPnP

A interface UPnP deve ser escrita utilizando um *kit* de desenvolvimento de *software*, do inglês *Software Development Kit* (SDK), portátil recomendado pelo Fórum UPnP. Com isso, é esperado que a arquitetura se beneficie diretamente de evoluções e documentação

desses SDKs. O funcionamento deve estar de acordo com as recomendações do Fórum para garantir sua interoperabilidade total.

A Camada de controle gera e mantém atualizados os XML de todos os dispositivos e serviços. Caso a interface UPnP receba requisições por eles, basta repassá-las. É de responsabilidade da interface UPnP gerenciar os assinantes de serviços, notificá-los de alterações sempre que a camada de controle pedir e atender a requisições de controle nas URLs especificadas pelos XML de descrição.

Caso a interface UPnP receba uma requisição por ação SOAP, utilizando o DCP UPnP, ela deve repassar um JSON, contendo o identificador único do dispositivo, o identificador único do serviço, o nome único da ação e os nomes únicos e os valores dos argumentos de entrada. Esses dados devem ser passados por meio de um *socket* para a camada de controle. O JSON tem seus campos como mostrado na Tabela 4.

Tabela 4 - Campos do JSON de pedido de ação para camada de controle

Nome do campo	Tipo do campo	Item pai	Observações
Dispositivo	Número	raiz	-
Serviço	Número	raiz	-
Ação	Texto	raiz	-
Argumentos	Objeto	raiz	-
Argumento de entrada	Número, texto ou booleano	Argumentos	Podem existir vários no mesmo JSON

A camada de controle pode enviar JSONs para notificar a chegada de um novo dispositivo na rede, para notificar a saída de um dispositivo da rede, ou para notificar a alteração de estado de alguma variável.

No caso de um novo dispositivo se conectar ou desconectar da rede, a camada de controle deve enviar uma mensagem para a camada de serviços, contendo o identificador único do dispositivo e o *status* dele que pode ser *online* ou *offline*. Os campos do JSON dessa mensagem seguem o padrão da Tabela 5.

Tabela 5 - Campos do JSON de notificação de disponibilidade, ou não, de um dispositivo

Nome do campo	Tipo do campo	Item pai
Dispositivo	Número	raiz
Status	Texto	raiz

Ao receber tais mensagens, a camada de serviços deve enviar mensagens SSDP, como manda o UPnP, notificando da disponibilidade, ou não, do dispositivo.

No caso da alteração de uma variável de estado, a camada de controle irá enviar uma mensagem contendo o identificador único do dispositivo e do serviço, um *status* da mensagem como “*change*”, o identificador da variável e estado e o novo valor que ela assumiu. O JSON dessa mensagem deve seguir os campos especificados na Tabela 6.

Ao receber tal mensagem, a camada de serviços deve notificar a todos os assinantes desse serviço do novo valor, via GENA.

Todos os JSONs supracitados estão exibidos no ANEXO 4 – JSONS TROCADOS ENTRE AS CAMADAS DE SERVIÇOS E DE CONTROLE.

Tabela 6 - Campos do JSON de notificação de alteração de variável de estado

Nome do campo	Tipo do campo	Item pai
Dispositivo	Número	raiz
Status	Texto	raiz
Serviço	Número	raiz
Variável de estado	Objeto	raiz
Nome	Texto	Variável de estado
Valor	Texto, Número ou Booleano	Variável de estado

3.1.3.1.2 . API REST

No caso da API REST, ela apenas comunicará a camada de controle sobre requisições de ação que ocorrem por requisições do tipo PUT. As demais notificações não serão recebidas, pois seus *sockets* só devem perdurar enquanto uma requisição de controle não for finalizada. Para controle, a API REST deve seguir a mesma regra imposta para a Interface UPnP e deve obedecer aos campos da Tabela 4.

Diferentemente de uma API UPnP, os campos de uma API REST não são definidos, apenas é proposta a forma como uma requisição deve acontecer. Por tal motivo, serão explicitados os campos da API REST para que possa garantir a interoperabilidade do sistema. A

traz todos as requisições GET suportadas pela API REST e o que essa ação deve retornar. Essas requisições são para que aplicações REST sejam capazes de descobrir todas as entidades disponíveis no sistema bem como suas características. Os JSONs Array referidos

na tabela são apenas uma estrutura *array*, como mostrado na Tabela 1 - Tipos JSON, contendo o identificador de cada entidade em questão. O JSON Objeto citado contém como seus campos o que foi especificado na seção 3.1.2.

Para retornar os dados para os GETs, reduzir a complexidade e o processamento e para possibilitar URLs mais complexas no futuro, a API REST deve buscar por esses dados diretamente no banco de dados, sem necessitar pedir os dados ou autorização da camada de controle.

Tabela 7 - Requisições GET suportadas pela API REST

URL do recurso	Ação esperada
/controladores_escravos/	JSON Array com ID de controladores-escravo existentes
/controladores_escravos/id_controlador/	JSON Objeto com os dados do controlador
/controladores_escravos/id_controlador/dispositivos/	JSON Array com ID de dispositivos associados ao controlador-escravo
/dispositivos/	JSON Array com ID de dispositivos existentes
/dispositivos/id_dispositivo	JSON Objeto com os dados do dispositivo
/dispositivos/id_dispositivo/serviços/	JSON Array com ID de serviços associados ao dispositivo
/serviços/	JSON Array com ID de serviços existentes
/serviços/id_serviço/	JSON Objeto com os dados do serviço
/serviços/id_serviço/ações/	JSON Array com ID de ações associadas ao serviços
/serviços/id_serviço/variáveis_de_estado/	JSON Array com ID de variáveis de estado associadas ao serviços
/ações/	JSON Array com ID de ações existentes
/ações/id_ação/	JSON Objeto com os dados da ação e de seus argumentos
/variáveis_de_estado/	JSON Array com ID de variáveis de estado existentes
/variáveis_de_estado/id_variavel_de_estado	JSON Objeto com os dados da variável de estado

A maneira de controlar dispositivos é com requisições do tipo PUT. Uma vez que a aplicação saiba o identificador único da ação que gostaria de invocar ou do nome único de controlador-escravo, dispositivo, serviço e ação, ele poderá acessar as URLs de controle. Além da URL, ele deve informar, via parâmetro, os argumentos de entradas daquela ação. A Tabela 8 traz as URLs e a forma de parâmetros possíveis, por PUT, que permitem o

controle de dispositivos. O retorno dessas requisições é um JSON, contendo um *status*, que pode assumir as *strings* “feito” ou “erro”, uma mensagem, contendo um texto explicando o que aconteceu, e, caso o *status* seja “feito”, um objeto JSON, contendo os argumentos de saída com nome e valor.

Tabela 8 - Requisições PUT suportadas pela API REST

URL do recurso	Parâmetros
/ações/id_ação	?argumento_de_entrada_1=valor_argumento_1 &argumento_de_entrada_2=valor_argumento_2 &argumento_de_entrada_3=valor_argumento_3 ... &argumento_de_entrada_n=valor_argumento_n
/nome_controlador/nome_dispositivo/ nome_serviço/nome_ação	

A URL base para acesso dessa API deve ser o IP ou domínio do controlador-mestre seguido de “api_rest”, seguido da versão da API, como mostra o exemplo abaixo:

http://192.168.0.2/api_rest/v1/

Depois da URL base, é escrita a URL do recurso e, por último, no caso do PUT, os parâmetros do recurso.

3.1.3.2 . CAMADA DE CONTROLE

Como explicitado pela camada de serviços, a camada de controle é responsável por prover um *socket* de comunicação para a camada superior a fim de atender a requisições de controle de dispositivo, de notificar alteração de disponibilidade de dispositivo na rede e de notificar de alterações em variáveis de estado. Todas essas ações são tomadas por meio de JSONs, já definidos na seção 3.1.3.1.

Além de trabalhar com a camada de serviços, a camada de controle é responsável por administrar a abstração lógica dos dispositivos e prover meios para que usuários administradores também o façam, por enviar requisições de controle para controladores-escravo por meio da camada de comunicação e por receber dela notificações de alteração de estado de dispositivos.

Para adicionar, editar e remover dispositivos ou controladores-escravo do *middleware*, a camada de controle provê uma *interface web* para que a pessoa que configurou o *hardware*, denominado *administrador*, cadastre sua abstração lógica e a gerencie

posteriormente. A configuração deve acontecer através de formulários *web* padrão que devem ser acessados, via IP ou domínio do controlador-mestre, seguido de “admin” como mostra o exemplo abaixo: <http://192.168.0.2/admin/>

Nesse endereço, o gerenciador da arquitetura poderá visualizar, cadastrar, editar ou excluir controladores-escravo, dispositivos, serviços, ações, variáveis de estado, argumentos de entrada ou de saída relativos ao *hardware* que ele gerencia. Uma vez que o administrador remova ou adicione novo dispositivo, a camada de controle deve notificar a camada de serviços sobre o novo *status*, *via* um JSON, já explicitado na

Tabela 5 - Campos do JSON de notificação de disponibilidade, ou não, de um dispositivo. Os campos dos formulários para o gerenciamento dessas entidades deve seguir os padrões expostos na seção 3.1.2, para que assim a camada de controle possa utilizá-lo e permitir que as demais camadas também o façam.

Uma vez que um dispositivo esteja corretamente configurado, com sua abstração lógica inserida com êxito, a camada de controle estará apta a controlar e gerenciar o *hardware* e sua abstração quando receber mensagens da camada de serviço e da camada de execução, respectivamente.

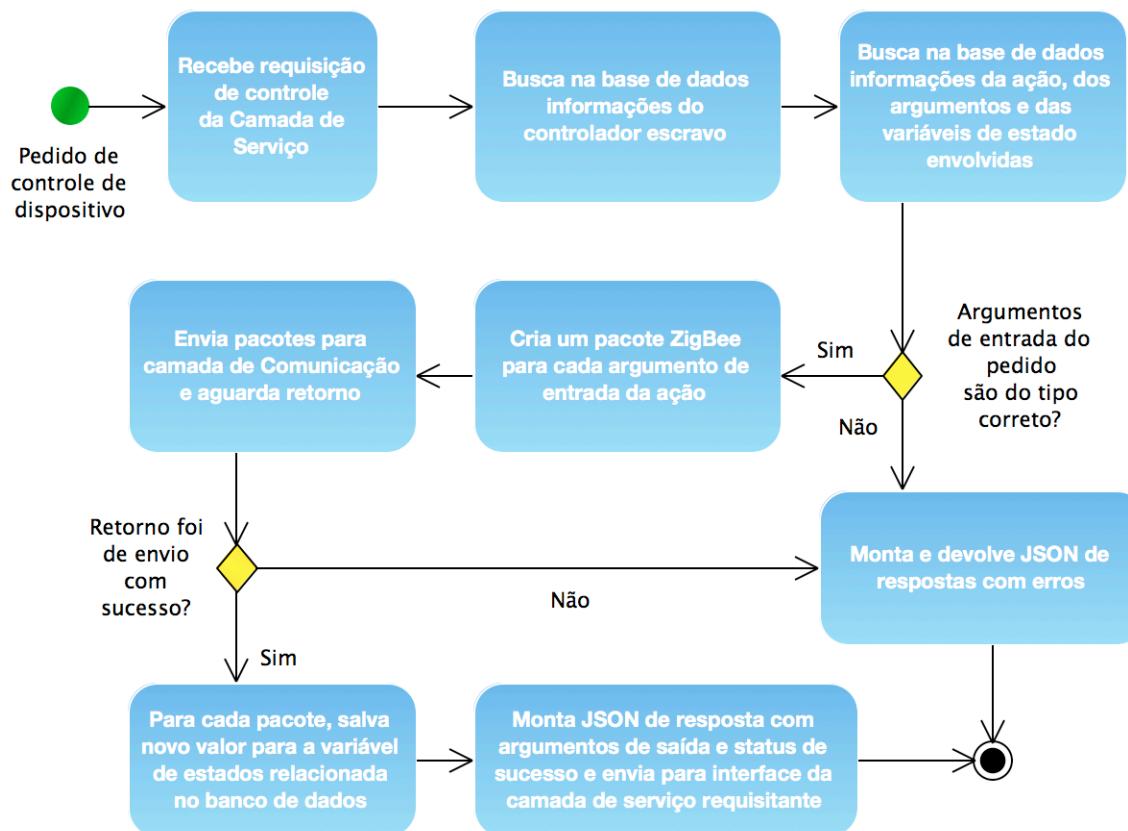


Figura 3.6 - Procedimento da camada de controle para alterar estado de um dispositivo

Ao receber uma requisição da camada de serviços para controle de um dispositivo, a camada de controle utiliza o JSON para buscar em seu banco de dados as informações da ação e do controlador-escravo. Do controlador-escravo, é utilizado o campo “Endereço ZigBee”. Da ação são pegos os dados de pinos de escrita de cada argumento. Se os valores recebidos no JSON forem dos tipos corretos para o pino, a camada de controle gera, para cada valor de argumento de entrada, um JSON contendo o “Endereço ZigBee” do controlador de destino, o pino em que a informação deve ser escrita nesse controlador, a forma de operação do pino (digital ou analógico), o tipo do dado a ser escrito (inteiro, texto, booleano), a taxa a se transmitir os *bytes* da informação no pino e, por fim, os dados a serem escritos no pino. Após gerar os JSONs de pedido, a camada de controle os passa, um a um, para a camada de comunicação e aguarda todos JSONs de resposta. Quando recebidas e se houver falhas, a camada de controle notifica a camada de serviço da mensagem de erro em conjunto com *status* de erro. Caso não haja erros, a camada de controle notifica que a ação foi requisitada com sucesso e retorna os argumentos de saída junto com *status* de sucesso. Esse procedimento está demostrado na Figura 3.6. e os JSONs estão especificados no ANEXO 5 – JSONS TROCADOS ENTRE AS CAMADAS DE COMUNICAÇÃO E DE CONTROLE. Uma vez que perceba a alteração de estado de um dos pinos do controlador-escravo em que está instalada, a camada de execução envia, por camada de comunicação, uma mensagem à camada de controle. Ao receber essa mensagem, contendo um objeto JSON, também especificado no ANEXO 5 – JSONS TROCADOS ENTRE AS CAMADAS DE COMUNICAÇÃO E DE CONTROLE, a camada de controle busca, em seu banco de dados, quais variáveis de estado estão vinculadas a aquele pino do controlador-escravo, altera seus valores e as salva novamente. Em seguida, ela notifica todas *interfaces* da camada de serviços da alteração em cada pino, por meio do JSON, especificado na seção 3.1.3.1.1. Um resumo desse procedimento está demonstrado na Figura 3.7.

A mensagem a ser recebida da camada de comunicação contém os campos endereço ZigBee do controlador-mestre, pino em que foi detectada a alteração, momento de detecção do novo valor. Uma vez que o controlador-escravo não é obrigado a possuir um relógio de tempo real, o campo do momento da detecção deve ser preenchido com números um maior que o outro, de acordo com o acontecimento das alterações. Esse campo é especificado na seção relativa à camada de comunicação e é repassado à camada de controle por meio de um JSON, demostrado no ANEXO 5 – JSONS TROCADOS ENTRE

AS CAMADAS DE COMUNICAÇÃO E DE CONTROLE, deve ser utilizado pela camada de controle para saber se, de fato, a última alteração recebida foi a última enviada pela camada de execução, haja visto que as mensagens podem tomar caminhos diferentes na rede em malha e chegar em momentos distintos e, talvez, com ordem invertida.

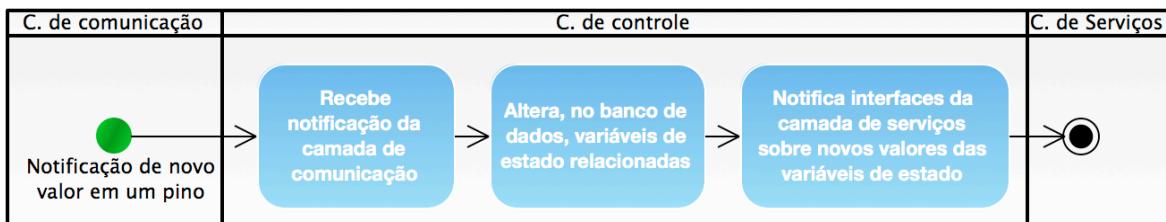


Figura 3.7 - Procedimento da camada de controle para notificações de alteração de estado

Caso a camada de comunicação não consiga, por qualquer motivo, repassar uma alteração de estado para a camada de controle, ela salvará esses dados no banco de dados para que, quando a camada de controle se reestabelecer, possa saber das alterações e seguir o fluxo da Figura 3.7.

3.1.3.3 . CAMADA DE COMUNICAÇÃO

A camada de comunicação é responsável por levar mensagens da camada de controle até a camada de execução, e vice-versa. Sua missão é fazer com que a comunicação seja a menos onerosa possível para a rede em malha, que possui baixas taxas de transmissão, e para seus nós, que – em sua maioria – possuem limitações de processamento e de consumo de energia.

Métodos de compressão e codificação de canal para atingir os melhores resultados possíveis na transmissão estão fora do escopo deste trabalho. Foi escolhido o ZigBee para cuidar do roteamento dos pacotes entre o nó de origem e o de destino na rede em malha, para tornar a solução aqui proposta padronizada e para poder se beneficiar diretamente de qualquer avanço tecnológico no padrão, que é mantido por entidades de peso no mundo acadêmico e no mundo empresarial, como descrito na Seção 2, deste trabalho.

Existem quatro tipos de mensagens que trafegam na camada de comunicação. São elas: pedido de escrita de valor em pino, confirmação de escrita de valor em pino, notificação de novo de valor lido em pino, confirmação de recebimento da notificação de novo lido em valor de pino.

Tabela 9 - Campos da informação carregada por pacotes ZigBee de pedido de escrita de valor em pino

Campo	Conteúdo	Tamanho	Valores permitidos
0	Cabeçalho ZigBee	24 bytes	Não se aplica
1	Byte de início	1 byte	0xFF
2	Identificador da mensagem	2 bytes	De 0x00 0x00 a 0xFF a 0xFF
3	Tamanho dos dados	Ilimitado	Cadeia de bytes terminada com o byte 0x00
4	Número do pino	1 byte	de 0x00 a 0xFF
5	Forma de operação do pino	2 bits	00 – Digital 01 – Analógico 10 – PWM 11 - UART
6	Tipo do dado	2 bits	00 – Inteiro 01 – Booleano 10 – Texto ASCII 11 – Longo
7	Taxa de transmissão	4 bits	0000 – 0 0001 – 75 0010 – 150 0011 – 300 0100 – 600 0101 – 1200 0110 – 2400 0111 – 4800 1000 – 9600 1001 – 14400 1010 – 16457 1011 – 19200 1100 – 28800 1101 – 38400 1110 – 57600 1111 – 11520
8	Dados a serem escritos no pino do campo 3	ilimitado	Cadeia de bytes com o tamanho especificado no campo 3

O pedido de escrita de valor em pino é realizado da camada de controle para a camada de execução. O objeto JSON recebido da camada de controle é convertido em um pacote de dados com destino ao endereço ZigBee recebido, contendo o tamanho do pacote, o pino em

que a informação deve ser escrita, a forma de operação do pino (digital ou analógico), o tipo do dado a ser escrito, a taxa de transmissão dos *byte* da informação no pino e, por fim, os dados a serem escritos. Foi decidido utilizar um padrão próprio para troca de dados dentro da rede ZigBee para reduzir ao máximo a quantidade de informação a ser trocada dentro dessa rede, economizando recursos e aumentando a eficiência. Por isso, foram retirados quaisquer dados desnecessários, tirando a prioridade do fácil entendimento do pacote e priorizando a eficiência e a economia de recursos. O pacote final formado está especificado na Tabela 9 - Campos da informação carregada por pacotes ZigBee de pedido de escrita de valor em pino.

O primeiro campo é composto pelo *byte* 0xFF. É utilizado para que o código receptor saiba que começou um novo pacote pois o canal é sempre mantido em 0x00 quando não está em transmissão. Os próximos 3 *bytes* contém o identificador da mensagem que deve ser enviado de volta na resposta de falha ou de sucesso de recebimento. Os próximos *bytes* contêm o tamanho, em *bytes*, dos dados da mensagem carregada no campo 8. Aos *bytes* de tamanho da mensagem é adicionado um último *byte* com valor 0x00 para notificar seu próprio fim. Em seguida, vem um *byte* para denotar o número do pino em que a informação deve ser escrita. Em seguida, vem um *byte* utilizado para notificar como a informação deve ser escrita no pino. É recomendado que seus dois primeiros *bits* sejam para notificar a forma como o pino deve operar para escrever os dados, os dois próximos *bits* para notificar o tipo do dado definindo, o tamanho do símbolo e os quatro restantes para a taxa de transmissão de símbolos a ser feita no pino. Em seguida, vêm os *bytes* da mensagem a ser escrita no pino especificado. Seu tamanho é especificado no campo 3 do pacote.

Por padrão, uma mensagem ZigBee tem que ser dividida quando passa de 104 *bytes* para otimizar o tráfego da rede. A camada de controle é responsável por dividir a mensagem, que para o protocolo ZigBee compreende do campo 1 ao 8 da tabela, em porções de até 104 *bytes* no emissor e depois remontar no receptor e entregar tudo de uma vez para a camada de destino. Quando houver a necessidade de divisão de pacote, o pacote primeiro carregará os campos de 0 até onde couber em 100 *bytes*. Os demais pacotes vêm com o novo campo 0, campos 1 e 2 repetidos e os demais campos, ou pedaços de campos, ainda não transmitidos que caibam no pacote ZigBee.

Ao concluir a transmissão da mensagem completa com sucesso para a camada de execução do controlador-escravo de destino, a camada de comunicação repassa o pacote para a camada de execução que checa sua integridade e congruência. Após receber e validar, a mensagem, a camada de execução deve responder a camada de controle com uma mensagem de confirmação de recebimento. Caso esteja tudo correto com o pedido de escrita de valor em pino, a camada de execução gera uma mensagem de sucesso e, caso exista algum erro, retorna o código do erro. A camada de comunicação monta um pacote contendo o mesmo identificador do campo 2 da mensagem recebida e o envia para a camada de comunicação do controlador-mestre. Os campos desse pacote estão especificados na Tabela 10.

Tabela 10 - Campos da informação carregada por pacotes ZigBee de mensagem de confirmação de recebimento

Campo	Conteúdo	Tamanho	Valores permitidos
0	Cabeçalho ZigBee	24 bytes	Não se aplica
1	Byte de início	1 byte	0xFF
2	Identificador da mensagem	2 bytes	De 0x00 0x00 a 0xFF a 0xFF
3	Status	1 byte	0x00 – Sucesso 0x01 – Pino inválido 0x02 – Pino não suporta a forma de operação 0x03 – Pino não suporta a taxa de transmissão 0x04 – Pino não suporta o tipo de dado 0xFF – Erro desconhecido

Em posse desses dados, a camada de comunicação monta um JSON, exibido no ANEXO 5 – JSONS TROCADOS ENTRE AS CAMADAS DE COMUNICAÇÃO E DE CONTROLE, contendo o código da resposta como, “sucesso” ou “erro”, e adiciona o significado do campo 3, *status*, como mensagem para a camada de controle, assim finalizando um pedido de alteração de valor de pino. Em caso de erro, não deve haver a retransmissão do pedido pois essa decisão ficará a cargo da aplicação que utiliza o *middleware*, pela camada de serviços.

Outra operação que a camada de comunicação dá suporte é a de notificação de alteração de estado de pino. Uma vez que a camada de execução detecte a alteração do valor de um pino, ela pede à camada de comunicação que transporte o valor lido e o número do pino para a camada de controle. A camada de comunicação do controlador-escravo então monta um pacote contendo um *byte* de início, identificador da mensagem, tamanho do valor lido, número do pino no qual o valor foi lido, e o valor lido. Os campos desse pacote estão especificados na Tabela 11.

Tabela 11 - Campos da informação carregada por pacotes ZigBee de notificação de novo valor lido em pino

Campo	Conteúdo	Tamanho	Valores permitidos
0	Cabeçalho ZigBee	24 bytes	Não se aplica
1	Byte de início	1 byte	0xAA
2	Identificador da mensagem	2 bytes	De 0x00 0x00 a 0xFF a 0xFF
3	Tamanho dos dados	ilimitado	Cadeia de bytes terminada com o byte 0x00
4	Número do pino	1 Byte	De 0x00 a 0xFF
5	Dados lidos no pino do campo 3	ilimitado	Cadeia de bytes com o tamanho especificado no campo 3

A camada de comunicação do controlador-mestre recebe o pacote ZigBee e monta um JSON, demonstrado no ANEXO 5 – JSONS TROCADOS ENTRE AS CAMADAS DE COMUNICAÇÃO E DE CONTROLE, e envia para a camada de controle. Em seguida, a camada de controle do controlador-mestre envia um pacote de confirmação de recebimento para a camada de comunicação do controlador-escravo, nos mesmos moldes da confirmação de pedido de escrita de valor em pino, que está especificado na Tabela 10 - Campos da informação carregada por pacotes ZigBee de mensagem de confirmação de recebimento. O valor do campo *status* assume somente “sucesso” ou “erro desconhecido”, para quando houver algum erro genérico na aplicação. Em caso de erro, para sanar o caso sobrecarga, a camada de comunicação do controlador-mestre deve aguardar um tempo aleatório de até um minuto e tentar enviar mais uma vez o JSON para a camada de controle. Caso não consiga, mesmo assim, deve salvar o pino, momento de recebimento,

valor recebido e controlador-escravo recebido no banco de dados para que a camada de controle possa posteriormente recuperar os dados e fazer as devidas notificações.

Caso o pacote de notificação de novo valor lido em pino exceda 104 *bytes*, ela será separado pela camada de comunicação da mesma forma que é feito para a mensagem de pedido de escrita de valor em pino.

Para as mensagens de notificação de novo valor lido em pino e de pedido de escrita de valor em pino, o valor do campo “identificador da mensagem” deve ser zero e, para cada nova mensagem deve incrementar consecutivamente de um em um até chegar no valor máximo, quando retorna a zero. Mensagens fragmentadas ou confirmações de recebimento devem utilizar o mesmo “identificador da mensagem” que o primeiro pacote ZigBee ou da mensagem para a qual se está notificando o recebimento.

3.1.3.4 . CAMADA DE EXECUÇÃO

Abaixo da camada de comunicação se encontra a camada de execução. Como já dito nas camadas superiores, sua função é de escrever valores em pinos e ler valores em pinos. Com estas simples funcionalidades ela permite à camada de controle alterar estado de um dispositivo e de saber em qual estado um dispositivo está. O objetivo de existência dessa camada é para que ela seja a interface entre o mundo real e o mundo virtual para um dispositivo. Ela serve de ponte para que a abstração lógica da camada de controle de fato seja verdadeira para o dispositivo em questão.

Como um dos intuios da arquitetura aqui proposta é ser extensível a qualquer dispositivo, é preciso englobar desde o caso de dispositivos finais sem poder computacional, como uma lâmpada comum, ao caso de um dispositivo poderoso, como um computador. Para isso, a camada de execução utiliza-se de comunicação via sinais digitais e analógicos de circuitos de escrita e leitura para trazer objetos inertes ao mundo real e para repassar dados a objetos já inteligentes.

Suas principais funcionalidades de operação são receber, da camada de controle, comandos para escrever valores em suas interfaces digitais e analógicas e monitorar suas interfaces

para, sempre que detectar alterações, notificar qual o valor lido e em qual pino ocorreu a alteração para a camada de controle.

O comando para escrita de valores em uma *interface* vem dos pacotes ZigBee da camada de controle, com campos especificados na Tabela 9 - Campos da informação carregada por pacotes ZigBee de pedido de escrita de valor em pino, que é reconvertido no mesmo objeto JSON que era quando saiu da camada de controle, exibido no ANEXO 5 – JSONS TROCADOS ENTRE AS CAMADAS DE COMUNICAÇÃO E DE CONTROLE, que é repassado da camada de comunicação para a camada de execução. Após recebê-lo, checa se possui a *interface* pedida pelo campo “número do pino”. Se possuir, checa se a forma de escrita dos dados proposta nos campos “forma de operação”, “taxa de transmissão” e “tipo de dado” são suportados. Caso esteja tudo coerente com os dados da mensagem, ele deve escrever os dados como proposto na *interface* e retornar a mensagem com código de *status* 0x00, representando sucesso. Caso contrário, deve retornar um código de *status* entre 0X01 e 0XFF, indicando erro. É importante frisar que o *byte* que contém “forma de operação”, “taxa de transmissão” e “tipo de dado” pode ser alterado para outros significados, haja vista que são transmitidos como um byte independentemente de seu significado *bit a bit*. Foram escolhidos os valores padrão da Tabela 10 - Campos da informação carregada por pacotes ZigBee de mensagem de confirmação de recebimento. porque são os que funcionam melhor com os microcontroladores expostos na seção 2 deste trabalho.

A segunda funcionalidade é a de monitorar sua *interfaces* e, ao reparar que ocorreu alteração, ler o valor completo e repassar para a camada de controle através da camada de comunicação. Para isso, basta criar um JSON contendo o número do pino e o novo valor e repassar para a camada de comunicação. O JSON é o mesmo que é repassado pela camada de comunicação para a camada de controle porém sem o atributo “*endereço_zigbee*”.

Para auxiliar na comunicação com dispositivos finais existem os circuitos de escrita e de leitura. Muitos atuadores e sensores são encontrados no mercado, a baixíssimo custo, que trabalham com essa função.

De forma simplificada, dispositivos controláveis por botões, por infravermelho, do inglês *Infrared* (IR), ou por radiofrequência, do inglês *Radio Frequency* (RF), poderiam ser

controlados por circuitos como os da Figura 3.8 - Exemplo de circuito de escrita. Para produzir o circuito de radiofrequência, bastaria trocar o LED IR por um modulador acoplado a uma antena. E para monitorar o estado final de um dispositivo, bastaria utilizar sensores como de luminosidade, tensão, distância e temperatura.

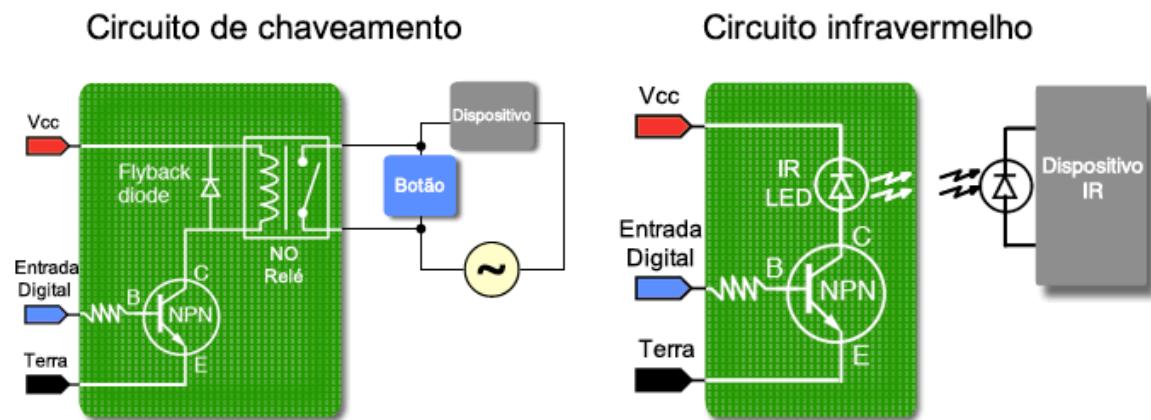


Figura 3.8 - Exemplo de circuito de escrita

3.1.4. EXEMPLOS DE SISTEMAS

Esta seção traz exemplos de abstrações lógicas de dispositivos, com a aplicação dessas abstrações em seis sistemas com propósitos bem definidos e para mostrar quais circuitos de leitura e escrita seriam necessários para implantar fisicamente tais sistemas bem como qual seria sua abstração lógica.

Em cada tabela de sistema, embaixo do circuito de escrita e leitura se encontra o número do *hardware* especializado para a função. O valor da grande maioria desses circuitos não passa de três dólares americanos e todos são compatíveis com a arquitetura proposta por este trabalho.

3.1.4.1 . MONITOR DE HUMIDADE

O primeiro serviço apresentado é o de um monitor de humidade. Primeiramente, deve-se conectar o circuito de leitura no *hardware* do controlador-escravo. Depois, configurar o dispositivo no controlador-mestre de acordo com os dados da Tabela 12. Uma vez configurado, qualquer aplicação REST que, a cada 30 segundos, fizesse uma requisição e recebesse o valor da humidade medida pelo sensor poderia gerar um gráfico autoatualizável com os valores recebidos da camada de serviços da arquitetura. Outra

possibilidade seria uma aplicação UPnP, que receberia o valor sempre que a humidade fosse alterada, podendo exibir para o usuário a humidade em tempo real. Este é um exemplo de como utilizar ambas as APIs da arquitetura proposta para ler estado de dispositivos.

Tabela 12 - Abstração do sistema monitor de humidade

Nome do dispositivo	Serviço	Ação	Variável de estado	Valores permitidos para a variável de estado	Círculo de leitura (análogo)	Círculo de escrita
Higrômetro	monitor	ler_humidade	humidade	De 0 a 1023	Sensor de humidade (YL-38)	--

3.1.4.2 . CONTROLE DE ILUMINAÇÃO

O segundo sistema a ser apresentado é o controlador de iluminação que funciona controlando a potência que é entregue às lâmpadas. Uma vez conectado e configurado, permite que qualquer aplicação saiba o estado de uma lâmpada, em tempo real, e a controle quando bem entender. Se conectada a *interfaces* PWM e trocando o circuito de escrita por outro um pouco mais complexo, é possível realizar o gerenciamento da intensidade de luminosidade a ser liberada pela lâmpada, ao invés de apenas “ligada” ou “desligada”. A abstração desse sistema está representada na Tabela 13. Este é um exemplo de como utilizar a arquitetura proposta para controlar um dispositivo.

Tabela 13 - Abstração do sistema de iluminação

Nome do dispositivo	Serviço	Ação	Variável de estado	Valores permitidos para a variável de estado	Círculo de leitura	Círculo de escrita
Lâmpada	energia	ajustar_energia	energia	ligado, desligado	Sensor de Luminosidade (i0526018)	Círculo de chaveamento

3.1.4.3 . DETECÇÃO PARA INCÊNDIOS

O sistema de detecção de incêndio é um pouco mais complexo pois envolve três partes. Basicamente, quando a camada de execução notar alteração no pino do sensor de fumaça, notificará a camada de controle, que saberá tratar-se de fumaça. A camada de controle pedirá para que a camada de serviço notifique todos os assinantes da alteração da variável fumaça. A aplicação de detecção de incêndio irá enviar dois pedido de ação para a camada de serviços, sendo o primeiro para o serviço sirene, para ajustar o som para *ligado*; e o segundo, para o serviço irrigação, para ajustar a energia para *ligado*. Com isso, ao detectar a fumaça, será acionado o alarme para notificar as pessoas e a água para reduzir o fogo.

Uma vez que a fumaça cesse, a aplicação receberá o *broadcast* do novo valor e fará o pedido para *desligar* a sirene e o esguicho de água. A abstração desse sistema está representada na Tabela 14. Este é um exemplo de como utilizar a arquitetura proposta em um ambiente fechado para entender o contexto e reagir de acordo com a situação.

Tabela 14 - Abstração do sistema para incêndios

Nome do dispositivo	Serviço	Ação	Variável de estado	Valores permitidos para a variável de estado	Círcuito de leitura	Círcuito de escrita
Detector de fumaça	monitor	existe_fumaca	fumaça	verdadeiro, falso	Sensor de fumaça (MN-EB-MQ2GS)	--
Alarme	sirene	ajustar	som	ligado, desligado	--	Sirene (YL-44)
Esguicho de água	Irrigação	ajustar_energia	energia	ligado, desligado	Sensor de voltagem (LM393-3144)	Círcuito de chaveamento

3.1.4.4 . IRRIGAÇÃO

O sistema de irrigação utiliza um higrômetro de solo para monitorar a humidade do solo. Sempre que o sensor muda de valor, uma mensagem GENA é disparada. O sistema de irrigação faz o monitoramento e, quando detectar que o valor ultrapassou um limiar, ele envia um pedido de ação ligando o esguicho de água. Quando receber uma mensagem

notificando que o valor da humidade ultrapassou um outro limiar, ele envia o comando para desligar o esguicho. A abstração desse serviço encontra-se na Tabela 15. Este é um exemplo de como utilizar a arquitetura proposta para auxiliar em tarefas da agricultura através do entendimento do contexto e tomada de decisão automática.

Tabela 15 - Abstração lógica do sistema de irrigação

Nome do dispositivo	Serviço	Ação	Variável de estado	Valores permitidos para a variável de estado	Círculo de leitura	Círculo de escrita
Higrômetro de solo	monitor	ler_humidade	humidade	De 0 a 1023	Sensor de humidade do solo (YL-69)	--
Esguicho de água	irrigação	ajustar_energia	energia	ligado, desligado	Sensor de voltagem (LM393-3144)	Círculo de chaveamento

3.1.4.5 . CONTROLE DE TEMPERATURA

Tabela 16 - Abstração lógica do sistema de controle de temperatura

Nome do dispositivo	Serviço	Ação	Variável de estado	Valores permitidos para a variável de estado	Círculo de leitura	Círculo de escrita
Ar condicionado e aquecedor	ajustar	mudar_temperatura	temperatura	De 15 a 30	Medidor de temperatura do ar (DS18B20)	Círculo IR
		mudar_modo_operacao	modo_de_operacao	Aquecer, gelar	--	Círculo IR
	energia	ajustar_energia	energia	ligado, desligado	--	Círculo IR
Termômetro	monitor	medir_temperatura	temperatura	-55 to 125	Medidor de temperatura do ar (DS18B20)	--

Este exemplo é análogo ao sistema de irrigação porém, ao invés de monitorar a humidade, ele monitora a temperatura de um ambiente, como uma estufa, um escritório ou um

dormitório. Ao detectar que a temperatura cai ou sobe de um determinado valor, por meio das mensagens GENA que abstraem o termômetro conectado ao controlador-escravo do ambiente, a aplicação do sistema de temperatura envia um pedido de ajuste da temperatura para um determinado valor. Outra aplicação interessante seria manter a temperatura de um ambiente fechado sempre alguns graus a mais ou alguns graus a menos do que a temperatura do ambiente externo. Para isso, teria que possuir um termômetro externo, um interno e o módulo IR para controlar o ar condicionado. A abstração para poder realizar esses dispositivos está apresentada na Tabela 16. Este é um exemplo de como utilizar a arquitetura proposta para influenciar em um contexto baseada em parâmetros de outros contextos.

3.1.4.6 . ESTACIONAMENTO

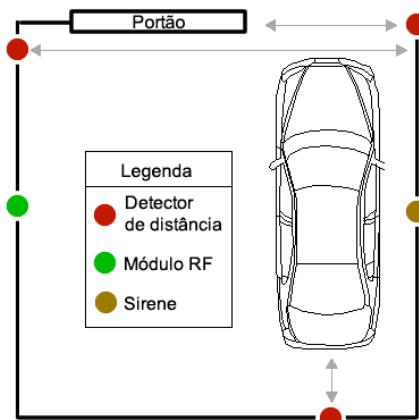


Figura 3.9 - Disposição física de objetos para o sistema de estacionamento

O sexto exemplo de sistema é o de um auxiliar de estacionamento representado na Figura 3.9. Este sistema serve para abrir e fechar o portão e evitar que o motorista colida com a parede, por meio de alarme sonoro. O usuário abriria o portão, via *smartphone*, que faria uma requisição REST ou UPnP para abrir o portão. Depois o *smartphone* ficaria monitorando, por mensagens GENA *unicast* que chegariam a ele, se portão já abriu e se o carro já terminou de passar por ele. Quando se detecta esse acontecimento, por dois sensores de distâncias na parte superior da Figura 3.9, ele enviaria um comando para que o portão fosse fechado, pelo módulo RF. A partir desse momento, o *smartphone* detectaria a distância do carro para a parede de trás e, quando estivesse próximo o suficiente, ele acionaria a sirene. Ao detectar que o carro não se moveu por mais de 5 segundos, ele desligaria a sirene. Também poderia ser utilizado para sair da garagem. O usuário pediria

ao *smartphone* que abrisse o portão e, em seguida, ele tiraria o carro. Ao *smartphone*, por notificações GENA, detecta-se que o carro não está mais na garagem, ordenaria que o portão fosse fechado.

A abstração para poder realizar esses dispositivos está apresentada na Tabela 17. Este é um exemplo de como utilizar a arquitetura proposta para auxiliar o posicionamento e a locomoção de objetos que, no exemplo, foi relacionado a trânsito.

Tabela 17 - Abstração lógica do sistema de estacionamento

Nome do dispositivo	Serviço	Ação	Variável de estado	Valores permitidos para a variável de estado	Círculo de leitura	Círculo de escrita
Alarme	sirene	ajustar	som	ligado, desligado	--	Sirene (YL-44)
Portão	mover	mudar_posicao	movimento	Abrir, fechar, parar	--	Círculo RF
	medidor	medir_distancia		distancia	De 2 a 400	Detector de distância (HC-SR04)
		ler_se_esta_obstruido	distancia	distancia		
Carro	distancia_parede	medir_distancia	distancia	distancia	--	--

3.1.4.7 . CONTROLE DE DISPOSITIVOS DOMICILIARES

O último exemplo aqui abordado é o de controle de vários dispositivos domésticos. Pela abstração proposta na Tabela 18, seria possível controlar televisores, sistemas de som, cortinas elétricas, cafeteiras, portões da garagem e banheiras.

Tabela 18 - Abstração de um controlador de dispositivos doméstico

Nome do dispositivo	Serviço	Ação	Variável de estado	Valores permitidos para a variável de estado	Círculo de leitura	Círculo de escrita
TV	ajustador	mudar_canal	canal	De 1 a 1000	--	Círculo IR
		mudar_volume	volume	De 1 a 100		
	energia	ajustar_energia	energia	ligado, desligado		

Cortina	mover	mudar_posicao	movimento	Abrir, fechar, parar	--	Circuito RF
Cafeteira	energia	ajustar_energia	energia	ligado, desligado	Sensor de voltagem (LM393-3144)	Circuito de chaveamento
Portão	mover	mudar_posicao	movimento	Abrir, fechar, parar	--	Circuito RF
Portão	posicao	ler_posicao	distancia	De 2 a 400	Detector de distância (HC-SR04)	--
Sistema de Som	energia	ajustar_energia	energia	ligado, desligado	--	Circuito IR
Sistema de Som	ajustar	mudar_volume	volume	1 to 100		
Banheira	Ajustar_agua	mudar_estacao	estacao	1 to 6	Sensor de voltagem (LM393 - 3144) com válvula pneumática eletrônica (EV-3M)	Circuito de chaveamento
Banheira	Ajustar_agua	esquentar	atuador_da_valvula_quente	aberta, fechada		
Banheira	Ajustar_agua	esfriar	atuador_da_valvula_fria	aberta, fechada		
Banheira	medir	esvaziar	atuador_do_ralo	aberto, fechado	Medidor de temperatura da água (DS18B20 à prova d'água)	--
Banheira	medir	ler_temperatura	temperatura	-55 to 125	Medidor de temperatura da água (DS18B20 à prova d'água)	--

Para o exemplo de abstração aqui proposto, seria possível, para TVs, ligar e desligar e ajustar o volume e o canal, tudo com infravermelho, da mesma forma que o controle remoto faria, porém aplicações teriam acesso a isso por REST e UPnP. Para cortinas elétricas, seria possível abri-las, fechá-las e pará-las, durante o movimento de abertura ou fechamento, por RF, da mesma forma que seu controle remoto o faria. Para cafeteiras ou outros eletrodomésticos que funcionam basicamente com “liga/desliga”, seria possível ligá-las e desligá-las, utilizando relés e dando o mesmo efeito de conectá-las na tomada, ou

não. Para o portão, seria possível saber se ele está aberto, fechado ou entreaberto e seria possível abri-lo, fechá-lo e pará-lo, durante o movimento de abertura ou fechamento, da mesma forma que seu controle remoto o faria. No caso de sistemas de som, seria possível ligar, desligar, alterar a estação e ajustar o volume por IR, da mesma forma que seu controle remoto o faria. O último item abordado foi uma banheira. Ao instalar válvulas pneumáticas eletrônicas nas comportas de água quente, água fria e no ralo, e instalando um termômetro onde a água é depositada, seria possível gerenciar seu enchimento, esvaziamento e temperatura desejada.

A abstração para poder realizar esses dispositivos está apresentada na Tabela 18. Este é um exemplo de como utilizar a arquitetura proposta para auxiliar no gerenciamento de objetos cotidianos, desde os que têm alguma inteligência eletrônica embutida, como um televisor, aos que não têm nenhuma, como uma banheira.

3.1.5. EXEMPLO DE CENÁRIO DE APLICAÇÃO

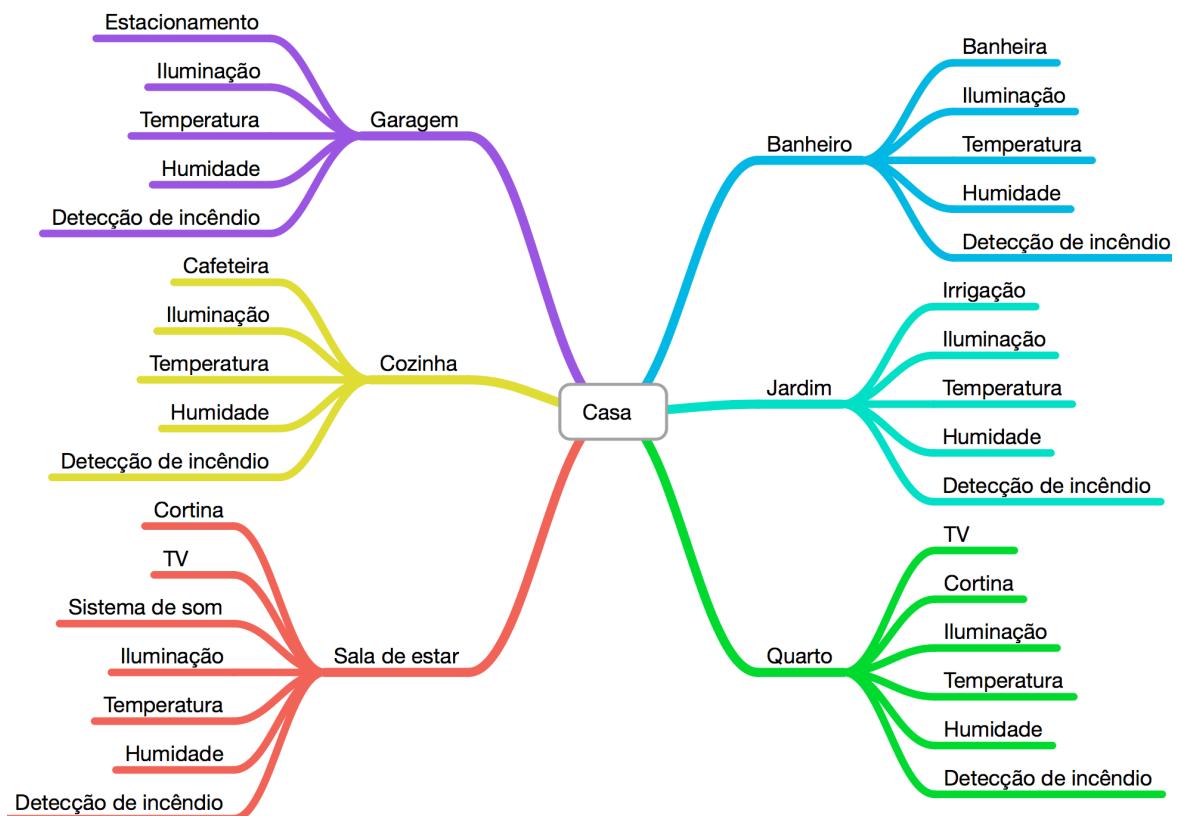


Figura 3.10 - Sistemas por recinto de uma casa

Na seção anterior, foram apresentadas abstrações de dispositivos para que pudessem prestar serviços. Esta seção tem como objetivo colocar todos os sistemas da seção anterior dentro de um mesmo cenário. Foi escolhida uma residência contendo uma garagem, uma cozinha, uma sala de estar, um banheiro, um jardim e um quarto. Para isso, cada serviço foi dividido entre os recintos da casa de acordo com Figura 3.10.

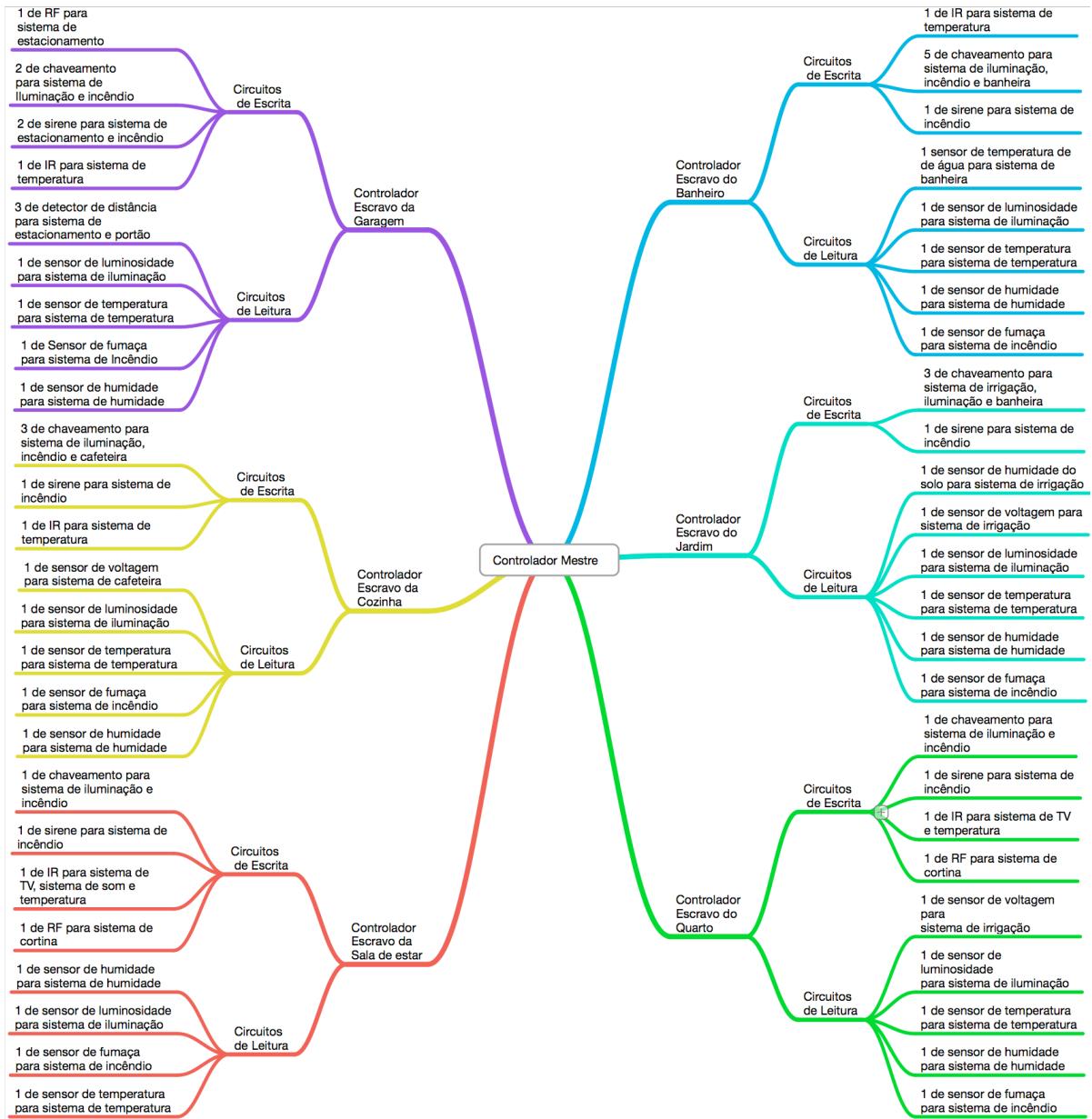


Figura 3.11 - Distribuição física de componentes para cenário de aplicação

Os sistemas de iluminação, temperatura, humidade e detecção de incêndio estão presentes em todos os cômodos e, por isso, seus circuitos devem estar repetidos em todos os ambientes. Para cada recinto, foi implantado um controlador-escravo nos quais foram

acoplados os sistemas de leitura e escrita de cada sistema. Em alguns casos, sistemas puderam compartilhar os circuitos de escrita e leitura, como no recinto sala para a escrita dos sistemas de TV, sistema de som e temperatura. A distribuição e a subordinação de componentes físicos final já adequados à arquitetura proposta neste trabalho estão representadas na Figura 3.11.

A circuitaria proposta nesta seção, utilizando as abstrações da seção 3.1.4, permitirá que aplicações possam utilizar as APIs da camada de serviços para construir os sistemas propostos na seção 3.1.4, todos funcionando simultaneamente abaixo do mesmo controlador-mestre e com todas as resoluções da seção 3.1.3 para facilitar auxiliá-los em suas tarefas.

3.1.6 . SEGURANÇA

O tema segurança não é o objetivo deste trabalho porém, devido à sua importância para a Internet das Coisas (Weber, 2010), é mostrado como utilizar mecanismos para garantir a privacidade, a autenticidade, a integridade e a confidencialidade na troca de informação entre os componentes físicos do *middleware* proposto.

Por utilizar tecnologias já existentes, a arquitetura aqui proposta beneficia-se de muitos métodos nativos em suas camadas para prover segurança.



Figura 3.12 - TLS na comunicação externa

Para a troca de dados na camada de serviço da arquitetura proposta, pode-se utilizar a segurança da camada de transporte do TCP/IP, do inglês *Transport Layer Security* (TLS), nos endereços HTTP para obter o HTTPS e assim trocar dados com confidencialidade (Dierks, 2008). Devido ao uso pelo TLS do padrão de criptografia avançado, do inglês *Advanced Encryption Standard* (AES), também são utilizados os mecanismos de código de integridade de mensagem, do inglês *Message Integrity Code* (MIC), e o de código de autenticidade de mensagem, do inglês *Message Authentication Code* (MAC*), para

garantir, respectivamente, a integridade e a autenticidade das mensagens (McGrew *et al.*, 2012).



Figura 3.13 - AES na comunicação interna

Para garantir a integridade, a autenticidade e a confidencialidade na camada de comunicação, que faz a comunicação entre as partes físicas de controlador-escravo e de controlador-mestre, basta pré-compartilhar uma chave de segurança entre todos os controladores, que são considerados autênticos por princípio, e utilizar o mecanismo de AES provido pela tecnologia ZigBee. Com isso, tem-se um canal criptografado e que aplicam MIC e MAC* para comunicações que utilizem a camada de controle.

Para garantir a privacidade dos dados, é preciso controlar quem tem acesso a quais serviços de dispositivos. Para isso, pode-se adicionar a tecnologia de autenticação como o OAuth (Hardt, 2012) na camada de controle e perfis de usuários para que cada um tivesse seu acesso restrito a áreas da arquitetura.

Os perfis seriam quatro: administradores, donos de objetos, usuários de objetos e aplicações. Administradores poderiam gerenciar a abstração de todas entidades, com campos e funcionalidades especificadas na seção 3.1.2, por meio da *interface web*, provida pela camada de controle. O administrador não pode executar ações em dispositivos e nem ver seus estados mas pode criar qualquer outro tipo de usuário. O dono de objeto é quem é responsável por adicionar uma abstração lógica de dispositivo que conectou a um controlador-escravo. O dono de objeto pode ver e gerenciar apenas abstrações cadastradas por ele no sistema e pode, inclusive, controlar e ver o estado de tudo aquilo que inseriu. Ele pode criar usuários do tipo usuário de objeto e aplicação e dar a eles permissão de uso por ação, serviço ou dispositivo que gerencia. O usuário de objeto pode acessar a *interface web* e controlar e ver o estado de dispositivos. As aplicações, quando autorizadas por um dono de objeto ou usuário de objeto, podem executar ações e ver o estado de todos os dispositivos ou serviços ou ações para as quais foi autorizada.

Todos os perfis, com exceção do de aplicação, podem criar grupos de usuários e adicionar usuários aos grupos e dar permissões ao grupo de acesso a dispositivos, serviços ou ações que desejar. Todos os usuários de um grupo ganham acesso aos recursos liberados para aquele grupo. Qualquer usuário, exceto perfil de aplicação, pode dar autorização a um grupo, mesmo que não tenha sido criado por ele, para acessar um determinado recurso que lhe pertença.

O oAuth controla o acesso a recursos de duas maneiras, cujas interações estão mostradas na Figura 3.14. Sessões e *tokens* podem ser reutilizados em outras requisições até que expirem, evitando assim o *login* ou pedido de uso de recurso para cada requisição adjacente.

Recursos aqui mencionados são ler ou receber valores de variáveis de estado, tomar ações em serviços, usuários do sistema, ou gerenciar abstração de controladores escravos, dispositivos, serviços, ações, variáveis de estados ou argumentos de entrada e saída.

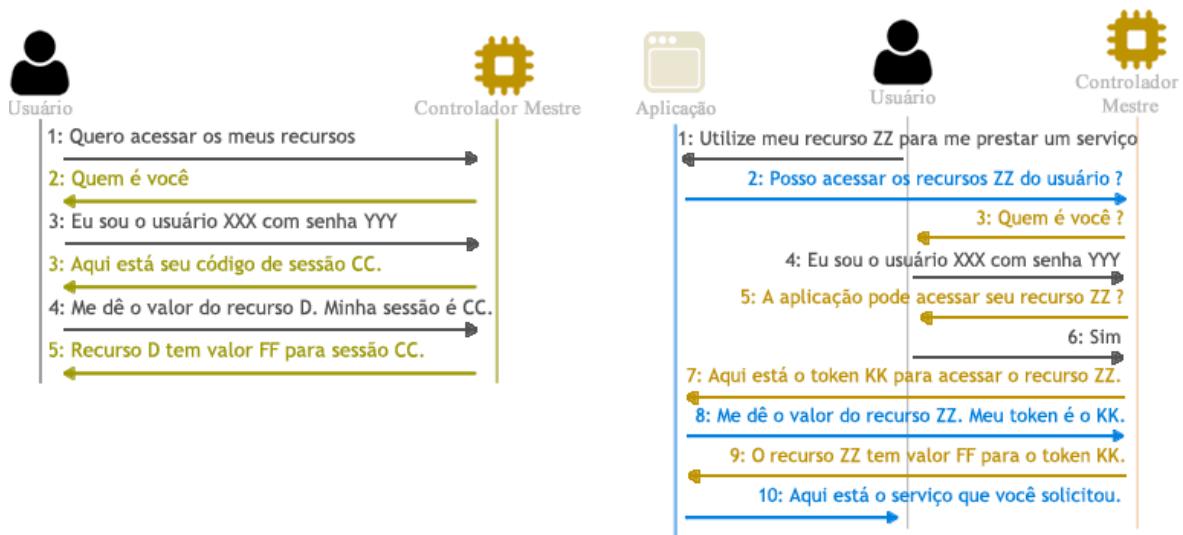


Figura 3.14 - Métodos de autenticação oAuth

Com o uso do TLS para comunicações com partes externas à arquitetura, AES para comunicações entre partes internas da arquitetura e oAuth e perfis para requisições quaisquer, é dado um nível de segurança básico e desejável à arquitetura, trazendo, como pretendido, privacidade, autenticidade, integridade e confidencialidade às comunicações e aos dados.

3.2. SÍNTESE DO CAPÍTULO

Neste capítulo, foi mostrado todo o funcionamento da arquitetura de *middleware* para a Internet das Coisas proposta por este trabalho. Foi mostrado como utilizá-la para trazer pervasividade e mobilidade para dispositivos, transparência e segurança para dispositivo e aplicações. Foi apresentado como a flexibilidade entre suas partes lógicas torna escalável e capaz de atender a volumes pequenos e grandes de requisições e ainda assim manter sua extensibilidade para novos dispositivos.

Neste capítulo, também foi mostrado um cenário de uso, com sistemas que podem se beneficiar da arquitetura, e ilustrando como abstrair dispositivos reais sortidos para que possam ser controlados através do *middleware*.

4. PROTÓTIPO FUNCIONAL

Baseado nas tecnologias relacionadas expostas na seção 2.2 e na arquitetura de *middleware* para Internet das Coisas, proposta na seção 3.1, foi codificada uma arquitetura real para validar as hipóteses teóricas propostas e resolvidas na seção 3.

Por falta de recursos financeiros e temporais, o protótipo produzido não foi para cenários gigantescos que demandem um serviço para larga escala, ou seja, não foi utilizado *clusters* de computadores para o controlador-mestre e não foi utilizado banco de dados NoSQL para armazenamento de informações. Porém, devido à grande difusão de serviços de computação em nuvem, é possível afirmar que o protótipo é escalável por basear-se, principalmente, em tecnologias *web*, como todas as aplicações escaladas, via computação em nuvem.

4.1. MODELO RELACIONAL DOS DADOS

O modelo gerado é baseado nos relacionamentos da Figura 3.4 - Relacionamento da abstração lógica de dispositivos, acrescido dos valores possíveis para uma variável de estado, controladores escravos, histórico de alterações de uma variável de estado e de usuários e recursos para o qual ele tem autorização de uso.

Os dados a serem armazenados no banco de dados, com exceção da tabela de histórico, dificilmente passarão de 1 bilhão de registros, o que não pede por uso da tecnologia NoSQL. Para implementar o banco de dados, foi escolhido o MySQL pois há a necessidade de velocidade e, devida à baixa complexidade do modelo de dados, não há a necessidade do uso estendido de JOINS nas pesquisas, o que traria a vantagem no uso do PostgreSQL. Caso fosse utilizado o NoSQL para armazenar a tabela histórico, recomendaria-se o open-source Cassandra.

A Figura 4.1 traz o modelo relacional utilizado no protótipo. Nesta imagem, constam as entidades, seus campos, os tipos dos campos e o relacionamento entre elas. Cada campo é iniciado por duas letras representando seu tipo sendo PK para chave primária, FK para chave estrangeira, TE para texto, IN para inteiro, FL para número de ponto flutuante, BO para booleanos, DT para datas e EN para listas enumeráveis de valores.

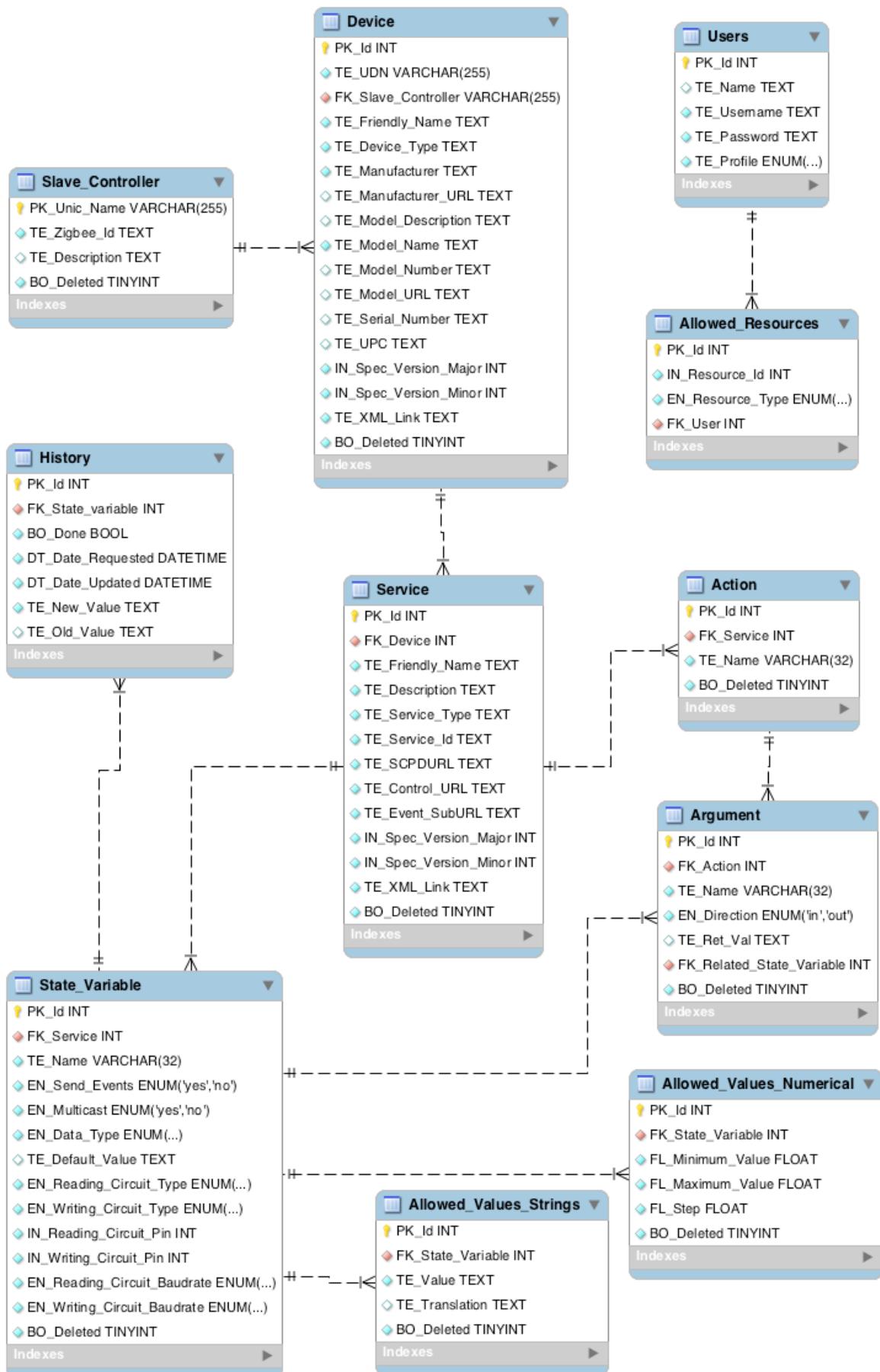


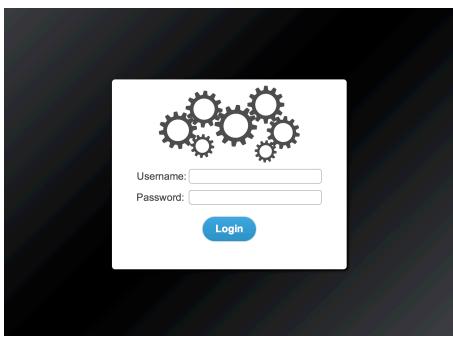
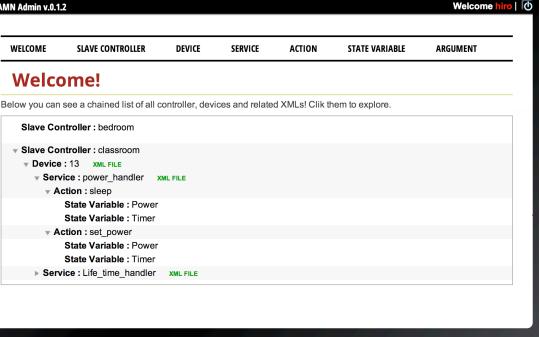
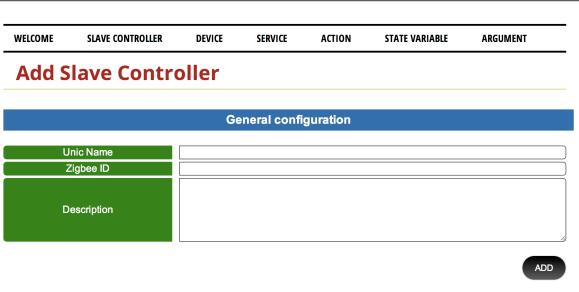
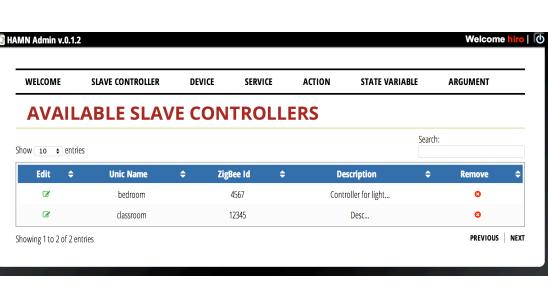
Figura 4.1 - Modelo relacional do protótipo

4.2. ADMINISTRAÇÃO DE ABSTRAÇÕES

O sistema de administração permite gerenciar as abstrações lógicas das entidades controlador-escravo, dispositivo, serviço, ação, variável de estado e argumento. Ele é acessível por meio de *interface web* e fica hospedado no servidor Web Apache do controlador-mestre.

Primeiramente, é preciso fazer autenticação para gerenciar qualquer abstrações lógica. A tela inicial conta com um lista encadeada de todas as entidades do sistema e *links* para os XML de descrição de dispositivo e de serviço disponíveis no sistema. Todas as telas do sistema possuem um *menu* superior que possui *link* para as telas de cadastrar nova entidade e listar entidades já existentes. As telas de listagem possuem um *link* para editar cada entidade listada ou removê-la do sistema. Com essas telas, faz-se a criação, a leitura, a atualização e a exclusão, do inglês *Create, Read, Update and Delete* (CRUD), de todas as entidades do sistema. As alterações feitas por esse sistema alteram o banco de dados e devem notificar a API UPnP das alterações para que possam recarregar os dispositivos que foram alterados. A Tabela 19 traz uma impressão de cada tela do sistema.

Tabela 19 - Telas do sistema de gerência de abstração de dispositivos

Tela de Login	Tela de boas vindas
	
Tela de cadastrar controlador escravo	Tela de listar controladores escravo
	

<h3>Tela de alterar controlador escravo</h3>	<h3>Tela de cadastrar dispositivo</h3>
<h3>Tela de listar dispositivos</h3>	<h3>Tela de alterar dispositivo</h3>
<h3>Tela de cadastrar serviço</h3>	<h3>Tela de listar serviços</h3>

<h3>Tela de alterar serviço</h3>	<h3>Tela de cadastrar ação</h3>
<h3>Tela de listar ações</h3>	<h3>Tela de alterar ação</h3>
<h3>Tela de cadastrar variável de estado</h3>	<h3>Tela de listar variáveis de estado</h3>

O sistema foi desenvolvido utilizando a linguagem *web* de programação PHP, a mais utilizada para a construção de *web sites* e que, em 2013, foi a utilizada na construção de mais de 244 milhões de *sites* (PHP, 2014).

4.3 . CONTROLADOR-MESTRE

O controlador-mestre é o retentor das camadas de serviço, de controle e de comunicação da arquitetura proposta. Com isso, ele engloba a APIs REST, a API UPnP, o banco de dados, a *interface* de administração, o código de comunicação com os controladores-escravos e o nó coordenador da rede ZigBee. O computador utilizado para servir como essa entidade possui sistema operacional OS X v10.9, 8 GB de memória RAM DDR3 1600 MHz, processador 2.2 GHz quad-core Intel Core i7 com disco de unidade de estado sólido(SSD) de 256 GB, sendo que nenhum dos componentes foi de uso exclusivo para transações da arquitetura.

Para a API REST, localizada na camada de serviços, foi utilizado o mesmo servidor *web apache* do sistema de gerenciamento de abstrações lógicas e foi escrita uma outra aplicação, também com PHP, para gerenciar os pedidos *web*. Ao receber demandas de controle, o código PHP abre um *socket TCP* para comunicar-se com a camada de controle e requisitar a ação.

No caso da API UPnP, localizada na camada de serviços, foi utilizado o HERQQ, que é uma biblioteca de *software*, escrita em C++ e baseada no *framework* QT, para a construção dinâmica de dispositivos UPnP e pontos de controle em conformidade com a Arquitetura de dispositivos UPnP, versão 1.1 (HUPnP, 2014).

Para a lógica da arquitetura, localizada na camada de controle, foi utilizado C++, em cima do *framework* QT, para fazer as consultas ao banco de dados com a maior velocidade possível.

Para o banco de dados, localizado na camada de controle, foi utilizado o MySQL, como já explicitado na seção 4.1.

Para a administração das abstrações lógicas, localizada na camada de controle, foi utilizado o servidor Web Apache e PHP, como já explicitado na seção 4.2.

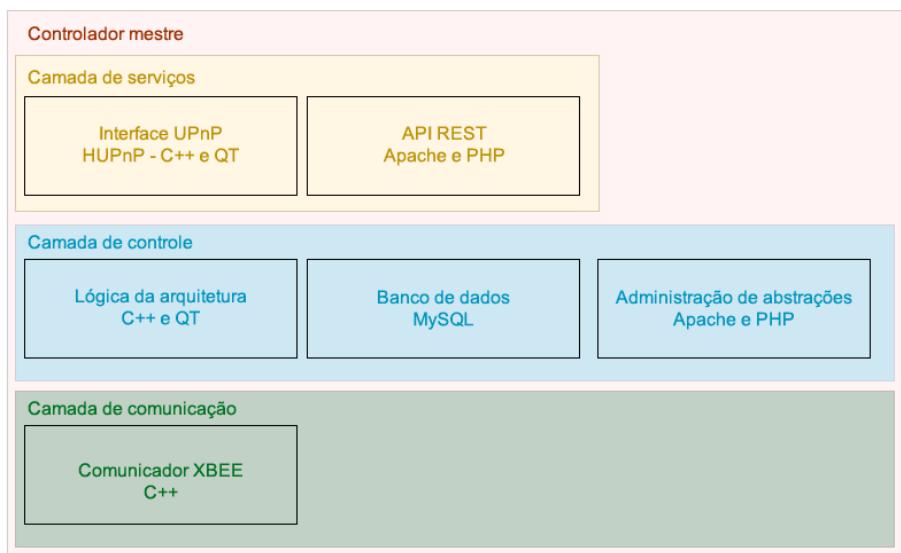


Figura 4.2 - Componentes do controlador-mestre

Tanto o Apache, quanto o PHP, quanto os códigos utilizando o QT são portáveis, ou seja, podem ser utilizados em outras máquinas e sistemas operacionais. Com isso, os códigos produzidos podem ser repassados para uma Raspberry Pi, Windows ou Linux, e manter sua funcionalidade. A Figura 4.2 traz o resumo de partes dessa entidade física.

A aplicação usada para testes é uma página *web* simples que contém uma imagem de um interruptor e uma imagem de uma lâmpada. A cada 2 segundos, a página *web* faz uma requisição AJAX para checar se a lâmpada real trocou de estado. Caso sim, ele atualiza as imagens, ascendendo ou apagando a lâmpada, e alterando o estado do interruptor. Caso o usuário clique no interruptor, a aplicação tentará alterar o valor da lâmpada através de outra requisição AJAX, porém dessa vez de controle. As telas do aplicativo estão mostradas na Figura 4.3.

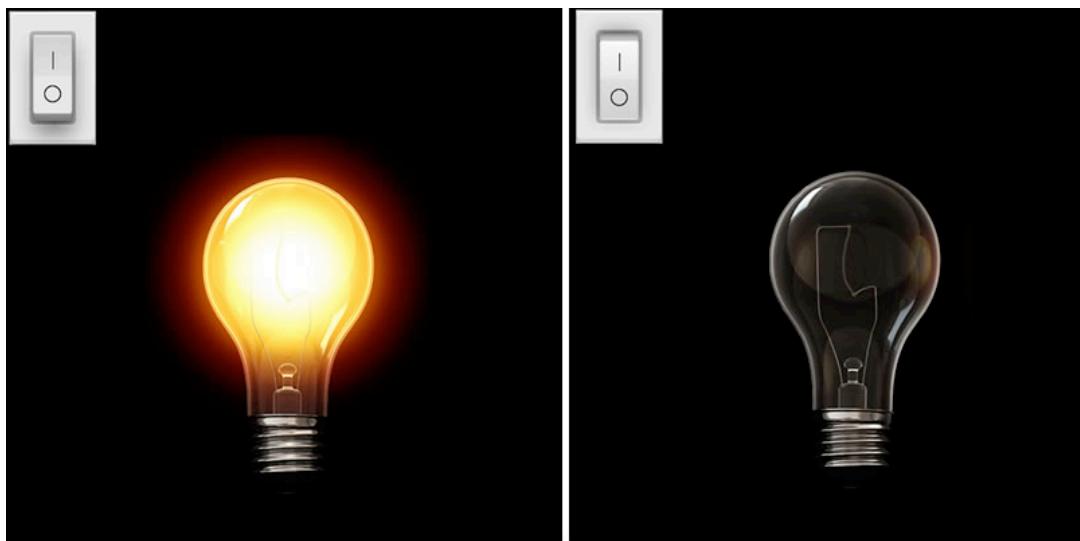


Figura 4.3 - Telas do aplicativo de testes

4.4. COMUNICADOR XBEE

O comunicador XBEE foi escrito em C++ e sua função, no controlador-mestre, é tratar os JSONS da camada de controle, converter em pacotes Zigbee e enviar pelo cabo USB para o módulo XBEE. O módulo XBEE Mestre (coordenador ZigBee) enviará para o módulo XBEE do controlador-escravo (roteador) devido. Ao receber os dados no controlador-escravo, o código comunicador XBEE vai receber, via porta serial do módulo XBEE, os dados. Em seguida, ele deve convertê-los e JSON novamente e repassar para a camada de execução. Sua função é análoga no sentido de comunicação inverso.

Foram utilizados três módulos XBEE PRO Serie 2 com protocolo Digimesh sem criptografia, cujas configurações foram feitas com o *software* XCTU 6.0. Para realizar a conexão dos módulos XBEE com os controladores-escravo, foi utilizado o circuito WRL-11373. Para Ligar o módulo na porta USB do controlador-mestre, foi utilizado o WRL-11697.

Os códigos para traduzir JSONs em pacotes ZigBee, traduzir pacotes ZigBee em JSONs, realizar a comunicação USB do controlador-mestre com o módulo XBEE, e a comunicação serial do controlador-escravo com o módulo XBEE, foram escritas em C++ e foram testadas nos sistemas operacionais OS X e UBUNTU, em que funcionaram como o esperado.

Todos os códigos relativos ao comunicador XBEE se referem à camada de comunicação e estão distribuídos entre os controladores-escravo e o controlador-mestre.

A primeira placa arduino não possui nenhum dispositivo conectada a si, apenas serve para rotear na rede ZigBee. A segunda placa possui uma lâmpada e um detector de luminosidade para saber se a lâmpada está ligada ou não.

4.5. CONTROLADOR-ESCRAVO

Devido à sua fácil programação, baixo custo de aquisição e baixa capacidade de processamento, o microcontrolador Arduino foi escolhido para representar a entidade controlador-escravo em ambientes mais limitados do que se fosse escolhido o microcontrolador Raspberry Pi ou Intel Galileo.

Foram utilizados dois controladores-escravo simultaneamente, um sendo uma Arduino ATmega 1280 e o outro uma Arduino nano ATmega328, para testar a rede ZigBee em funcionamento.

O código Arduino dá suporte parcial a interrupções de alterações de pinos. A interrupção nesse microcontrolador pode ocasionar perda de dados seriais e sobrecarga de processamento e, dependendo do modelo da placa, somente alguns pinos conseguem receber essa funcionalidade (Arduino, 2014). Por esses motivos e tendo em vista que o intuito aqui é de validar a troca de estado de todos os pinos mantendo a comunicação

serial, foi decidido monitorar todas as entradas da placa a cada 500ms, comparar com o último resultado obtido e, caso encontre diferenças, notifica o controlador-mestre sobre o novo valor no pino, como especificado na seção 3. Seus componentes estão representados na Figura 4.4.

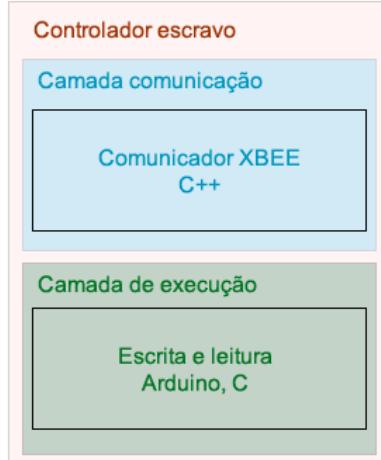


Figura 4.4 - Componentes do controlador-escravo

4.6. CENÁRIO DE TESTES E RESULTADOS OBTIDOS

É importante ressaltar que a programação do protótipo aqui descrito foi feita em cooperação com 3 alunos de graduação do departamento de Engenharia Elétrica da Universidade de Alexandria, Egito, e que testes intercontinentais foram feitos para averiguar a funcionalidade da solução implementada.

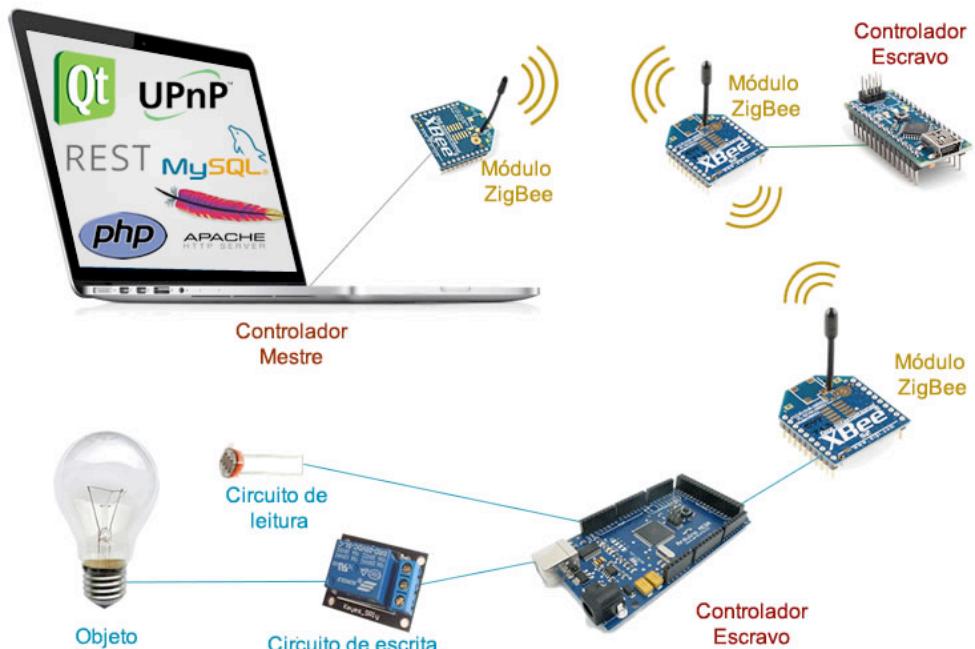


Figura 4.5 - Componentes físicos do protótipo

A organização do *middleware* para validação ficou como mostrado na Figura 4.5. Um controlador-mestre e dois controladores-escravo. Um escravo serve de rotador para os pacotes trocados entre o controlador-mestre e o escravo que contém o objeto a ser controlado e monitorado, podendo assim testar a rede ZigBee em malha.

As entidades foram configuradas como mostrado nas imagens de 4.6 a 4.11, que contêm uma impressão da tela do sistema da configuração do protótipo.

Edit	Unic Name	ZigBee Id	Description	Remove
<input checked="" type="checkbox"/>	room_mega	0013A20040A14151	Controller for light...	
<input checked="" type="checkbox"/>	room_nano	0013A20040A859FC	ZigBee ADDR is 0013A...	

Figura 4.6 - Configuração de controladores-escravo

Edit	Friendly Name	Slave Controller	Remove
<input checked="" type="checkbox"/>	Bedroom Light [13]	room_mega	

Figura 4.7 - Configuração de dispositivos

Edit	Service	Device it belongs	Slave Controller	Remove
<input checked="" type="checkbox"/>	power_handler [18]	Bedroom Light [13]	room_mega	

Figura 4.8 - Configuração de serviço

Edit	Action	Service it belongs	Device it belongs	Slave Controller	Remove
<input checked="" type="checkbox"/>	set_power [2]	power_handler [18]	Bedroom Light [13]	room_mega	
<input checked="" type="checkbox"/>	get_power [5]	power_handler [18]	Bedroom Light [13]	room_mega	

Figura 4.9 - Configuração de Ações

Edit	State variable	Service it belongs	Device it belongs	Slave Controller	Remove
<input checked="" type="checkbox"/>	Power [2]	power_handler [18]	Bedroom Light [13]	room_mega	

Figura 4.10 - Configuração de variável de estado

Edit	Argument	Action	State Variable	Service	Device	Slave Controller	Remove
<input checked="" type="checkbox"/>	power_value [1]	set_power [2]	Power [2]	power_handler [18]	Bedroom Light [13]	room_mega	
<input checked="" type="checkbox"/>	set_power_to [2]	set_power [2]	Power [2]	power_handler [18]	Bedroom Light [13]	room_mega	
<input checked="" type="checkbox"/>	value [9]	get_power [5]	Power [2]	power_handler [18]	Bedroom Light [13]	room_mega	

Figura 4.11 - Configuração de Argumento de entrada e saída

A prova teórica da proposta de arquitetura de *middleware* pervasiva, móvel, transparente, extensível e escalável para IoT foi apresentada na Seção 3. O intuito de validação aqui é se ela é funcional e possível de existir. Tomando essa premissa, a configuração de entidades e de disposição física escolhidas nesta subseção permitem validar o funcionamento das comunicações externas e internas da arquitetura proposta e, consequentemente, se ela é funcional e implementável.

O circuito de escrita é um módulo de chaveamento e está ligado no pino digital 13 da Arduino Mega. O circuito de leitura é um sensor de luminosidade que é lido no pino analógico A0 da mesma Arduino Mega. Tal microcontrolador está conectada a um módulo XBEE de endereço ZigBee estendido 0013A200 40A14151 que só tem alcance do rádio do outro controlador-escravo. O objeto monitorado e controlado por esse conjunto é uma lâmpada que fica ligada diretamente à tomada e cujo fio de energia foi interceptado pelo relé de controle.

A segunda Arduino é uma Nano, cujo módulo XBEE tem alcance dos módulos do controlador-mestre e do outro controlador-escravo. Sua função é de rotear pacotes entre os dois componentes da rede ZigBee como prova de funcionamento da topologia em malha.

Os demais componentes, físicos e lógicos, já foram explicitados nas imagens de 4.6 a 4.11 a nas seções 4.3, 4.4 e 4.5.

Após codificar todas as camadas da arquitetura proposta, a configuração do dispositivo lâmpada, desde ligar no controlador-escravo até criar toda sua abstração lógica, assim permitindo seu monitoramento e controle por meio das APIs, tomou menos de 15 minutos.

O aplicativo de teste abordado no final da seção 4.3 realizou requisições para os serviços *get_power* e *set_power* para, respectivamente, ajustar a imagem da lâmpada e do

interruptor que exibe para o usuário, e para trocar estado da lâmpada quando o usuário clicar na imagem do interruptor.

Foram monitoradas 309 requisições de ação na lâmpada num intervalo de 30 segundos utilizando o Web Inspector do navegador Google Chrome. O gráfico com as latências está exibido na Figura 4.12. Para esse teste, foi obtida a média de 483 bytes por resposta de requisição.

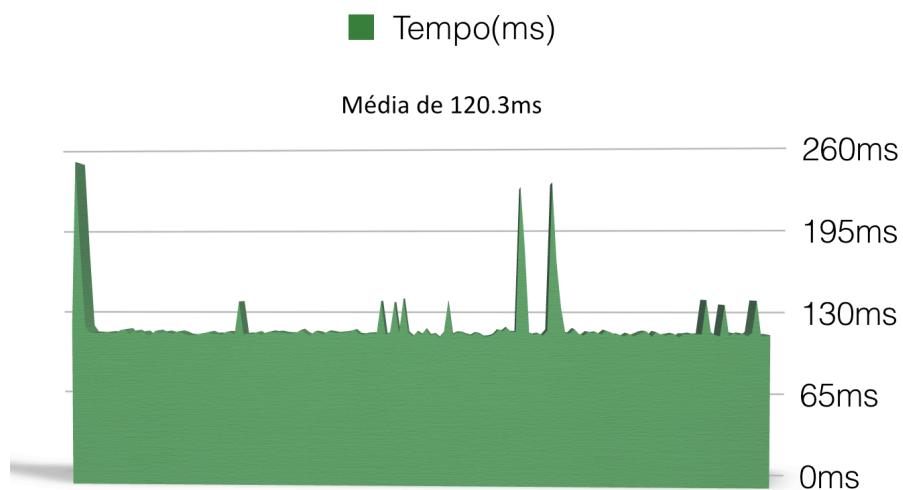


Figura 4.12 – Tempo para atender requisições de controle

Foi escrito um código auxiliar para monitorar o uso percentual da CPU e da memória RAM para os códigos QT, banco de dados MySQL e para o servidor Apache, assim englobando todas as partes do controlador mestre. Foram coletadas 2 mil amostras nos mesmo trinta segundos em que foi observada a latência das requisições. Os gráficos estão exibidos na Figura 4.13, na Figura 4.14 e na Figura 4.15.

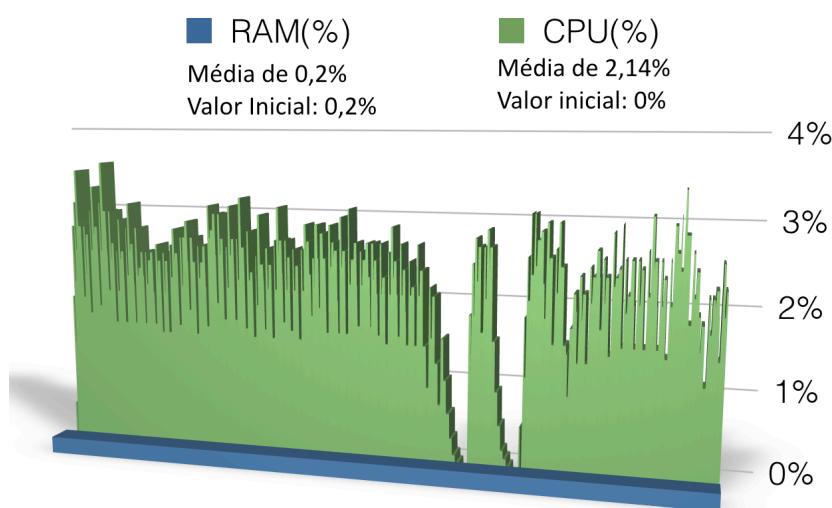


Figura 4.13 - Utilização de RAM e CPU por códigos QT do controlador-mestre

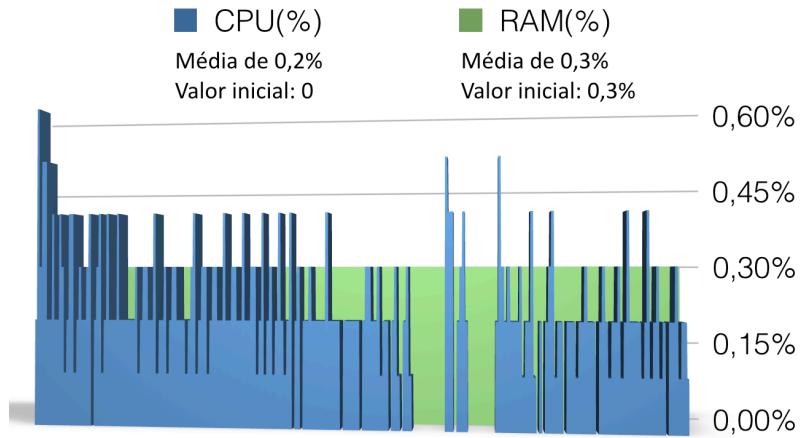


Figura 4.14 - Utilização de RAM e CPU por servidor MySQL do controlador-mestre

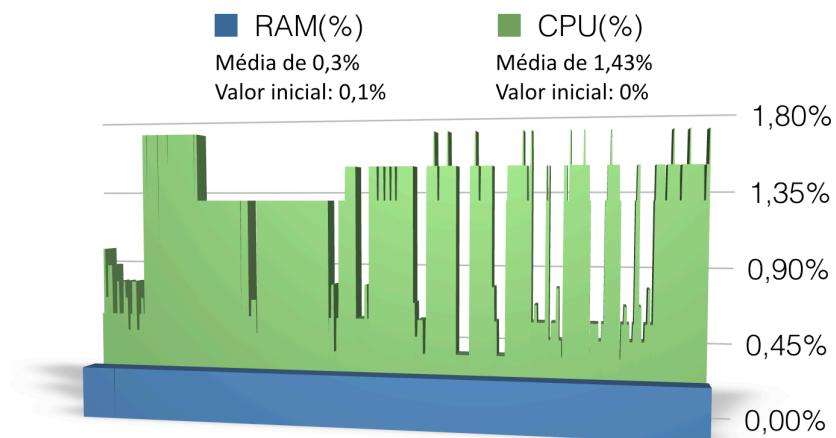


Figura 4.15 - Utilização de RAM e CPU por servidor Apache do controlador-mestre

O tempo médio para as requisições sem utilizar a rede ZigBee, apenas retornando um valor randômico de sucesso ou erro na lógica da camada de controle, ou seja, retornando requisições sem passar para controladores-escravo, foi de 8 milissegundos. Houve uma redução de mais de 90% na latência das requisições, as respostas ficaram 15 vezes mais rápidas. Acredita-se que o ocorrido se dá devido à comunicação do computador hospedeiro do controlador mestre com o módulo XBEE através de conexão USB.

Para reduzir a latência na rede ZigBee seria possível experimentar a conexão do módulo XBEE com taxa de transmissão mais elevada, que por padrão, é de 9600. Assim como os módulos XBEE, nenhum outro componente sofreu otimização de configuração, foram utilizados com as configurações padrão para simplificar testes comparativos de trabalhos futuros com a arquitetura aqui proposta.

Mesmo com a latência média de 120 milissegundos, a arquitetura se mostrou bastante ágil e consumindo pouquíssimos recursos do hospedeiro, o que a torna adequada para uso em hospedeiros mais restritos em processamento, energia e memória.

A configuração do dispositivo através dos formulários web foi eficaz para torná-lo padronizado e utilizável com transparência para aplicações com API REST e UPnP, simultaneamente. Com a entrada padronizada de dados, a arquitetura pôde garantir interoperabilidade entre dispositivos distintos, como discutido na seção 3.

A mobilidade trazida pela rede ZigBee e a flexibilidade trazida pela Arduino, suas bibliotecas e seus circuitos, fazem com que grande gama de dispositivos possa ser englobada pela arquitetura. No protótipo, foram utilizados os conceitos básicos de controle e evento e, com as abstrações sugeridas na seção 3.1.5, muitos outros dispositivos poderiam passar a fazer parte da arquitetura sem empasses e serem utilizados por meio de suas interfaces uniformes. Utilizando os códigos e técnicas propostas nesta seção, os 69 circuitos que englobam o exemplo completo de uso arquitetura mostrado na seção 3.1.5 poderiam ser escolhidos de forma a custar menos de mil reais(BRL) e poderiam ser montados e configurados na arquitetura em menos de 40 horas.

Como já apresentado, as partes da arquitetura são desacopladas e baseadas em tecnologias *web*. Com isso, ela pode ser escalada utilizando as técnicas da computação em nuvem e pode ser replicada facilmente para outros sistemas operacionais e hospedeiros.

4.7. SÍNTESE DO CAPÍTULO

Neste capítulo, foi mostrado como utilizar o modelo teórico e abstrações propostas na seção 3, pelas tecnologias apresentadas na seção 2, trazendo assim um protótipo de *middleware* para Internet das Coisas extensível e escalável que habilita a pervasividade, a mobilidade e a transparência.

Aqui foram abordadas questões técnicas de implementação como quais tecnologias utilizar em cada camada da arquitetura proposta para trazer o *middleware* à vida com facilidade. Também foi proposto um ambiente de testes e homologação das partes da arquitetura para averiguar se são funcionais e qual é o uso de recursos das máquinas hospedeiras. Os resultados obtidos com esses testes, e de outros que possam ser feitos em cima do protótipo, podem servir para descobrir novos desafios na área de IoT, auxiliar na base

comparativa para trabalhos relacionados e/ou futuros, e para encontrar gargalos dentro da própria arquitetura a serem explorados e mitigados.

5. CONCLUSÕES

Nesta dissertação, foi proposta uma arquitetura de *middleware* para a Internet das Coisas, tema largamente explorado em pesquisas acadêmicas e privadas. Nessa arquitetura, foi proposto um padrão de comunicação simples e implementável, baseado em camadas desacopladas e capaz de se adaptar a múltiplos cenários. Foi proposta uma solução que permite explorar a fundo, com experimentos reais, o ambiente da IoT e que pode ser adaptada, melhorada e estendida para assim prover um padrão que, de fato, se ajuste às necessidades que ainda serão descobertas para a IoT. Assim, buscou-se a interoperabilidade de dispositivos e de aplicações, com o uso racional dos recursos, possibilitando a evolução e a descoberta de novos desafios dentro da área.

A metodologia utilizada neste trabalho, dividida em fases, mostrou-se eficaz para o direcionamento da construção da arquitetura proposta. A pesquisa bibliográfica realizada identifica o estado da arte sobre tecnologias envolvendo a IoT, às quais foram relevantes e imprescindíveis para o desenvolvimento do trabalho e da arquitetura final.

O *middleware* proposto neste trabalho, dividido em entidades físicas e lógicas desacopladas, mostrou-se promissor e eficaz para controlar dispositivos e monitorar seu estado. Para uma gama extensa de dispositivos, foi possível propor um modelo transparente, extensível e escalável que trouxe pervasividade e manteve a mobilidade dos dispositivos englobados, mesmo quando utilizado para aqueles considerados como objetos simples.

O protótipo construído, por ser baseado em tecnologias já existentes e difundidas, é de fácil entendimento, utilização e reconstrução. Com ele, foi possível averiguar e atestar a viabilidade técnica do modelo proposto para uso de aplicações capazes de controlar e monitorar dispositivos via APIs REST e UPnP uniformes.

Por fim, os resultados obtidos com testes no protótipo e as abstrações de dispositivos demonstradas pelo modelo teórico permitem verificar que a arquitetura proposta é passível de escalonamento e de pervasividade. Testes em redes locais e intercontinentais foram feitos, em colaboração com alunos da Universidade de Alexandria no Egito e, em ambos os casos, foi possível observar que as partes lógicas da arquitetura fizeram uso reduzido de

recursos computacionais existentes nos seus hospedeiros. Ainda nos testes, foi notado que, mesmo utilizando configurações padrões das tecnologias que engloba e com as limitações da comunicação serial USB, o modelo proposto possui uma performance adequada para aplicações em tempo real de controle e notificação de alteração no estado de dispositivos presentes nas implementações.

5.1. TRABALHOS FUTUROS

Como proposta de trabalhos futuros são indicadas alguns pontos que podem ser evoluídos, testados e/ou implementados.

Existe a necessidade de aumentar a quantidades de APIs disponíveis na camada de serviços. Como a comunicação entre as camadas é bem definida, qualquer API pode ser adicionada à arquitetura, desde que converta seus comandos nos JSONs de comunicação entre camadas.

Se à camada de comunicação fossem adicionadas comunicações via outras tecnologias, como *bluetooth*, *WiFi*, 6LoWPAN, DTMF e *power line*, muitos outros dispositivos poderiam ser acessados uniformemente pelas APIs, ficando a abstração e a uniformização dos serviços e dispositivos de responsabilidade da camada de controle.

Também existe a necessidade de um estudo mais profundo de segurança, focando em privacidade de dados e na confiança entre controladores-escravo, com autenticação para execução de mensagens de controle.

Outro ponto importante e sempre interessante é a proposição de mais cenários de aplicação completos, como na seção 3.1.4. Para o cenário aqui proposto e para os novos sugeridos, é importante que ocorram a posterior as implementações e a análise de performance nesses ambientes.

5.2. PUBLICAÇÕES RELACIONADOS A ESTE TRABALHO

- 1) Ferreira, H. G. C., Dias Canedo, E., & de Sousa, R. T. (2013, Outubro). IoT architecture to enable intercommunication through REST API and UPnP using IP, ZigBee and arduino. In *Wireless and Mobile Computing, Networking and*

- Communications (WiMob), 2013 IEEE 9th International Conference on* (pp. 53-60), IEEE.
- 2) Ferreira, H. G. C., Canedo, E. D., & Junior, R. T. D. S. (2014). A ubiquitous communication architecture integrating transparent UPnP and REST APIs. *International Journal of Embedded Systems*, 6 (2), 188-197.
 - 3) Ferreira, H. G. C.; de Sousa, R. T.; de Deus, F. E. G.; Canedo, E. D.(2014, Junho). Proposal of a secure, deployable and transparent middleware for Internet of Things. *Information Systems and Technologies (CISTI), 2014 9th Iberian Conference on*, vol., no., pp.1,4, 18-21 doi: 10.1109/CISTI.2014.6877069.

REFERÊNCIAS BIBLIOGRÁFICAS

- ALIANÇA IPSO (2014), disponível em <www.ipso-alliance.org>, acesso em 10.jun.2014.
- ARAÚJO, R. B. de, *Computação Ubíqua: princípios, tecnologias e desafios*, in XXI Simpósio Brasileiro de Redes de Computadores (Vol. 8, pp. 11-13), mai.2003.
- ARDUINO Board (2014), disponível em <www.arduino.cc>, acesso em 11.jun.2014.
- ASHTON, Kevin, *That ‘internet of things’ thing*, RFID Journal, 22, 97-114, 2009.
- ATMEL Corporation (2014), disponível em <www.atmel.com>, acesso em 11.jun.2014.
- ATZORI, Luigi; IERA, Antonio; MORABITO, Giacomo, *The internet of things: A survey*, Computer networks, v. 54, nº 15, p. 2787-2805, 2010.
- BARONTI, P., Pillai, P., Chook, V. W., Chessa, S., Gotta, A., & Hu, Y. F. (2007). *Wireless sensor networks: A survey on the state of the art and the 802.15. 4 and ZigBee standards*. Computer communications, 30 (7), 1655-1695.
- BERNERS-LEE, T., Hendler, J., & Lassila, O. (2001), *The semantic web*. *Scientific american*, 284 (5), 28-37.
- BRAY, T. (2014). *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC7158. Disponível em <tools.ietf.org/html/rfc7158>, acesso 8.jul.2014.
- BRAY, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., & Yergeau, F. (1998). *Extensible markup language (XML)*. World Wide Web Consortium Recommendation REC-xml - 19980210. <http://www.w3.org/TR/1998/REC-xml-19980210>.
- BROADCOM (2014), disponível em <www.broadcom.com>, acesso em 11.jun.2014.
- CHEN, G., & Kotz, D. (2000). *A survey of context-aware mobile computing research* (Vol.1, nº 2.1, pp. 2-1). Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College.
- COHEN, J., Aggarwal, S., & Goland, Y. Y. (2000). *General event notification architecture base: Client to arbiter*. work in progress, Internet draft, expired Apr.
- DIERKS, T. (2008). *The transport layer security (TLS)*, protocol version 1.2.
- Digital Living Network Alliance, DLNA (2013), disponível em <www.dlna.org>, acesso 9.jul.2014.
- FIELDING, R. T., & Taylor, R. N. (2002). Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2 (2), 115-150.
- GERSHENFELD, N., & Cohen, D. (2006). Internet 0: Interdevice internetworking-End-to-end modulation for embedded networks. Circuits and Devices Magazine, IEEE, 22(5), 48-55.

- GUBBI, J., Buyya, R., Marusic, S., & Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7), 1645-1660.
- HANSMANN, U. (Ed.). (2003). Pervasive computing: The mobile world. Springer.
- HARDT, D. (2012). The OAuth 2.0 authorization framework.
- HECHT, Robin, and S. Jablonski. "NoSQL Evaluation." *International Conference on Cloud and Service Computing*. 2011.
- HUI, J., & Thubert, P. (2011). Compression format for IPv6 datagrams over IEEE 802.15. 4-based networks. RFC-6282, disponível em <tools.ietf.org/html/rfc6282>, acesso 9.jul.2014.
- HUPnP (2014). Disponível em <hupnp.linada.fi>, acesso 30.jul.2014.
- Hydra Middleware, (2014). Middleware for networked devices. Disponível em <<http://www.hydramiddleware.eu>>. Acesso em 10 jul. 2014.
- iCore, (2014). iCore, empowering IoT through cognitive technologies. Disponível em <www.iot-icore.eu>. Acesso em 10 jul. 2014.
- IEEE Standards, IEEE 802.15™: Wireless Personal Area Networks (PANs). Disponível em <<http://standards.ieee.org/about/get/802/802.15.html>>, acesso 7.jul.2014.
- Intel Corporation. Disponível em <www.intel.com>, acesso 12.jun.2014.
- IoT-A, (2014). Internet of Things Architecture of the European Lighthouse Integrated Project. Disponível em <wew.iot-a.eu>. Acesso em 10 jul. 2014
- KLEINROCK, L. (2010). An early history of the internet [History of Communications]. *Communications Magazine, IEEE*, 48(8), 26-36.
- KORTUEM, G., Kawsar, F., Fitton, D., & Sundramoorthy, V. (2010). Smart objects as building blocks for the internet of things. *Internet Computing, IEEE*, 14(1), 44-51.
- KUROSE, J. F., & Ross, K. W., *Computer Networking: A top-down approach featuring the Internet* (vol. 2), reading: Addison-Wesley, 2001.
- KUSHALNAGAR, N., Montenegro, G., & Schumacher, C. (2007). IPv6 over low-power wireless personal area networks (6LoWPANs): overview, assumptions, problem statement, and goals. *RFC4919, August, 10*, disponível em <tools.ietf.org/html/rfc4919>, acesso 9.jul.2014.
- LEAVITT, N. (2010). Will NoSQL databases live up to their promise? *Computer*, 43 (2), 12-14.
- LinkSmart Wiki, (2014). *Middleware for networked embedded systems*, disponível em <www.sourceforge.net/p/linksmart/wiki/>, acessado em 10.7.2014.

- LYYTINEN, K., & Yoo, Y., *Ubiquitous Computing*, in Communications of the ACM, 45 (12), 6, 2002.
- McGREW, D., & Bailey, D. (2012). AES-CCM Cipher Suites for Transport Layer Security (TLS). RFC 6655, july.
- MELL, P., & Grance, T. (2011). The NIST definition of cloud computing.
- MERK, L., Nicklous, M. S., & Stober, T. (2001). Pervasive computing handbook (Vol. 3, pp. 53-62). Heidelberg: Springer.
- MILLER, B.A., Nixon, T., Tai, C. and Wood, M.D. (2001) ‘Home networking with universal plug and play’, *IEEE Communications Magazine*, December, Vol. 39, No. 12, pp.104–109, DOI: 10.1109/35.968819.
- MIORANDI, D., Sicari, S., De Pellegrini, F., & Chlamtac, I. (2012). Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7), 1497-1516.
- MOORE, G. E. (1965). Cramming more components onto integrated circuits.
- NPD DisplaySearch, 2014. Disponível em <<http://www.displaysearch.com/>>. Acessado em 11 jun. 2014.
- OLIVEIRA, A. (2011). Introdução a Web Semântica, Ontologia e Máquinas de Busca. *REVISTA TECNOLOGIAS EM PROJEÇÃO*, Vol 2, n1.
- PHP (2014), Disponível em <php.net>, acesso 30.jul.2014.
- RaspberryPi (2014), Disponível em <www.raspberrypi.org>. Acesso em 11 jun 2014.
- SHELBY, Z., Hartke, K., Bormann, C., & Frank, B. (2012). Constrained Application Protocol (CoAP), draft-ietf-core-coap-13. *Orlando: The Internet Engineering Task Force–IETF, Dec.* RFC-7252. Disponível em <tools.ietf.org/html/rfc7252>. Acesso em 9 jul. 2014
- TAN, L., & Wang, N., *Future internet: The internet of things*, in Advanced Computer Theory and Engineering (ICACTE), 3rd International Conference, vol. 5, pp. V5-376, IEEE, ago.2010.
- Universal Plug and Play Forum (2000) *Understanding Universal Plug and Play*. Disponível em <http://www.upnp.org/download/UPNP_understandingUPNP.doc>, acesso 9.jul.2014.
- WEBER, R. H. (2010). Internet of Things—New security and privacy challenges. *Computer Law & Security Review*, 26 (1), 23-30.
- WEISER, M., *The Computer for the 21st Century*, Scientific American, vol. 265, no. 3, setembro, pp. 94-100, 1991.

ZigBee Alliance, disponível em <www.zigbee.org>, acesso 7.jul.2014.

ANEXOS

ANEXO 1 – CÓDIGO ARDUINO PARA FAZER UM LED PISCAR

```
/*
Este código faz com que um LED conectado no pino 13 da placa Arduino ligue e desligue a
cada um segundo, repetidamente.
*/
// Declara o pino em que o LED está conectado
int led = 13;

// A rotina SETUP é executada quando a placa é ligada
void setup() {
    // define que o pino do LED é de saída
    pinMode(led, OUTPUT);
}

// A rotina loop se repete infinitamente até que a placa seja desligada
void loop() {
    // Altera o estado do pino do LED para alta voltagem, acendendo o LED
    digitalWrite(led, HIGH);
    // A placa pausa sua execução por um segundo
    delay(1000);
    // Altera o estado do pino do LED para baixa voltagem, apagando o LED
    digitalWrite(led, LOW);
    // A placa pausa sua execução por um segundo
    delay(1000);
}
```

ANEXO 2 - XML UPNP DE DESCRIÇÃO DE DISPOSITIVOS

```
<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0" configId="número de configuração">
<specVersion>
    <major>1</major>
    <minor>1</minor>
</specVersion>
<device>
    <deviceType>urn:schemas-upnp-org:device:deviceType:v</deviceType>
    <friendlyName>nome amigável curto</friendlyName>
    <manufacturer>nome do fabricante</manufacturer>
    <manufacturerURL>URL para o site do fabricante</manufacturerURL>
    <modelDescription>nome amigável longo</modelDescription>
    <modelName>nome do modelo</modelName>
    <modelNumber>número do modelo</modelNumber>
    <modelURL>URL para o site do modelo</modelURL>
    <serialNumber>número serial do fabricante</serialNumber>
    <UDN>uuid:UUID</UDN>
    <UPC>Universal Product Code</UPC>
    <iconList>
        <icon>
            <mimetype>image/formato</mimetype>
            <width>pixels horizontais</width>
            <height>pixels verticais</height>
            <depth>intensidade da cor</depth>
            <url>URL para o ícone</url>
        </icon>
        <!-- Espaço para declaração de outros ícones, se houverem -->
    </iconList>
    <serviceList>
        <service>
            <serviceType>urn:schemas-upnp-org:service:serviceType:v</serviceType>
            <serviceId>urn:upnp-org:serviceId:serviceID</serviceId>
            <SCPDURL>URL para descrição de serviço</SCPDURL>
            <controlURL>URL para controle</controlURL>
            <eventSubURL>URL para assinatura</eventSubURL>
        </service>
        <!-- Espaço para declaração de outros serviços definidos por uma comissão de trabalho do fórum UPnP, se houverem -->
        <!-- Espaço para declaração de outros serviços definidos pelo fabricante, se houverem -->
    </serviceList>
    <deviceList>
        <!-- Espaço para declaração de dispositivos embutidos definidos por uma comissão de trabalho do fórum UPnP, se houverem -->
        <!-- Espaço para declaração de dispositivos embutidos definidos pelo fabricante, se houverem -->
    </deviceList>
    <presentationURL>URL para apresentação</presentationURL>
</device>
</root>
```

ANEXO 3 - XML UPNP DE DESCRIÇÃO DE SERVIÇOS

```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0" xmlns:dt1="urn:domain-
name:more-datatypes" <!-- Declarations for other namespaces added by UPnP Forum
working committee (if any) go here --> <!-- The value of the attribute must remain as
defined by the UPnP Forum working committee.--> xmlns:dt2="urn:domain-
name:vendor-datatypes" <!-- Declarations for other namespaces added by UPnP vendor
(if any) go here --> <!-- Vendors must change the URN's domain-name to a Vendor
Domain Name --> <!-- Vendors must change vendor-datatypes to reference a vendor-
defined namespace --> configId="configuration number">
<specVersion> <major>1</major> <minor>1</minor> </specVersion>
<actionList>
    <action>
        <name>actionName</name>
        <argumentList>
            <argument>
                <name>argumentNameIn1</name>
                <direction>in</direction>
                <relatedStateVariable>stateVariableName</relatedStateVariable>
            </argument>
            <!-- Declarations for other IN arguments defined by UPnP Forum working
Committee (if any) go here -->
            <argument>
                <name>argumentNameOut1</name>
                <direction>out</direction>
                <retval/>
                <relatedStateVariable>stateVariableName</relatedStateVariable>
            </argument>
            <!-- Declarations for other OUT arguments defined by UPnP Forum
working committee (if any) go here -->
        </argumentList>
    </action>
    <!-- Declarations for other actions defined by UPnP Forum working committee (if any)go
here -->
    <!-- Declarations for other actions added by UPnP vendor (if any) go here -->
</actionList>
<serviceStateTable>
    <stateVariable sendEvents="yes" | "no" multicast="yes" | "no">
        <name>variableName</name>
        <dataType>basic data type</dataType>
        <defaultValue>default value</defaultValue>
        <!--Should have either allowedValueRange or allowed value list or none -->
        <allowedValueRange>
            <minimum>minimum value</minimum>
            <maximum>maximum value</maximum>
            <step>increment value</step>
        </allowedValueRange>
```

```

<allowedValueList>
    <allowedValue>enumerated value</allowedValue>
    <!-- Other allowed values defined by UPnP Forum working committee (if any) go here -->
    <!-- Other allowed values defined by vendor (if any) go here -->
    >
</allowedValueList>
<stateVariable>
    <!-- Declarations for other state variables defined by UPnP Forum working committee (if any) go here -->
    <!-- Declarations for other state variables added by UPnP vendor (if any) go here -->
</serviceStateTable>
</scpd>

```

ANEXO 4 – JSONS TROCADOS ENTRE AS CAMADAS DE SERVIÇOS E DE CONTROLE

```
//Notificação da camada de serviços para a camada de controle sobre uma nova requisição de ação
{
    dispositivo: id_dispositivo,
    serviço: id_serviço,
    ação: "nome da ação",
    argumentos_de_entrada: {
        nome_do_argumento_1: valor_do_argumento,
        //...
        nome_do_argumento_n: valor_do_argumento_n
    }
}
```

```
//Resposta da camada de controle para a camada de comunicação sobre a requisição de ação
{
    código_status: "CÓDIGO", //SUCESSO ou ERRO
    mensagem_status: "mensagem", //mensagem do ocorrido
    argumentos_de_saida: {
        nome_do_argumento_1: valor_do_argumento,
        //...
        nome_do_argumento_n: valor_do_argumento_n
    }
}
```

```
//Notificação da camada de controle para a camada de serviços sobre novo dispositivo disponível
{
    status: "online",
    dispositivo: id_dispositivo,
}
```

```
//Notificação da camada de controle para a camada de serviços sobre dispositivo indisponível
{
    status: "offline",
    dispositivo: id_dispositivo,
}
```

```
//Notificação da camada de controle para a camada de serviços sobre alteração de
variável de estado
{
    status: "change",
    dispositivo: id_dispositivo,
    serviço: id_serviço,
    variavel_de_estado: {
        id: id_variavel_de_estado,
        novo_valor: novo_valor,
    }
}
```

ANEXO 5 – JSONS TROCADOS ENTRE AS CAMADAS DE COMUNICAÇÃO E DE CONTROLE

```
//Pedido da camada de controle para a camada de comunicação para envio de troca de valor de pino
{
    endereco_zibee: endereco_escravo_destino,
    pino: numero_do_pino,
    forma_de_operacao: “forma de operação”,
    taxa_de_transmissao: taxa ,
    tipo_do_dado: “tipo”,
    dado: dado
}
```

```
//Resposta da camada de comunicação para a camada de controle sobre envio de troca de valor de pino
{
    codigo_status: “CODIGO”,      //SUCESSO ou ERRO
    mensagem_status: “mensagem”, //mensagem do ocorrido
}
```

```
//Notificação de novo valor lido em pino da camada de comunicação para a camada de controle
{
    numero_de_ordem: identificador_da_mensagem,
    endereco_zibee: endereco_do_controlador_escravo_emissor,
    pino: numero_do_pino_onde_valor_foi_lido,
    dado: dado
}
```