UNIVERSIDADE FEDERAL DE PERNAMBUCO

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Aura: Um middleware semântico para a Internet das Coisas

Aluno: Vinícius Matos da Silveira Fraga {vmsf@cin.ufpe.br}

Orientador: Djamel F. H. Sadok {jamel@cin.ufpe.br}

Recife, janeiro de 2016

Aura: Um middleware semântico para a Internet das Coisas

Aluno: Vinícius Matos da Silveira Fraga {vmsf@cin.ufpe.br}

Orientador: Djamel F. H. Sadok {jamel@cin.ufpe.br}

Resumo

Internet das Coisas significa ter objetos do dia-a-dia conectados à internet, enviando dados e podendo ser controlados. As coisas podem ser automóveis com sensores, geladeiras, tomadas, lâmpadas, implantes em animais, portas de garagem etc. Todos esses dispositivos são diferentes entre si, e apesar do objetivo ser que trabalhem em conjunto, não é assim que tem funcionado. É difícil integrá-los porque as tecnologias atuais exigem conhecimento muito específico e o desenvolvimento é muito trabalhoso. Para mitigar essas dificuldades de integração e desenvolvimento, este trabalho apresenta o Aura Middleware, uma alternativa para orquestrar dispositivos diferentes de forma simples. O Aura usa uma abordagem semântica baseada em ontologias, onde o único trabalho para integrar uma tecnologia é escrever um *gateway* que entenda suas ontologias, não importa qual a linguagem de programação ou *framework* utilizado. Além de integrar dispositivos, oferece aos desenvolvedores funcionalidades de monitoramento e controle da rede através de uma API REST.

Agradecimentos

Pelo apoio e orientação durante o desenvolvimento deste trabalho, além dos anos de orientação em pesquisa, agradeço ao prof. Djamel Sadok e à profª Judith Kelner. Agradeço também a Lucas Gomes, pelas diversas dicas de Python e arquiteturas SOA. A Walter Sobral e Heitor fontes, pela participação no projeto SARMIoT, que inspirou a abordagem semântica empregada no Aura Middleware. A Daniela Falcone, pelo apoio nas horas difíceis, pela revisão textual e pelo trabalho de design gráfico empregado neste documento. E a minha mãe, Moema Silveira, que sempre fez tudo que estava a seu alcance para me dar educação e garantir que eu chegasse até aqui.

Sumário

1. Introdução 5

2. Trabalhos relacionados 7

2.1 LinkSmart 7

2.2 WSNManager 9

2.3 The Internet of Things Database (IOTDB) 12

3. Aura Middleware 15

3.1 Arquitetura 17

3.2 Gateways 18

3.3 Aura Broker 19

3.4 Aura Worker 20

3.4.1 Semantic Manager 21

3.4.2 Device Manager 24

3.4.3 Task Manager 24

3.4.4 Storage Manager 25

3.5 API REST 26

4. Experimentos 29

4.1 Cenário 1: Entrada de Novos Dispositivos 29

4.2 Cenário 2: Requisições para a API 30

4.3 Métricas Observadas 30

4.4 Resultados e Discussão 31

5. Conclusão e Trabalhos Futuros 33

6. Bibliografia 34

1. Introdução

O que é a Internet das Coisas? Acredita-se que a primeira vez em que se falou sobre a “Internet das Coisas” foi no ano de 1999, durante uma apresentação do britânico Kevin Ashton na Procter & Gamble (P&G) [1]. Na visão de Kevin, os computadores de hoje dependem muito dos seres humanos para gerar dados e informação, e se as “coisas” gerassem dados, a humanidade teria diversas vantagens. Em outras palavras, é necessário que pessoas digitem, cadastrem, usem código de barras, acessem páginas na web, escrevam em seus blogs e redes sociais para gerar dados e informação. Porém, se os objetos do dia-a-dia estivessem de alguma forma conectados às redes de computadores, à internet, poderíamos contabilizá-los, extrair informação, evitar perdas, saber quando repará-los, controlá-los a distância, programar comportamentos e muito mais. Assim surgia a ideia da Internet das Coisas (IoT - do inglês, *Internet of Things*).

De acordo com a União Internacional de Telecomunicações (ITU - do inglês, *International Telecommunication Union*) [2], a IoT é uma infraestrutura global para a informação da sociedade, disponibilizando serviços avançados através da interconexão física e virtual entre coisas, baseada nas tecnologias de comunicação existentes e em desenvolvimento. Nesse sentido, as coisas podem ser automóveis com sensores, geladeiras, tomadas inteligentes, lâmpadas, televisores, implantes em animais, monitores cardíacos, portas de garagem etc. Esses são os conceitos que este trabalho se refere ao mencionar o termo IoT. Deste ponto em diante, sempre que o texto mencionar um dispositivo, entenda-o por uma coisa como as listadas acima com as capacidades mínimas para estabelecer comunicação com outros dispositivos ou computadores.

Em 2009 foram contabilizados 0,9 bilhões de dispositivos conectados à internet [3]. Segundo executivos da Cisco [4], em 2012 havia 8,7 bilhões de dispositivos conectados à internet. Na época, a estimativa para 2015 era cerca de 15 bilhões de dispositivos, chegando a 40 bilhões em 2020. Esses dispositivos variam de características físicas, funcionalidades e fabricantes. Tal diversidade faz com que a IoT seja encarada de pontos de vista diferentes [5], sobre como ela é e o que ela deve se tornar. Isso impacta o desenvolvimento da tecnologia. O resultado é que existem diferentes soluções para diferentes problemas, mas que não trabalham bem em conjunto, exigindo trabalho de especialistas para integrar dispositivos de fabricantes diferentes. Esse tipo de problema já é conhecido da área de redes de sensores sem fio (WSNs - do inglês, *Wireless Sensor Networks*) [6], o que significa que algumas soluções dessa área servem de inspiração para a IoT, como veremos a seguir.

Uma abordagem para lidar com essa complexidade é utilizar um *software* intermediário entre os diferentes sistemas. Esse tipo de *software* é chamado de *middleware*. Pode-se dizer que um *middleware* funciona como uma "cola" entre sistemas [7]. Quando se trata da IoT, problemas incomuns a sistemas distribuídos tradicionais surgem: existem dispositivos com pouca bateria, com pouco processamento, pouca memória, escalabilidade (bilhões de dispositivos) além da heterogeneidade entre os dispositivos da rede, como supracitado. Maiores detalhes sobre essas problemáticas podem ser encontrados em [8]. Além disso, a IoT está diretamente relacionada com *Big Data* [9][10][11], o que adiciona ao *middleware* a preocupação com o volume e velocidade de dados gerados pela rede.

Levantadas as problemáticas de heterogeneidade, escalabilidade e *Big Data*, embasadas por um levantamento do estado da arte da IoT quanto a essas questões, este trabalho propõe um *middlewar*e para facilitar o desenvolvimento de aplicações reduzindo ao máximo o trabalho de integração entre tecnologias. Para mitigar essa barreira da heterogeneidade, o *middleware* proposto usará uma estratégia semântica, baseada em ontologias [12] (abordagem análoga às utilizadas em [6][13][14]). Ao final, os resultados apontam a viabilidade computacional de uma abordagem semântica para a IoT, assim como as vantagens de desenvolvimento desse tipo de estratégia. O *middleware* apresentado neste trabalho foi denominado Aura, palavra latina que significa brisa, vento suave. Seu nome foi escolhido para simbolizar a suavidade que proporciona para integrar diferentes tecnologias e desenvolver aplicações de IoT.

Este documento está organizado da seguinte forma: o capítulo 2 trata dos trabalhos relacionados encontrados na revisão de literatura. O capítulo 3 apresenta o Aura Middleware, com subseções sobre sua arquitetura e as tecnologias utilizadas pelos componentes. No capítulo 4 são apresentados os experimentos realizados, assim como uma discussão sobre os mesmos. O capítulo 5 apresenta as conclusões e propõe alguns trabalhos futuros. O capítulo 6 contém a bibliografia.

2. Trabalhos relacionados

É possível encontrar propostas de padronização da IoT em termos de semântica, comunicação, roteamento, enfim, equivalências para praticamente todas as camadas da pilha de protocolos utilizadas na internet convencional. Não obstante, a diversidade de propostas vem resultando na não adoção de um padrão.

Por outro lado, alguns resultados de trabalhos nesse sentido são valiosos para a construção de um sistema mais amplo e menos acoplado a uma tecnologia específica. A *Internet Engineering Task Force* (IETF) propõe padrões de rede ideais para dispositivos de baixo poder computacional, como os de redes de sensores sem fio, por considerarem limitações de energia e processamento [15]. Ao passo que o *World Wide Web Consortium* (W3C) [16] e o *HyperCat Consortium* [17] propõem ontologias para descrever a IoT em diferentes níveis de detalhamento, sendo particularmente úteis ao Aura como inspiração para a semântica do middleware. Já o Open Geospatial Consortium (OGC) [18] propõe uma estrutura baseada em APIs para acessar os recursos dos dispositivos na IoT, abordagem inspiradora para a interface com dispositivos que podem atuar como clientes, tais como smartphones e computadores. Por último, a *AllSeen Alliance* propõe um *framework* que organiza a IoT em aplicações e roteadores dentro de uma rede local [19]. O *framework*, porém, suporta poucas tecnologias de comunicação, limitado a Wi-Fi, ethernet, serial e *Power Line Communication* (PLC), não permitindo o uso de dispositivos simplórios que usem padrões como Zigbee e afins.

Este trabalho não visa propor um novo padrão, mas utiliza alguns pontos das propostas atuais como alicerce para um *middleware* que contará com padronização apenas no tipo de mensagem que troca com os clientes e dispositivos. Maiores detalhes serão definidos no capítulo 3.

As subseções a seguir descrevem os *middlewares* encontrados na literatura que abordam problemas de heterogeneidade de dispositivos tanto na IoT quanto em redes de sensores, destacando algumas propostas de semântica (i.e., ontologias) para IoT, e que foram as principais referências para o middleware Aura, considerando tanto as boas como as más estratégias e também os modelos de arquitetura.

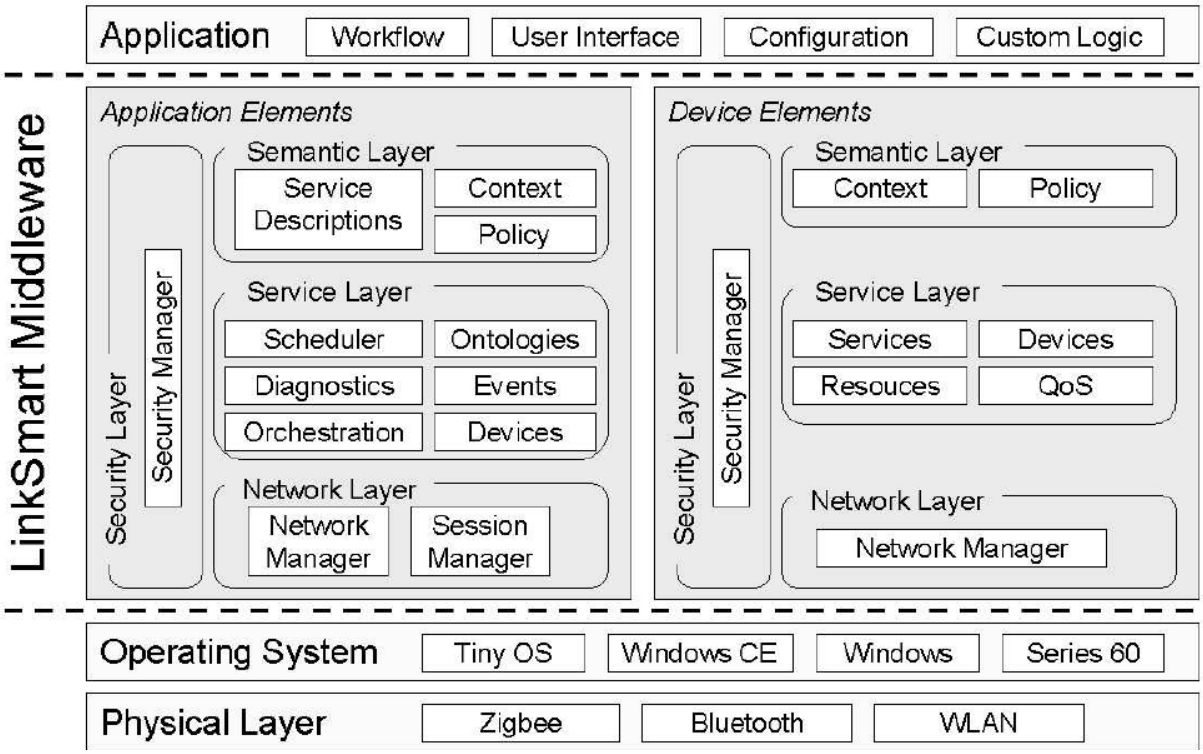
2.1 LinkSmart

O LinkSmart é um *middleware* semântico e orientado a serviços [20], escrito em Java, oriundo de um projeto de pesquisa europeu chamado HYDRA [21]. Ele é descrito na literatura como uma plataforma de *middleware* não apenas para a IoT, envolvendo também os conceitos de Internet das Coisas e Serviços (IoTS - do inglês, *Internet of Things and Services*) [22] e Internet das Pessoas, Coisas e Serviços (IoPTS - do inglês, *Internet of People, Things and Services*) [23].

A arquitetura desse *middleware* é orientada a modelos semânticos [24], de modo que o cerne de seu funcionamento é definir dispositivos e serviços de acordo com seus modelos e a partir disso gerar o código relativo às funcionalidades básicas de cada serviço. Mais precisamente, sua tática de abstração de heterogeneidade é interpretar todos os dispositivos como serviços *web*. Para adicionar um dispositivo a uma rede intermediada pelo LinkSmart, é necessário

descrevê-lo de acordo com o modelo semântico usado no *middleware,* para que então ele seja disponibilizado aos clientes como um serviço que publica dados e/ou realiza ações. Sua arquitetura está ilustrada[1] na Figura 1.

Figura 1: Arquitetura do LinkSmart.



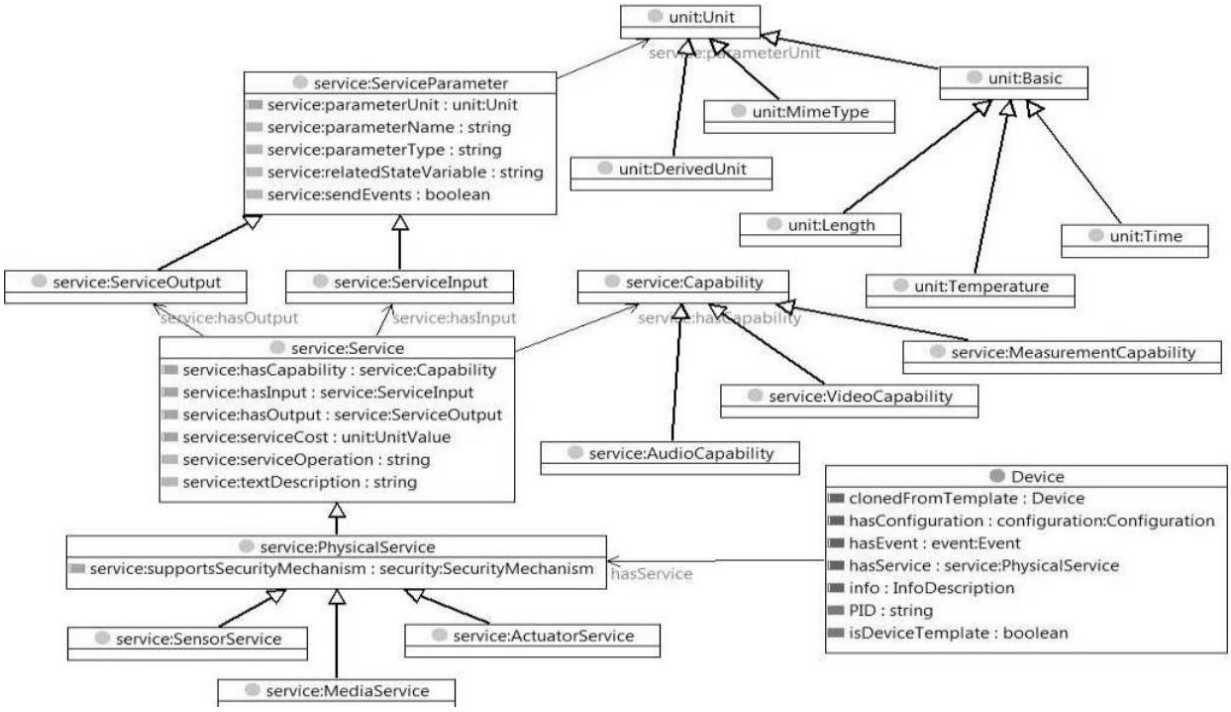
Fonte: P. Kostelník, M. Sarnovsky, K. Furdík. “THE SEMANTIC MIDDLEWARE FOR NETWORKED EMBEDDED SYSTEMS APPLIED IN THE INTERNET OF THINGS AND SERVICES DOMAIN”

A semântica é explorada basicamente de duas formas no LinkSmart: a primeira é durante o desenvolvimento, quando ela poupa esforços do desenvolvedor uma vez que os descritores semânticos geram código Java, como mencionado acima, para utilizar um serviço, buscar um dispositivo ou trabalhar com os dados da rede. A segunda forma é em tempo de execução, facilitando a descoberta de dispositivos e serviços.

Para prover acesso de forma unificada às ontologias que usa, o LinkSmart possui um gerenciador de ontologias. A Figura 2 mostra a ontologia de serviços.

1: Algumas figuras deste documento contém textos em inglês por serem referentes a trabalhados desenvolvidos nesse idioma. O próprio Aura Middleware foi desenvolvido em inglês por razões de facilidade de disseminação futura do projeto.

Figura 2: Ontologia de serviços do LinkSmart.



Fonte: P. Kostelník, M. Sarnovsky, K. Furdík. “THE SEMANTIC MIDDLEWARE FOR NETWORKED EMBEDDED SYSTEMS APPLIED IN THE INTERNET OF THINGS AND SERVICES DOMAIN”.

A priori, o LinkSmart seria uma boa opção para superar a heterogeneidade e ainda inferir conhecimento dos dados obtidos com a IoT. Porém, analisando sua arquitetura minuciosamente, nota-se que ela é complexa, conta com muitos módulos internos e camadas a serem assimiladas para que os desenvolvedores possam utilizar a ferramenta. Além disso, restringir o uso do *middleware* a determinadas tecnologias vai na contramão da proposta de facilitar a integração de dispositivos e acaba se tornando um funil para os desenvolvedores - neste caso, programação em Java e manipulação de arquivos XML.

2.2 WSNManager

O WSNManager é um *middleware* sensível a contexto para redes de sensores heterogêneas [25]. Apesar de não se tratar de um *middleware* diretamente para IoT, o fato dele focar na heterogeneidade de redes de sensores o torna relevante para este trabalho, visto que a IoT engloba as restrições encontradas em WSNs.

Mais precisamente, o WSNManager atua no monitoramento e controle de redes de sensores. Tal monitoramento consiste em coletar medições de diversos sensores e o controle se traduz em enviar comandos para os nós da rede, como ligar ou desligar algum eletrônico conectado.

Como dispositivos em redes de sensores comumente não possuem muito poder computacional, o WSNManager não é implementado a nível de rede, um outro dispositivo, como um computador, precisa hospedá-lo. Essa é uma característica interessante diante não só das restrições computacionais, mas também da heterogeneidade de dispositivos. Se o *middleware* fosse implementado como parte de uma rede, ele precisaria ser reimplementado para tecnologias diferentes.

No que concerne ao volume de dados, o WSNManager aplica uma estratégia de escalabilidade horizontal: seus principais componentes compõem um *worker* sem estado, de modo que vários *workers* podem ser instanciados por demanda para lidar com redes de muitos nós e dar vazão ao recebimento de grandes volumes de dados, distribuindo a responsabilidade de salvá-los em um banco de dados.

A arquitetura do WSNManager é composta de *workers*, de um *Client Broker*, que faz o intermédio entre os *workers* e aplicações cliente, além de um *Gateway Broker,* que intermedia os *gateways* das diferentes redes com o *middleware*. Sendo assim, cada rede precisa ter um *gateway* próprio.

A Figura 3 apresenta a arquitetura de um sistema com o uso do WSNManager e a Figura 4 ilustra a arquitetura do WSNManager.

Figura 3: Arquitetura de um sistema usando o WSNManager.

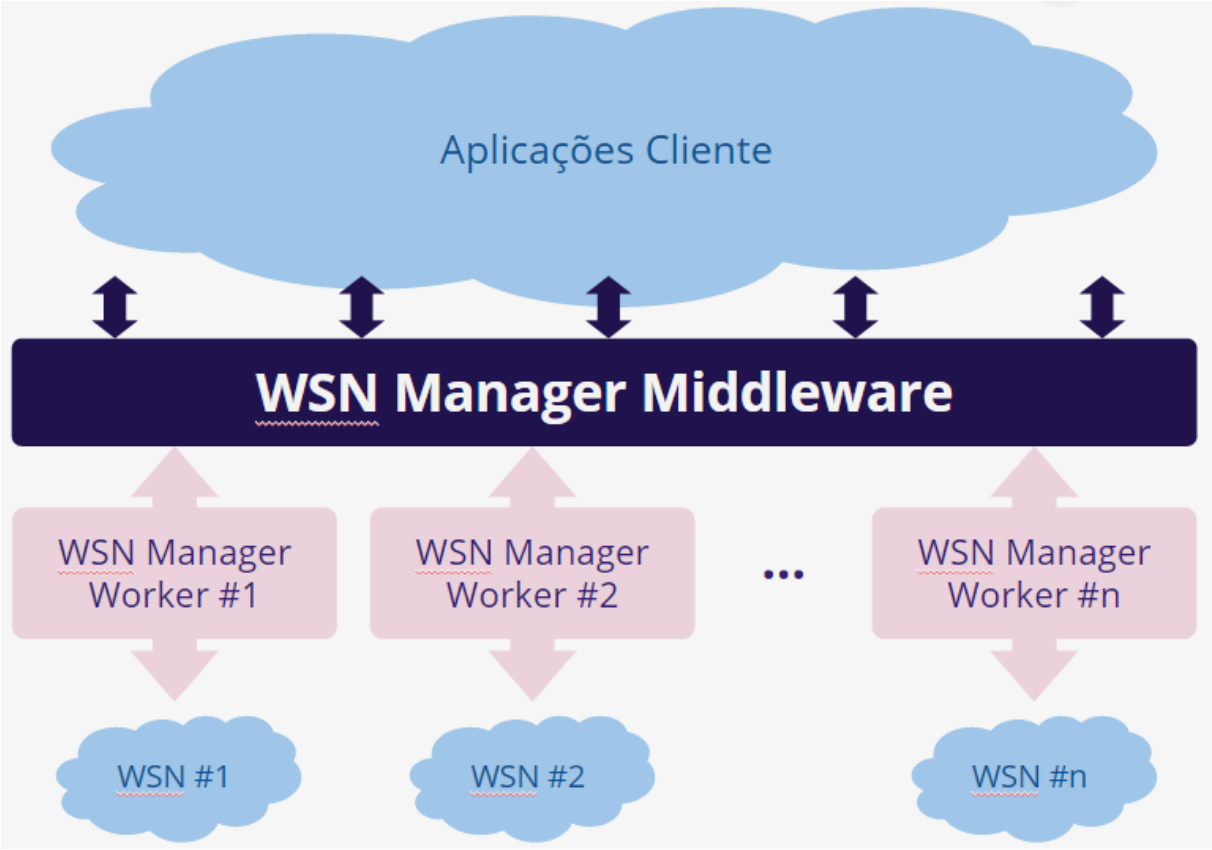
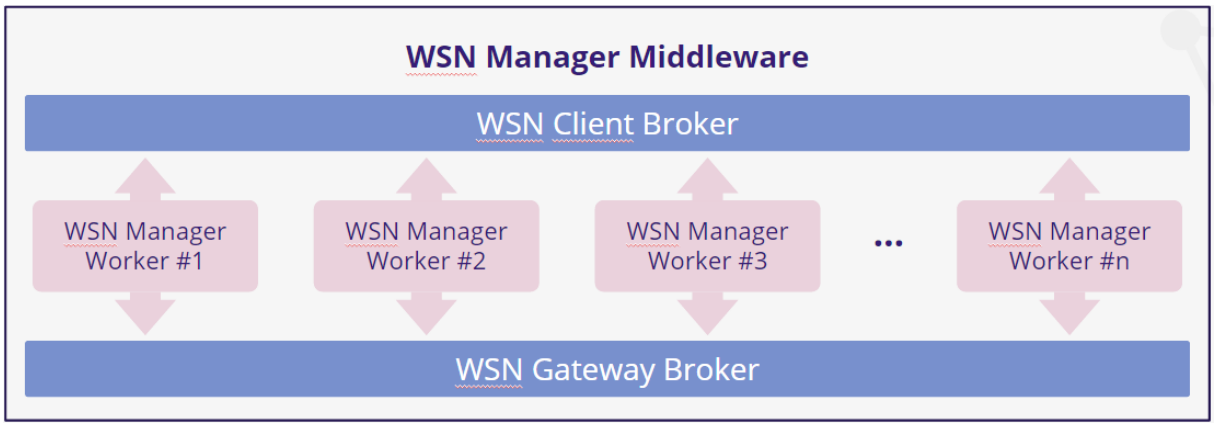


Figura 4: Arquitetura do WSNManager.



O WSNManager pode ser programado para realizar determinadas ações caso um conjunto de condições seja satisfeito, que pode ser denominado como ciência a contexto. Por exemplo, mandar um sensor medidor de corrente, cortar a passagem de corrente para um aparelho de ar-condicionado em uma sala se a temperatura ficar abaixo de 20ºC. Ou seja, na prática o *middleware* desligaria o ar-condicionado caso a temperatura na sala estivesse a menos de 20ºC. Nesse caso, ter temperatura menor que 20ºC seria a condição de contexto para disparar uma ação em um dos nós da rede gerenciada pelo *middleware*.

Diferentemente do que é proposto neste trabalho, o WSNManager não utiliza ferramentas de semântica, como ontologias, para implementar sua ciência a contexto. Ao invés disso, usa uma abordagem menos custosa computacionalmente, porém mais complexa de implementar e expandir para cenários diversos. Além disso, o WSNManager é limitado quanto à inferência de conhecimento a partir dos dados que já possui. Finalmente, a estratégia utilizada em sua ciência de contexto é dividida em partes: **aquisição**, **modelagem** e **decisão**.

Durante a fase de aquisição, os sensores enviam dados ao WSNManager através de seus respectivos *gateways*. Porém, antes de repassar os dados ao *Gateway Broker*, o *gateway* enriquece os dados com metadados, e.g., tipo de sensor utilizado, intervalo de medições etc.

Uma vez recebido o contexto de um gateway, passa-se à fase de modelagem, onde o middleware o descreve em um *JavaScript Object Notation* (JSON), acrescentando o tipo de política aplicada ao contexto.

Por último, na fase de decisão, as políticas são representadas por regras do formato *IF-THIS-THAN-THAT*, que se traduzem em expressões *booleanas*. De acordo com a avaliação dessas expressões o *middleware* pode realizar ações diferentes.

Fazendo uma análise geral, observa-se que o WSNManager tem em sua arquitetura pontos relevantes para a IoT, que permitem que ele seja utilizado nessa área. Porém, como ele não foi projetado com esse foco, existem pontos sensíveis quanto à complexidade de expandir sua inferência de conhecimento, porque seus mecanismos de ciência a contexto são limitados nesse sentido, e essa capacidade seria importante para a usabilidade prevista para a IoT. Além disso, sua abordagem para lidar com volumes maiores de dados poderia ser mais explorada, acrescentado bancos de dados baseados em memória e estratégias de fusão e agregação de dados, por exemplo. Enfim, seus aspectos mais interessantes são a

escalabilidade horizontal e a independência de tecnologia nas pontas da rede, ambos baseados na arquitetura modular e na comunicação entre *brokers* e *gateways*.

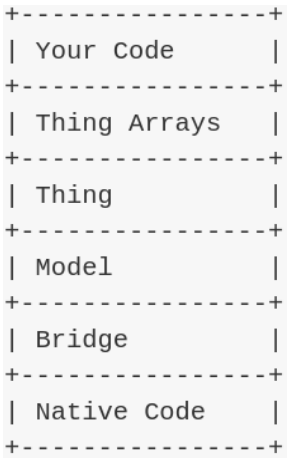
2.3 The Internet of Things Database (IOTDB)

IOTDB é uma biblioteca baseada no *framework* Node.js [26], feito na linguagem JavaScript. Apesar de não ser um *middleware*, é uma biblioteca robusta que também usa descrições semânticas para superar a heterogeneidade de tecnologias [27].

Como ferramenta de gerenciamento da biblioteca foi desenvolvida uma aplicação chamada HomeStar [28], que possui interface web e permite configurar os módulos da IOTDB, interagir com bancos de dados, além de incluir alguns exemplos de uso e prover uma *Application Programming Interface* (API) *Representational State Transfer* (REST). O retorno das requisições HTTP da API REST utiliza o formato *JavaScript Object Notation* (JSON), assim como a representação dos dispositivos dentro na biblioteca.

Naturalmente, sua arquitetura é bastante distinta dos *middlewares* supracitados, uma vez que a IOTDB não é um *middleware*. A biblioteca utiliza um padrão de engenharia de software conhecido como *bridge*, que oferece uma forma abstrata de acesso a determinada funcionalidade do sistema, tornando-a independente de sua implementação. Usando uma analogia simples, uma *bridge* funciona como um *driver* de dispositivo para determinada tecnologia. Para adicionar um dispositivo é necessário, então, escrever seu "*driver*". A Figura 5 mostra a arquitetura básica de um sistema implementado em Node.js que utiliza a IOTDB.

Figura 5: Arquitetura comum de um software que utilize IOTDB.



Fonte: https://github.com/dpjanes/iotdb-homestar

Caso o desenvolvedor deseje usar a IOTDB sem usar Node.js em seu projeto, a biblioteca prevê o uso de mecanismos chamados *transporters*, que cumprem o papel de "carregar" os JSONs. Um *transporter* pode ser, por exemplo, um módulo que utiliza um protocolo de comunicação específico. Observa-se na Figura 6 que, nesse caso de uso, aumentam as camadas na arquitetura do sistema final para usar a IOTDB.

Figura 6: Arquitetura de um software que usa IOTDB através de seus Transporters.

Fonte: https://github.com/dpjanes/iotdb-homestar

A característica mais interessante da IOTDB para a construção da proposta deste trabalho, é a simplicidade de suas descrições semânticas. Suas ontologias se pautam diretamente pelas funcionalidades dos dispositivos: se determinado dispositivo pode ser ligado e desligado, é nesses termos que a ontologia o descreve. Se é uma lâmpada que pode mudar de cor, é isso que consta na ontologia. Ou seja, o guia para se descrever dispositivos foi uma pergunta do tipo “o que esse dispositivo faz?”. Essa estratégia simples resultou em ontologias com propriedades intuitivas, facilitando o desenvolvimento a partir delas. Um exemplo pode ser conferido na Figura 7.

Figura 7: Exemplos de uso da IOTDB.

Fonte: https://iotdb.org/

Ademais, em termos de bancos de dados, a IOTDB provê o uso de algumas tecnologias, como o sistema de arquivos do sistema operacional hospedeiro, bancos de dados na nuvem e

aproveita facilidades de integração de Node.js com diversos bancos locais. Contudo, não há especificação quanto a grandes volumes de dados, assim como suporte a escalabilidade horizontal em caso de grande quantidade de dispositivos conectados.

Sumarizando as informações do estado da arte que foram mais relevantes para o desenvolvimento deste trabalho, observa-se que:

1. Utilizar semântica traz vantagens de facilidade de integração e desenvolvimento e é uma técnica que já foi utilizada na área da IoT;
2. Trabalhar com a descrição dos dispositivos da rede dentro do próprio middleware torna a solução mais acoplada e exige conhecimento de tecnologias muito específicas para expandir o trabalho para novas tecnologias, portanto é uma prática a ser evitada;
3. Aplicar técnicas de raciocínio e ciência a contexto muito simplórias torna custoso o trabalho de expansão para novas tecnologias, pois envolve desenvolvimento direto de mais componentes de software;
4. Uma arquitetura baseada em serviços permite escalabilidade horizontal, adequada para cenários com grandes volumes de dados como a IoT;
5. A simplicidade da integração e interação com dispositivos torna o desenvolvimento de aplicações de IoT uma tarefa fácil e mais atraente aos desenvolvedores.

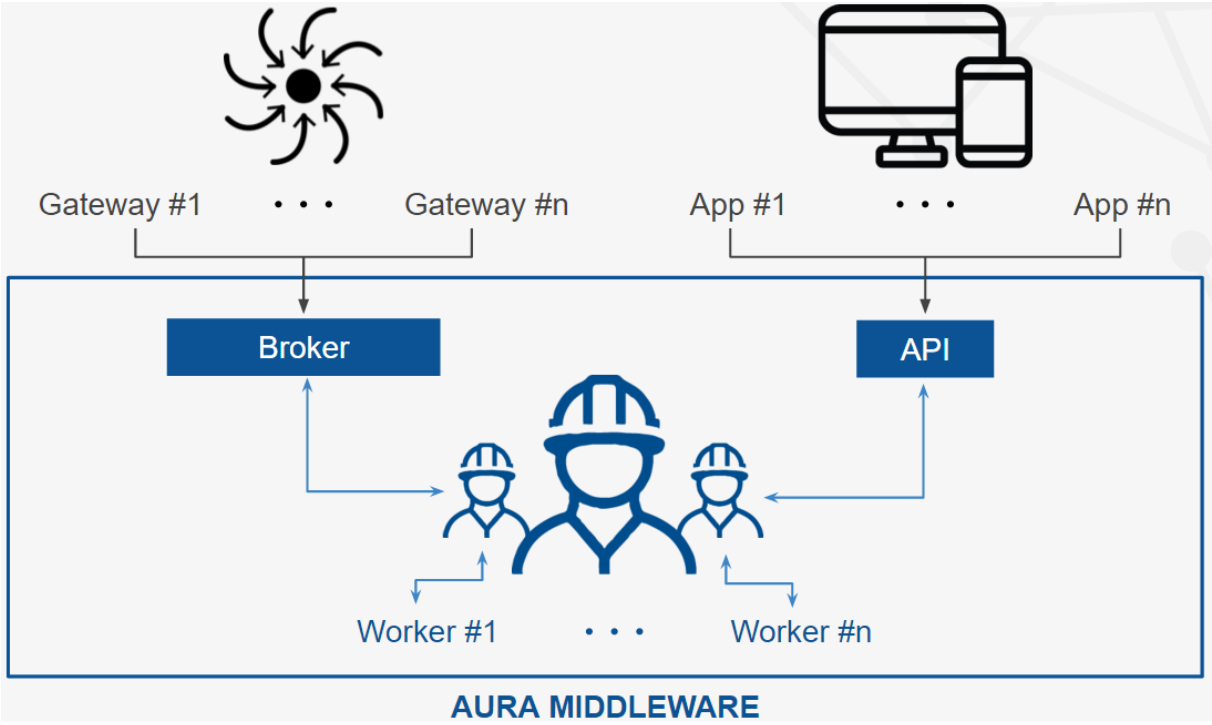
3. Aura Middleware

Dada a problemática da heterogeneidade de comunicação entre as tecnologias que constituem a IoT e após o destaque de alguns trabalhos com características promissoras encontrados no estado da arte, será apresentada a proposta deste trabalho: o Aura Middleware. Um *middleware* que objetiva orquestrar um conjunto de dispositivos de diferentes tecnologias, aproveitando os seus recursos e ao mesmo tempo facilitando o desenvolvimento de aplicações voltadas para monitoramento e controle na IoT.

O pilar da estratégia de abstração de heterogeneidade do Aura é seu caráter semântico: do ponto de vista das "coisas" que estão na IoT, sendo necessário para elas trabalharem em conjunto que os *gateways* sejam capazes de enviar e interpretar objetos no formato *JavaScript Object Notation for Linked Data* (JSON-LD) [29]. Esses objetos basicamente descrevem "coisas" usando ontologias. Maiores detalhes serão dados na seção 3.4.1, por enquanto basta entender que o Aura aposta na simplicidade de trocar apenas objetos entre o *middleware* e os dispositivos, fugindo de complexidades como a necessidade de descrever cada nova tecnologia através de código, como visto na seção 2.1. Além disso, as tecnologias de inferência de conhecimento e descrição através de ontologias, além de facilitarem a integração de novos tipos de dispositivo ao sistema, oferecem a possibilidade de integração com a web semântica, abrindo um leque de aplicações a partir do conhecimento inferido da miríade de fontes de informação que a IoT traz consigo.

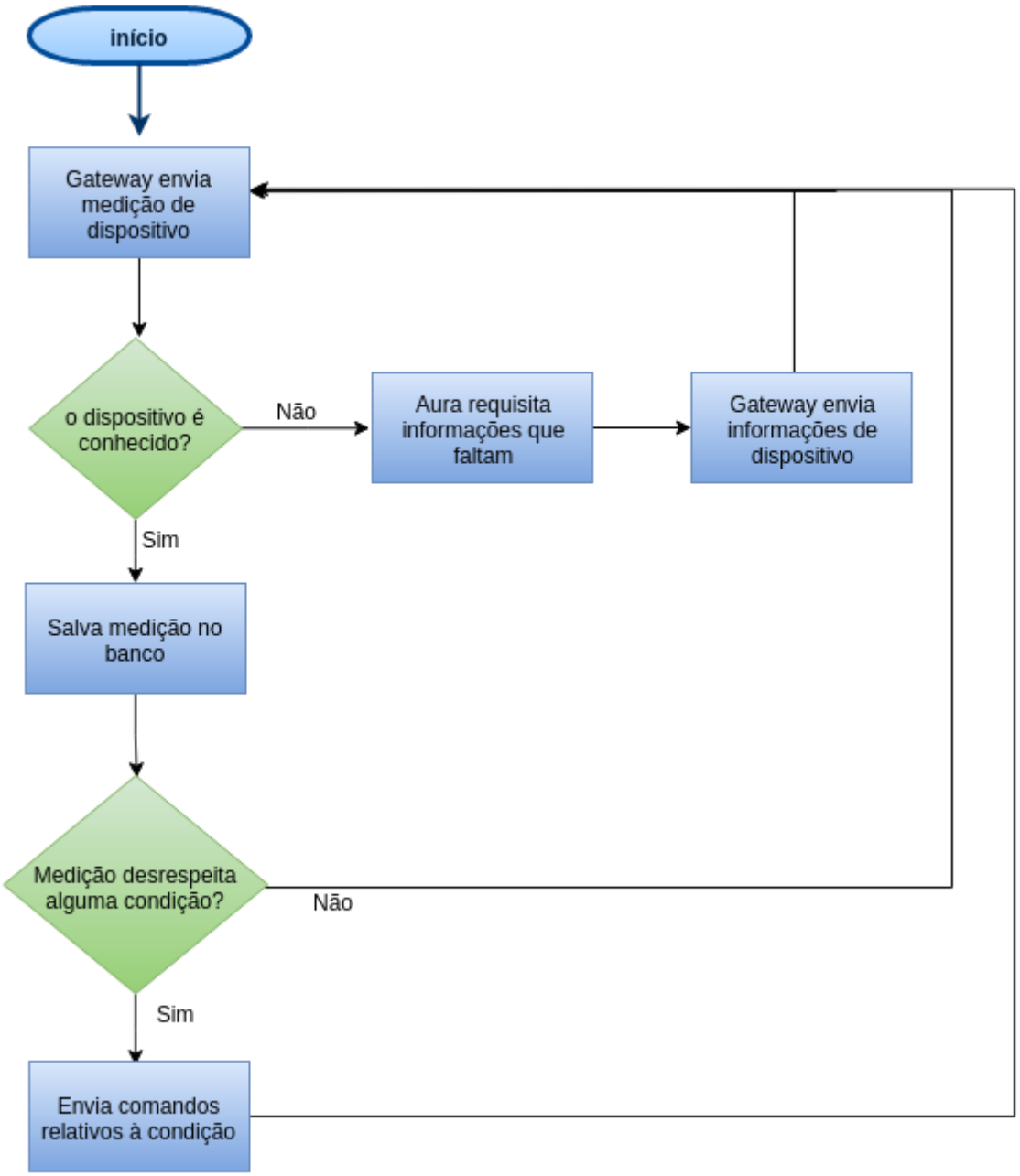
Uma vez que um tipo de dispositivo está integrado ao sistema e tem um *gateway* repassando suas medições para o *middleware*, suas informações são disponibilizadas a aplicações através de uma API REST, que será descrita na seção 3.5. A figura 8 ilustra o fluxo de trabalho de um sistema utilizando o Aura.

Figura 8: Fluxo de um sistema utilizando o Aura Middleware.



A descoberta de rede do *middleware* ocorre de forma simples, usando uma estratégia ociosa, isto é, não existe etapa de apresentação: supondo uma rede de dispositivos de mesmo tipo, que possui um *gateway* para o Aura, as medições dessa rede começam a ser enviadas e, caso o tipo de dispositivo não seja conhecido pelo sistema ainda, ele solicita ao *gateway* que envie os JSON-LDs com a descrição do dispositivo, seus sensores etc. A Figura 9 ilustra o fluxograma da interação entre um *gateway* e o *middleware*.

Figura 9: Fluxo de comunicação entre um *gateway* e o Aura Middleware.



3.1 Arquitetura

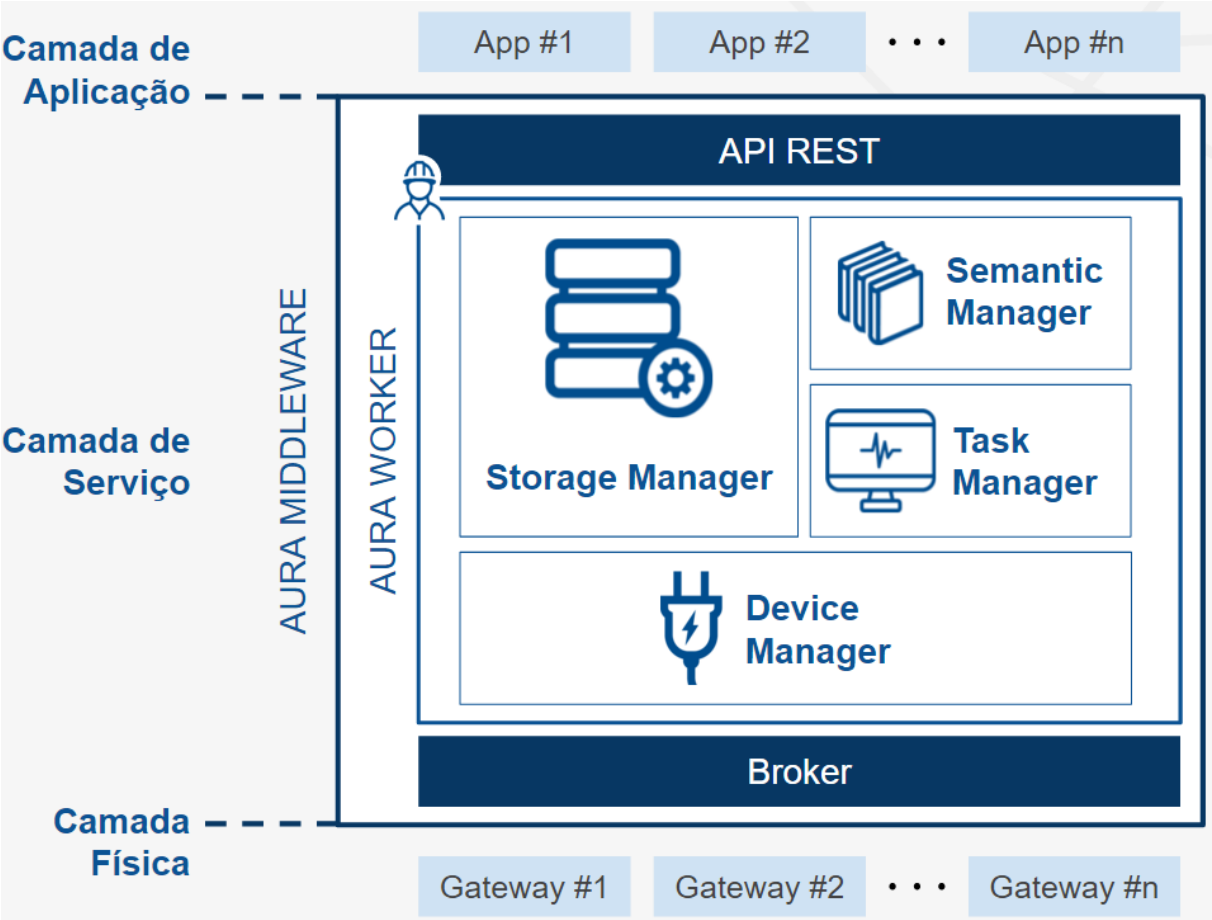
O Aura é estruturado por uma arquitetura orientada a serviços (SOA) [30], cujos componentes se baseiam em alguns dos princípios encontrados na literatura, especialmente os citados nas seções 2.2 e 2.3. Dentre esses, a escalabilidade horizontal é uma das principais características operacionais, para suportar até mesmo milhares de dispositivos conectados ao sistema. Para tanto, o Aura funciona como um serviço por demanda: sua modularização separa a camada de serviços, parte do *middleware* que realiza o "trabalho" de fato, em um componente abstrato chamado de *worker*. O *worker* oferece seus serviços a duas fontes de demanda: de um *broker*, que faz a comunicação com os *gateways* de dispositivos, e de requisições REST por meio da API.

Como o *worker* é um componente independente, ele pode ser instanciado múltiplas vezes para dividir o trabalho de grandes demandas. Assim, se existem diversos *gateways* enviando medições de sensores, o *broker* encaminha essas mensagens para um conjunto de *workers* separados em processos diferentes, de modo que cada um deles recebe uma parte do trabalho para realizar. Esse mecanismo de divisão de trabalho é implementado utilizando a tecnologia de comunicação ZeroMQ (ZMQ) [31]. Dessa forma, o middleware pode funcionar usando apenas um *worker*, ou vários, se for necessário.

A comunicação entre o *broker* e os *gateways*, por outro lado, é diferente da utilizada entre o *broker* e o *worker*. O protocolo usado é o *MQ Telemetry Transport* (MQTT) [32], desenvolvido pela IBM para estabelecer comunicação entre dispositivos de pouco poder computacional. Dentre as suas características, a simplicidade e a leveza o tornam ideal para estabelecer comunicação com os dispositivos mais simplórios da IoT, e ao mesmo tempo contemplam os dispositivos mais poderosos.

Como mencionado no início da seção 3, a API REST expõe os recursos da rede para o desenvolvimento de aplicações de gerenciamento, monitoramento e controle, e será detalhada na seção 3.5. A figura 9 representa arquitetura do middleware.

Figura 9: Arquitetura do Aura Middleware.



3.2 Gateways

No início da seção 3, foi dito que para que as “coisas” da IoT trabalhem em conjunto com o Aura, é necessário que existam *gateways* capazes de enviar JSON-LDs descrevendo os dispositivos e também de interpretar outros JSON-LDs para enviar-lhes comandos. Na prática, isso significa que algum modulo deve ser escrito para enviar e receber esses JSON-LDs. O *gateway* nesse caso nada mais é que um *software*, que a depender do fabricante pode ser executado em algum *hardware* específico como parte da rede de dispositivos (e.g., alguns produtos da DIGI [33]) ou pode ser executado em um computador conectado a um nó da rede por USB. Na verdade, ele pode ser executado no próprio dispositivo, se esse for dotado de recursos suficientes para fazer sua comunicação com o *middleware*.

O *gateway* é o protagonista quando se trata de integrar uma nova tecnologia ao Aura. Na verdade, esse processo consiste *precisamente* em escrever um *gateway* para a tecnologia a ser integrada. Isso quer dizer que, se for impossível desenvolver um *gateway* que envie, receba e interprete JSON-LDs usando MQTT para determinada tecnologia, o Aura Middleware não será capaz de se integrar com a mesma. Mas qual a probabilidade disso se tornar um problema real? Bem, ao examinar os requisitos envolvidos quando ao tipo de mensagem, e as características do gateway descritas no parágrafo anterior, é possível concluir que essa probabilidade não é muito grande: um JSON-LD nada mais é que um JSON comum, com

alguns campos específicos obrigatórios, para que as tecnologias de inferência semântica possam processá-lo adequadamente. Como JSON é um formato largamente utilizado, com suporte em mais de 60 linguagens de programação [34], essa parte não deve ser considerada como um problema. Quanto ao MQTT, já foi visto que é uma tecnologia que visa o funcionamento em diversas plataformas, ou seja, dos dispositivos mais simples aos mais complexos é possível escrever um software que use MQTT.

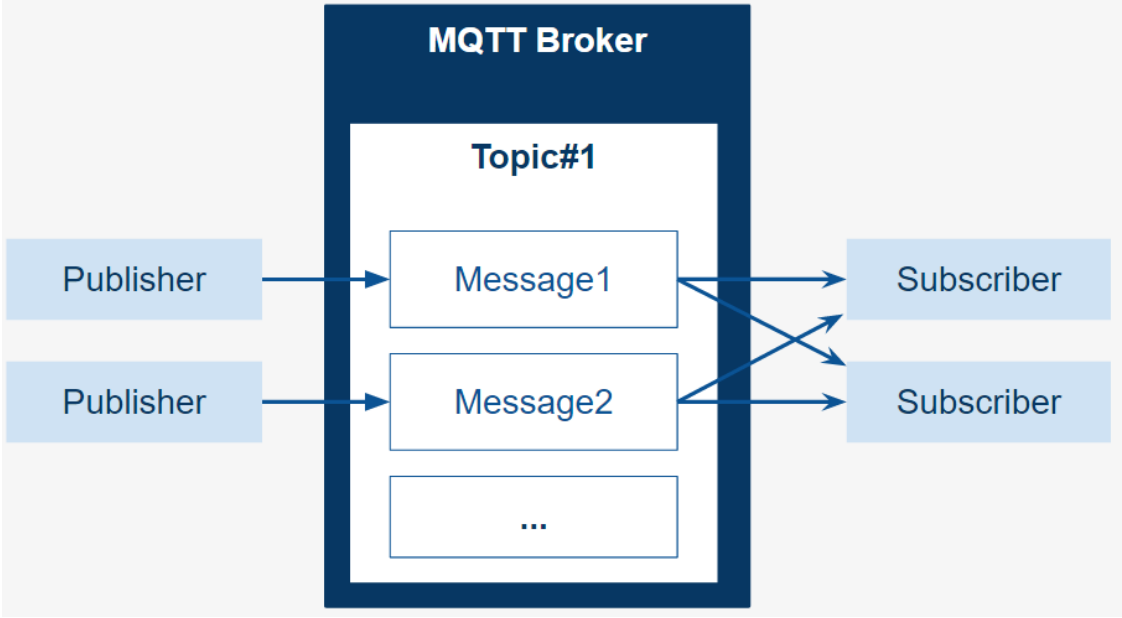
Sendo assim, o gateway pode ser considerado como uma entidade de software que se comunica via MQTT, enviando informações de um determinado grupo de dispositivos (ou até um único dispositivo) formatadas em JSON, e que interpreta JSONs recebidos para enviar comandos ao(s) dispositivo(s).

3.3 Aura Broker

O *broker* cumpre o papel de intermediar o fluxo de mensagens vindas dos dispositivos através dos *gateways* e repassar para o *worker*. Para tanto, ele precisa de duas interfaces de comunicação: do lado dos *gateways*, utiliza MQTT, enquanto do lado do *worker* utiliza ZMQ.

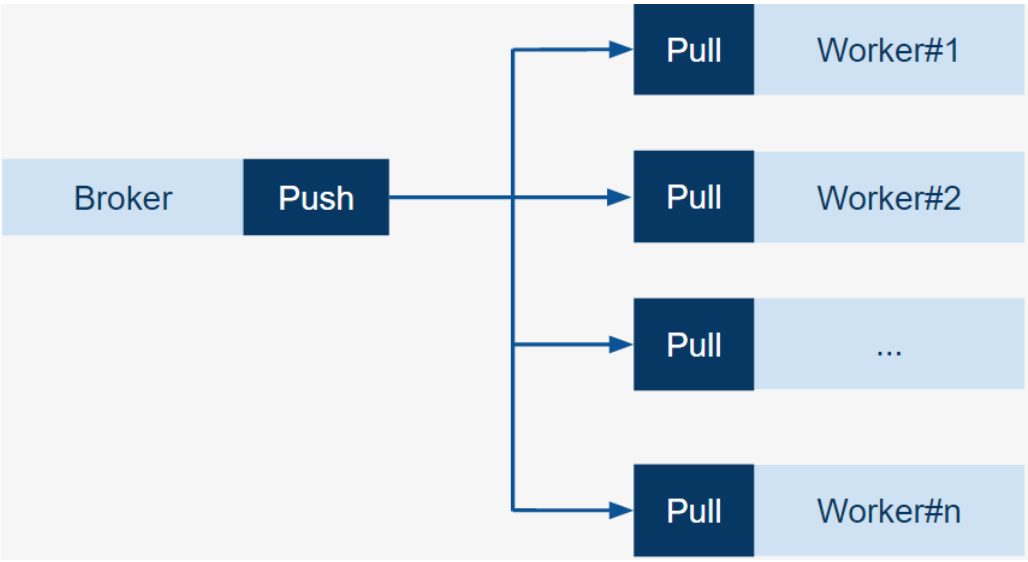
Do lado dos *gateways*, a comunicação acontece da seguinte forma: MQTT usa um padrão *publish/subscribe*, que funciona de forma análoga a um periódico. O *broker assina* um tópico, como a uma revista, e recebe todas as mensagens publicadas nele. Os *gateways*, por sua vez, *publicam* nesse tópico. Por exemplo, o broker assina um tópico chamado “aura/gateways”. Cada *gateway* publica suas mensagens nesse tópico, e o *broker* as recebe. Essa estrutura de tópicos é mantida por um agente de *software* à parte, que é um *MQTT broker*. Existem pelo menos 10 implementações de *MQTT brokers* disponíveis [35]. Um ponto interessante é que MQTT permite configurar o *broker* para armazenar as mensagens que são publicadas num tópico, e caso um “assinante” (um *subscriber*) não esteja conectado no momento da publicação, quando ele se conectar novamente receberá as mensagens que foram publicadas no tópico durante sua ausência.

Figura 10: Exemplo de comunicação via MQTT



Na seção 3.1, foi mencionado que o *broker* utiliza uma abordagem de ZMQ para se comunicar com o *worker* visando a distribuição de carga, caso existam várias instâncias do *worker* sendo executadas. Para tanto, é utilizado um padrão de comunicação *push/pull* baseado em *sockets* [36], o que na prática significa que o *broker* pode se conectar a vários *workers* ao mesmo tempo e, ao enviar uma mensagem por meio de um método *push* da biblioteca de ZMQ, é executado internamente um balanceamento de carga entre os vários *workers* aos quais o *broker* está conectado. A Figura 11 ilustra o padrão *push/pull* de ZMQ.

Figura 11: Padrão *push/pull* usado entre o Aura Broker e o Aura Worker.

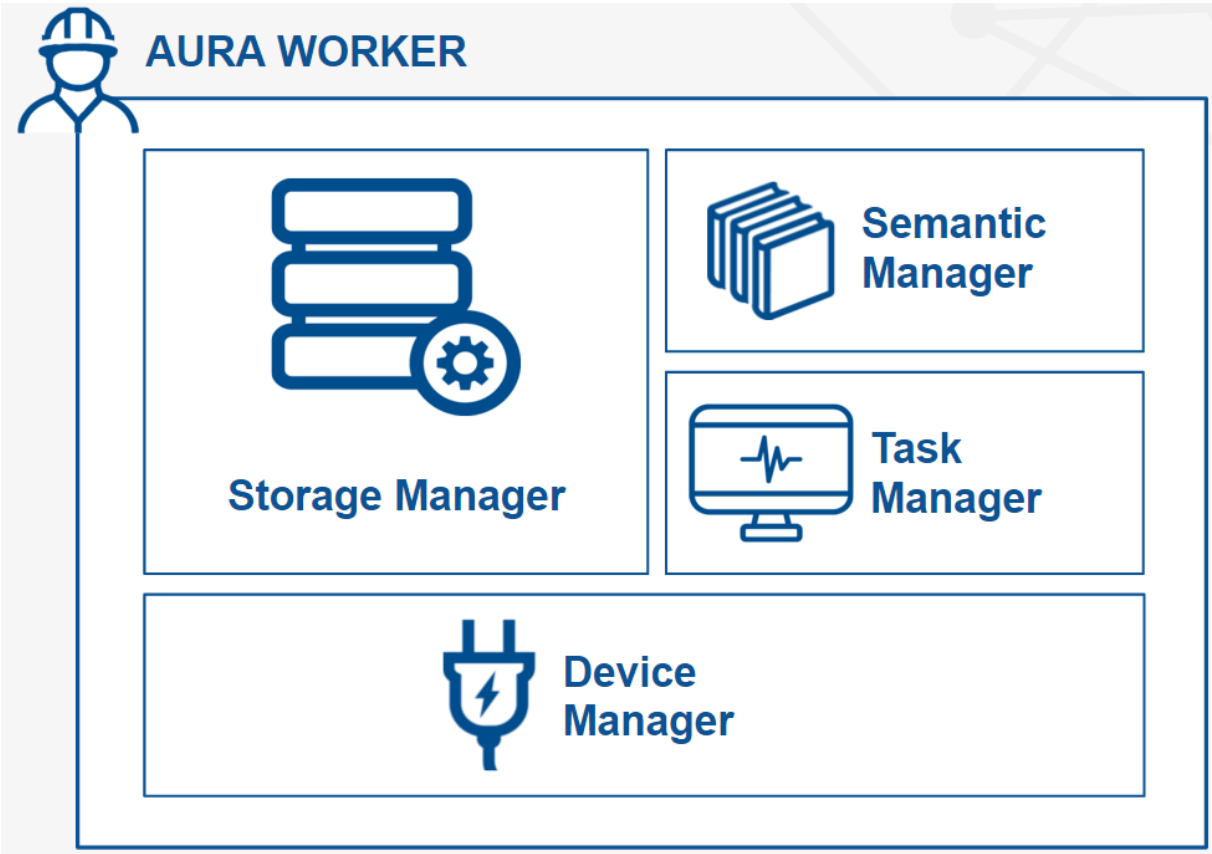


3.4 Aura Worker

O Aura Worker é o principal componente do *middleware* quando se trata de realizar o trabalho propriamente dito. Dentro dele estão os componentes que fornecem os serviços de armazenamento de dados, de inferência, de gerenciamento de dispositivos e tarefas da rede. É através de seus mecanismos que uma rede de sensores pode ser utilizada para realizar tarefas em cooperação com dispositivos diferentes que são apenas atuadores, por exemplo. E é também a partir dele que as informações serão expostas para desenvolvedores criarem aplicações de IoT sem se preocupar com a integração de dispositivos heterogêneos.

Partindo para a prática, o *worker* na realidade não é *um componente* composto de subcomponentes: cada um dos componentes internos é na verdade um agente de *software* independente. Alguns interagem através de troca de mensagens ZMQ, e outros são usados através de chamadas de função. Os componentes são: *Device Manager*, que provê os serviços para lidar com os dispositivos, *Task Manager*, que lida com as inferências de comandos e gerenciamento de condições, *Semantic Manager*, que provê ao *Task Manager* as funcionalidades semânticas para inferência, e o Storage Manager, responsável pela persistência dos dados. O trabalho cooperativo desses componentes é referenciado como *worker*, que como dito anteriormente é uma abstração. Abaixo, a Figura 12 apresenta a arquitetura interna do Aura Worker.

Figura 12: Arquitetura do Aura Worker.



3.4.1 Semantic Manager

O Semantic Manager é o primeiro componente do *worker* a ser explorado porque é nele que se descrevem as ontologias utilizadas no *middleware*, nas quais se baseia o trabalho do Device Manager e do Task Manager.

Resumidamente, uma ontologia é um estudo sobre entidades e suas relações. Mais especificamente, na computação, uma ontologia pode ser entendida como uma anotação dessas relações, uma representação de primitivas usadas para modelar conhecimento. Diversas linguagens podem ser utilizadas para escrever ontologias, como OWL DL, OWL 2, RDF e RDF Schema [37]. Anotar uma ontologia em uma linguagem significa, na prática, descrever relações entre duas entidades, que constituem *triplas* do formato entidade - relação - entidade. É através dessas relações que é possível realizar consultas e inferir conhecimento em bases de dados de triplas.

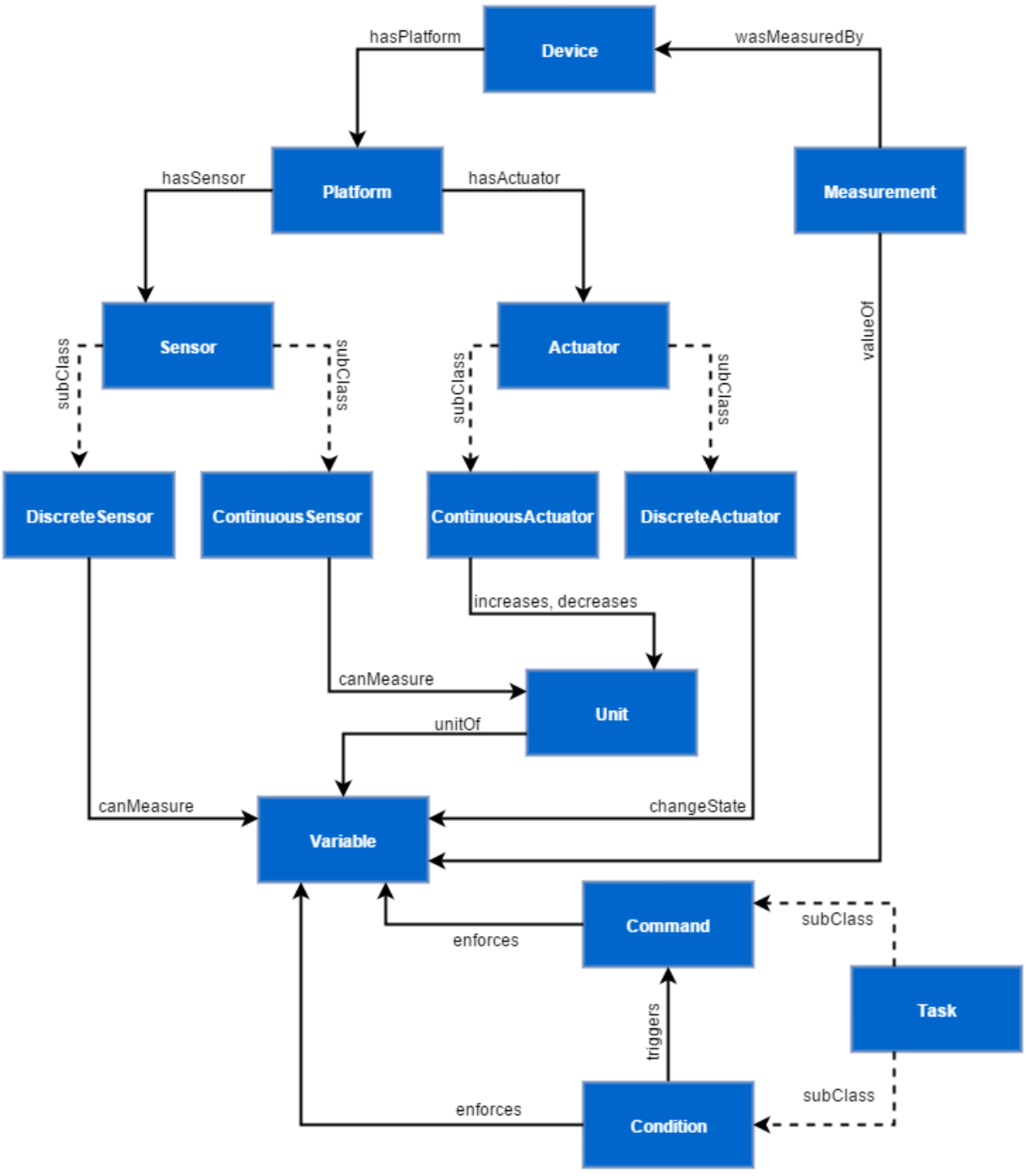
No Aura, as entidades das ontologias são descritas e trafegam no formato JSON-LD devido a facilidade de manipulação e amplo suporte (como mencionado em seções anteriores), porém, ao chegar ao Semantic Manager, esse objeto é transformado para ser processado via uma biblioteca de RDF (estabelece uma tabela de abreviações), que usa uma notação baseada em XML.

Uma vez que existem ontologias escritas, elas podem ser hospedadas em um servidor e referenciadas em algum objeto que deseje usar suas informações. Desse modo, é possível descrever entidades que contenham informações de várias ontologias de domínios diferentes,

assim como uma ontologia pode se relacionar com outra. Para ser devidamente referenciada, cada ontologia e entidade precisa ter um identificador único, chamado *Uniform Resource Identifier* (URI).

No caso do Aura, foram utilizadas 4 ontologias: AuraDevice, AuraSense, AuraActuate e AuraTask. Dentro de uma ontologia é possível derivar uma classe de outra, como por exemplo na ontologia AuraSense, onde as classes ContinuousSensor e DiscreteSensor são subclasses da classe Sensor. As relações podem ser identificadas por etiquetas, facilitando seu uso ao descrever uma entidade. A Figura 13 ilustra o grafo de relações entre as classes das ontologias do Aura.

Figura 13: Relações entre as classes das ontologias usadas no Aura.



Traduzindo as relações entre as classes para a prática: uma entidade de uma determinada ontologia que tem relação com outra ontologia precisa, em sua descrição, conter uma referência para uma entidade da ontologia com a qual se relaciona. Ou seja, ao descrever um dispositivo do Aura, que possui uma plataforma, o JSON-LD desse dispositivo deve conter um campo com a URI do documento que descreve sua plataforma. A Figura 14 mostra um trecho de código de criação de um dispositivo. O campo “@id” corresponde à URI do dispositivo, enquanto o campo “dev:hasPlatform” aponta para sua plataforma.

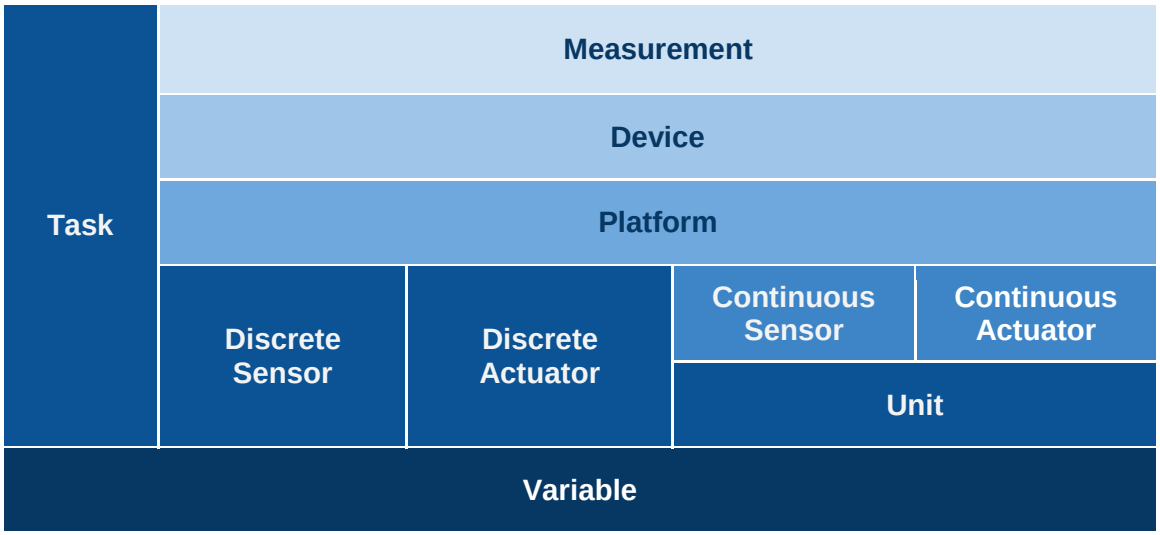
Figura 14: Função de criação de um JSON-LD de dispositivo do Aura.

```
def createDevice(id, url, platformId):
  device = {
    "id":id,
    "@context": urls.contextUrl + "deviceContext.jsonld",
    "@type":"Device",
    "@id":url + "devices/" + id,
    "dev:hasPlatform":url + "platforms/" + platformId
  }

  return device
```

Visto que quando uma entidade se relaciona com outra é necessário informar o URI dessa outra, é possível concluir que existem interdependências entre as classes das ontologias: uma medição precisa apontar para um dispositivo, que necessita de uma plataforma, que precisa de sensores ou atuadores e assim por diante. A Figura 15 simboliza as dependências de cima para baixo, em formato de pilha.

Figura 15: Pilha de dependência entre as classes das ontologias.



Naturalmente, essas interdependências geram a necessidade de algumas verificações por parte do *middleware* ao receber JSON-LDs, que são realizadas pelo Device Manager e serão vistas na próxima seção.

É desta forma, então, que o Aura Middleware se sobrepõe à heterogeneidade de dispositivos: através de descrições semânticas nos termos das classes mencionadas. Dotado dessas descrições ontológicas, o Semantic Manager pode executar consultas que saltam de entidade em entidade através das relações e inferir quais os dispositivos da rede podem executar determinadas tarefas. Por exemplo, através de uma unidade de graus Celsius, pode-se descobrir que ela está relacionada a uma variável Temperatura, e que existem alguns sensores diferentes que a medem, e que na rede existem vários dispositivos que possuem esses sensores. Assim como é possível verificar quais os dispositivos na rede que são capazes de atuar sobre a Temperatura, como, por exemplo, um ar-condicionado ou um aquecedor.

Para realizar essas consultas, é utilizada a linguagem SPARQL [38], escolhida por possuir uma sintaxe semelhante a SQL, o que confere flexibilidade de consulta e rápida familiarização visto que SQL é amplamente difundida na comunidade da computação. Os serviços do Semantic Manager são utilizados pelos outros componentes do *worker* através de chamadas de função.

3.4.2 Device Manager

O Device Manager é o agente de software que se comunica diretamente com o Aura Broker. Isto é, as mensagens dos dispositivos que passam pelos *gateways* e são encaminhadas pelo *broker* utilizando o modelo *push/pull* do ZMQ para o *worker*, na verdade são enviadas especificamente para o Device Manager.

A partir de seu processamento interno, as mensagens podem ser simplesmente armazenadas no banco de dados (utilizando, naturalmente, os serviços do Storage Manager) ou ser encaminhadas também ao Task Manager para verificar se estão dentro das condições criadas no sistema.

Cumprindo o papel de interface entre os níveis mais próximos da camada física e o restante dos componentes do *worker*, o Device Manager também é responsável por encaminhar mensagens “para baixo” no que remete à arquitetura em camadas da Figura 9. Por exemplo, se o Task Manager conclui que é necessário enviar um comando a um dispositivo, ele envia uma mensagem via ZMQ para o Device Manager, que então a endereça para o *gateway* correto para recebê-la através do *broker*.

O processamento das mensagens funciona de acordo com a seguinte lógica: se a mensagem é uma medição, o Device Manager verifica se o dispositivo autor da medição é conhecido pelo sistema. Caso positivo, a medição é armazenada no banco e enviada para o Task Manager verificar se ela satisfaz as condições existentes. Caso negativo, o Device Manager solicita que o *gateway* envie toda a pilha (Figura 15) de informações abaixo da medição, de baixo para cima. Esse processo se repete para cada tipo de mensagem, garantindo que nada será inserido no banco sem que o middleware tenha as devidas descrições semânticas que embasam aquela informação.

3.4.3 Task Manager

O Task Manager, assim como o Device Manager, é um agente de software independente. Ele se comunica com o Device Manager através de ZMQ, e utiliza os serviços do Storage Manager

e do Semantic Manager. Ele é o responsável por utilizar os serviços semânticos do *middleware* para criar condições de uso da rede e verificar se as medições às respeitam, examinando comandos possíveis de serem usados e, quando necessário, comunicar ao Device Manager que um comando deve ser enviado a um determinado dispositivo.

Existem dois tipos de tarefa com os quais o Task Manager lida: comandos e condições (ambas são classes, Command e Condition, que herdam da classe Task, vide Figura 12). Um comando é uma capacidade de atuação de um dispositivo, verificada através de sua relação com um atuador que, por conseguinte se relaciona com uma variável. Comandos podem ser utilizados para que um dispositivo realize uma ação prontamente. Uma condição, por outro lado, pode ser vista como um valor (ou um intervalo de valores, no caso de uma variável contínua) que uma variável deve possuir.

Compondo os dois tipos de tarefas, é possível programar comportamentos e reações na rede. Caso uma condição seja desrespeitada, comandos podem ser utilizados de formas variadas, para forçar a variável a voltar ao estado ideal ou mesmo realizar outras rotinas em consequência da quebra da condição. Cada condição pode ser ligada diretamente a seu correspondente, agilizando a reação do *middleware* à quebra da condição. Caso não exista essa ligação direta com um comando, o *middleware* gera apenas uma notificação sobre a quebra da condição, ficando a critério da aplicação como reagir.

O TaskManager expõe serviços de busca por possíveis condições e comandos a serem criados no sistema através da API REST, que será vista na seção 3.5.

3.4.4 Storage Manager

O Storage Manager serve como uma interface entre o banco de dados utilizado e os demais componentes. Seu principal serviço é encapsular operações típicas de armazenamento e recuperação de dados, de modo que a tecnologia do banco em si poderia ser trocada alterando apenas o Storage Manager, sem maiores impactos nos outros componentes que requisitam suas funcionalidades.

O banco atualmente utilizado no Aura é o MongoDB [39], pelas seguintes razões: é um banco NoSQL baseado em documentos, e os JSON-LDs são documentos por si só, fator que o torna uma alternativa confortável para trabalhar com esse tipo de dado. Não existe a necessidade de definir tabelas ou um modelo de dados, basta salvar o documento. Mais precisamente, o MongoDB possui estruturas chamadas *collections*, para agrupar documentos de um mesmo tipo. Essa funcionalidade se encaixa bem com um domínio modularizado como o do Aura: medições, dispositivos, plataformas etc. Cada tipo de JSON-LD é armazenado em uma *collection* própria no banco.

Outro aspecto proveitoso é a relação que existe entre o MongoDB e o *framework* Eve [40], que foi usado para construir a API REST: uma vez definidas as *collections*, pouco esforço de desenvolvimento é suficiente para relacioná-las aos *endpoints* da API, como descrito a seguir.

3.5 API REST

Uma vez que o Aura consegue interpretar que tipo de dispositivos fazem parte da rede, e que tipo de recurso eles oferecem, tanto em termos de sensoriamento quanto de atuação, ele expõe esses recursos e informações através de sua API REST. Além disso, o papel da API não é apenas disponibilizar os recursos da rede do Aura, mas também permitir que aplicações adicionem novas informações no banco de dados. Informações que podem ser novas plataformas, sensores, atuadores, enriquecendo o “conhecimento” do sistema sobre possíveis configurações de dispositivos. Naturalmente, as possibilidades de comandos e condições não podem ser criadas, pois dependem diretamente das capacidades dos dispositivos e são inferidos pelo próprio *middleware*, através do TaskManager em conjunto com o SemanticManager.

Por outro lado, é apenas através da API que as tarefas do sistema podem ser instanciadas Enviando um JSON-LD no corpo de uma requisição HTTP do tipo POST, uma aplicação insere no banco do Aura uma condição que será verificada sempre que uma nova medição daquela variável for inserida, ou um comando que pode ser enviado para um determinado dispositivo executar. Como mencionado na seção 3.4.3, uma condição pode apontar para o comando que deve ser executado caso ela seja desrespeitada. A Tabela 1 mostra os verbos HTTP utilizados na API, enquanto a Tabela 2 elenca os *endpoints* acompanhados dos verbos HTTP permitidos e de sua descrição.

Tabela 1: Verbos HTTP utilizados na API do Aura Middleware.

Verbo HTTP	Ação
GET	Leitura
POST	Criação
PUT	Substituição
DELETE	Remoção

Tabela 2: Endpoints da API REST do Aura Middleware.

Verbos HTTP	Endpoint	Descrição
GET, POST, PUT, DELETE	/measurements	Lista de medições dos dispositivos.
	/measurements/<id>	Medição com id correspondente ao especificado.
	/devices	Lista de dispositivos.
	/devices/<id>	Dispositivo com id correspondente ao especificado.
	/platforms	Lista de plataformas.
	/platforms/<id>	Plataforma com id correspondente ao especificado.
	/sensors/continuous	Lista de sensores de valores contínuos.
	/sensors/continuous/<id>	Sensor de valores contínuos com id correspondente ao especificado.
	/sensors/discrete	Lista de sensores de valores discretos.
	/sensors/discrete/<id>	Sensor de valores discretos com id correspondente ao especificado.
	/actuators/continuous	Lista de atuadores sobre valores contínuos.
	/actuators/continuous/<id>	Atuador sobre valores contínuos com id correspondente ao especificado.
	/actuators/discrete	Lista de atuadores sobre valores discretos.
	/actuators/discrete/<id>	Atuador sobre valores discretos com id correspondente ao especificado.
	/units	Lista de unidades de medição de valores contínuos.
	/units/<id>	Unidade com id correspondente ao especificado.
	/variables	Lista de variáveis observadas.
	/variables/<id>	Variável com id correspondente ao especificado.

	/conditions	Lista de condições programadas no sistema.
	/conditions/<id>	Condição com id correspondente ao especificado.
	/commands	Lista de comandos.
	/commands/<id>	Comando com id correspondente ao especificado.
GET	/conditions/possibilities	Lista que indica quais dispostivos podem medir quais variáveis
	/commands/possibilities	Lista que indica quais dispostivos podem atuar sobre quais variáveis
	/graph	Retorna o grafo semântico no formato RDF/XML

Para finalizar é possível destacar as principais características do Aura Middleware:

1. Possui uma arquitetura orientada a serviços (SOA), que lhe confere possibilidade de escalabilidade horizontal;
2. Utiliza MQTT para se comunicar com dispositivos, possibilitando comunicação mesmo com os mais simplórios;
3. Usa descrições semânticas em JSON-LD para abstrair a heterogeneidade dos dispositivos conectados, de modo que a única tarefa para integrar uma nova tecnologia de dispositivo é escrever um *gateway* que trabalhe com JSON-LD, independente de linguagem;
4. Utiliza um banco de dados NoSQL baseado em documentos, MongoDB;
5. Dispõe de API REST para expor seus recursos e permitir a criação de condições e comandos para executar na rede.

4. Experimentos

Neste capítulo serão apresentados alguns experimentos, acompanhados de seus resultados e uma discussão sobre eles. A Tabela 3 sumariza as tecnologias utilizadas na implementação, enquanto a Tabela 4 contém as informações do computador utilizado para realizar os testes.

Tabela 3: Tecnologias usadas no Aura Middleware.

Recurso/Atividade	Tecnologia
Componentes do middleware	Python 3.4
Simulador de Gateways	Python 3.4
Comunicação com gateways	MQTT
Comunicação entre processos	ZMQ
Serialização dos dados	JSON-LD
Descrição das Ontologias	OWL
Grafo Semântico	RDFS via rdflib e rdflib-jsonld
Consultas Semânticas	SPARQL
Banco de Dados	MongoDB
Profiler	line_profiler 1.0

Tabela 4: Informações técnicas do computador usado nos testes.

Processador	Intel Core i5 4690 3.5 GHz
Memória RAM	8GB 1600MHz *single channel*
Disco Rígido	2TB 5400 rpm
Sistema Operacional	Linux Kubuntu 14.04 x64

4.1 Cenário 1: Entrada de Novos Dispositivos

Este cenário consiste na simulação da entrada de novos dispositivos na rede do Aura. Para a realização dos testes foi utilizado um simulador de *gateway*, implementado em Python. Seu funcionamento consiste basicamente em criar instâncias de medições e enviar ao Aura Broker. A partir do momento em que o *middleware* responde avisando que desconhece os dispositivos que geraram as medições, o *gateway* envia essas informações, conforme

explanado no capítulo 3. Feito isso, o *gateway* volta a enviar medições com valores aleatórios. A Tabela 5 contém as informações sobre as mensagens enviadas durante os testes, exceto as primeiras medições, descartadas pelo *middleware* até que os dispositivos fossem reconhecidos.

Tabela 5: Mensagens enviadas pelo *gateway* de testes.

Tipo de Mensagem	Quantidade
Medição	1001
Dispositivo	50
Plataforma	50
Sensor Contínuo	25
Atuador Contínuo	25
Sensor Discreto	25
Atuador Discreto	25
Unidade	25
Variável	50

4.2 Cenário 2: Requisições para a API

Como descrito na seção 3.5, através da API é possível monitorar as informações da rede do Aura, como medições, dispositivos, plataformas etc. Além disso, pode-se conferir quais são as possibilidades de criação de condições e comandos: ao enviar uma requisição GET para /condtions/possibilities, por exemplo, é retornado um JSON com informações sobre qual dispositivo é capaz de medir valores de qual variável. A partir dessas informações, um *script* Python foi utilizado para enviar requisições HTTP automaticamente ao *middleware* e criar condições e comandos. Além disso, foi utilizado o software *Postman* [41], para enviar requisições manualmente e visualizar as informações possíveis de monitorar.

4.3 Métricas Observadas

Durante os testes, além da validação da arquitetura proposta, foram observadas algumas métricas de desempenho: a **vazão de mensagens** que uma instância do *worker* suporta trabalhar com atraso máximo de 1 segundo, o **espaço de armazenamento** usado, o **tempo de resposta da API** para as consultas SPARQL e os **gargalos de processamento** do *middleware*.

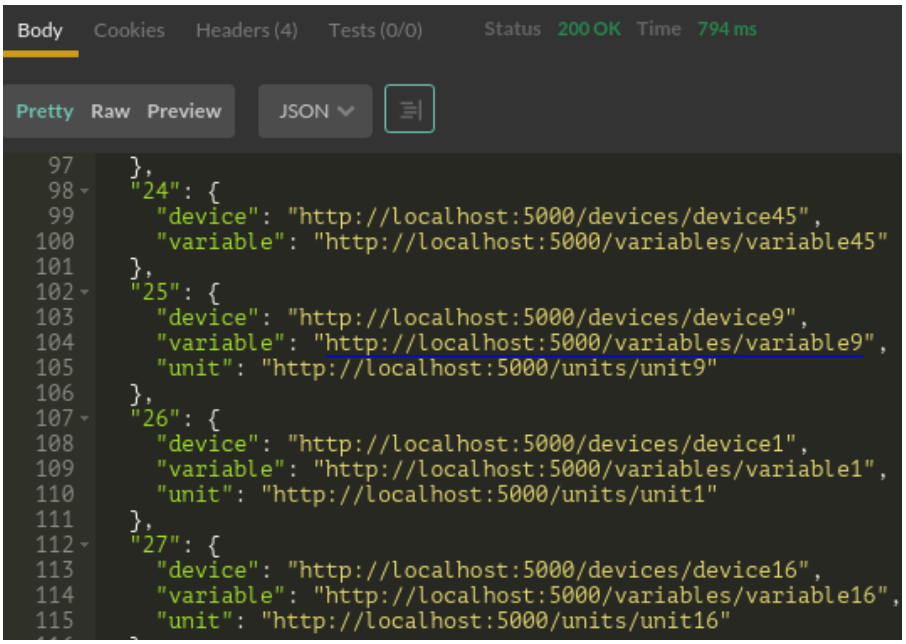
4.4 Resultados e Discussão

Quanto à vazão de mensagens, observou-se que o Aura Worker consegue trabalhar com atrasos de até 1 segundo para concretizar o processo de análise e armazenamento das mensagens quando recebe uma mensagem por segundo. Para uma vazão maior, seria adequado executar mais de uma instância, de modo a evitar atrasos e perda de informação.

O espaço em disco utilizado foi medido usando funções do próprio MongoDB; o tamanho total das informações armazenadas pelo Aura foi de aproximadamente 1 *mega byte* (MB). Como as medições são o tipo de mensagem esperado para existir em maior quantidade, é válido analisá-las em maiores detalhes. Cada medição tem em média 496 *bytes* (B), de modo que o tamanho total das 1001 medições foi de 484 *kilo bytes* (kB). Considerando a vazão máxima de uma mensagem por segundo, com apenas uma instância do Aura Worker, a cada dia seriam aproximadamente 40 *mega bytes* (MB) de medições. Porém, deve-se observar que essa quantidade de medições não necessariamente é um requisito para aplicações de IoT. Em um cenário residencial, onde o armazenamento de dados é mais restrito que na indústria, por exemplo, aplicações comuns seriam a medição de temperatura, controle de eletrodomésticos e afins, situações onde não é necessária uma frequência tão alta de envio de mensagens. Além disso, é possível que as aplicações utilizem técnicas de agregação de dados e até mesmo descarte de medições, especialmente para dados antigos. Em um cenário de maior porte, o mesmo tipo de estratégia pode ser aplicado, além de que a disponibilidade de armazenamento é naturalmente maior.

Quanto ao tempo que o Aura leva para responder a requisições com consultas SPARQL, foram verificados os tempos de resposta para se descobrir os possíveis comandos que podem ser criados, assim como as possíveis condições. Foram utilizados 50 sensores, 50 atuadores e 50 variáveis, que por questões de simplicidade dos experimentos foram mapeados de um para um. Sendo assim, as respostas tanto de condições quanto comandos contém 50 possibilidades diferentes (isso não impede que em uma situação real existam dispositivos com vários sensores e atuadores, aumentando as possibilidades)Os resultados em média demoram menos de um segundo para serem obtidos, como pode ser observado no topo da Figura 16, no campo *Time*.

Figura 16: Trecho do JSON de resposta a um GET para /conditions/possibilities.



Buscando compreender onde residem os gargalos de processamento, foi utilizado o *line_profiler* [42] de Python 3. Foram analisadas especialmente as funções de consulta SPARQL que buscam por possíveis condições e comandos. Os resultados apontaram que 99,9% do tempo de processamento é gasto para transformar os dados do grafo semântico, depois que ele é armazenado no banco em formato de cadeia de *bytes* (análogo ao formato *string*), novamente em um grafo. A Figura 17 mostra a função *get_graph()*, responsável pelo processo descrito acima, e na Figura 18 pode-se conferir no resultado do *profiler* que a linha 14 da função *get_graph()*, que corresponde à construção do grafo semântico, consome 99,9% do tempo de processamento.

Figura 17: Função *get_graph()*, que retorna o grafo semântico.

```
 9   @profile
10  def get_graph():
11      _graph = Graph(store)
12      stored_graph = db.get('graph','graph','id')
13      if stored_graph != None:
14          _graph.parse(data=stored_graph['graph'].decode(),format='xml')
15      return _graph
```

Figura 18: Resultado do *profiling* realizado na função *get_graph()*.

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     9                                           
    10                                           
    11         4        350     87.5      0.0  
    12         4       2907    726.8      0.1  
    13         4          5      1.2      0.0  
    14         4    3057880 764470.0     99.9  
    15         4          5      1.2      0.0  
```

Essa informação leva ao seguinte raciocínio: o grafo precisa ser reconvertido em seu formato original para que as consultas sejam realizadas. Caso essa modificação não ocorresse, esse tempo seria poupado. Logo, uma possibilidade de aperfeiçoamento seria investigar formas de manter o grafo semântico constantemente em memória, em seu formato original. Essa ideia será mais elaborada no capítulo 6, que aborda os trabalhos futuros.

Sumarizando os resultados, observa-se:

1. A arquitetura proposta é aplicável e seu fluxo de comunicação fornece respostas em tempos inferiores a um segundo;
2. Uma única instância é capaz de lidar com um fluxo de até uma mensagem por segundo;
3. O espaço em disco é plausível de ser utilizado com algumas restrições de vazão, podendo ser aplicadas técnicas de agregação de dados para otimizar o espaço;
4. O gargalo de processamento reside na construção do grafo semântico quando o mesmo é armazenado em um banco de dados que muda seu formato original, problema que pode ser melhorado buscando por formas mais eficazes de salvar o grafo.

5. Conclusão e Trabalhos Futuros

A Internet das Coisas já começou a se estabelecer e seu crescimento é previsto em grandes números. As contribuições e facilidades que ela trará à vida humana são diversas e parte delas ainda é difícil de imaginar com precisão. Portanto, é necessário continuar sua edificação e estabelecer as bases para os desenvolvimentos futuros.

Nesse sentido, este trabalho trouxe à tona a problemática da integração entre os dispositivos que devem compor a IoT, que são de fabricantes diferentes e usam tecnologias distintas. O levantamento do estado da arte apontou que as propostas existentes não oferecem facilidade de integração e desenvolvimento de aplicações. Assim, a maior contribuição aqui apresentada foi a proposta de um *middleware* capaz de abstrair a heterogeneidade entre tecnologias através de uma abordagem semântica, possibilitando desenvolvimento simplificado e com o mínimo de conhecimentos específicos. Usando o Aura Middleware, a integração de uma nova tecnologia precisa apenas do desenvolvimento de um *gateway*.

Foram realizados testes de funcionalidades e desempenho, através dos quais se comprovou a facilidade de desenvolvimento almejada: o processo de desenvolver um *gateway* requer apenas que ele trabalhe com descrições semânticas, usando o formato bem estabelecido de JSON, e MQTT, padrão que também conta com implementações nas mais diversas linguagens. Em contraste, o estado da arte necessita do desenvolvimento de componentes de software para adicionar ao próprio *middleware*, exigindo conhecimento específico do domínio e resultando mais uma vez em tecnologias fortemente acopladas e pouco flexíveis. Quanto ao desempenho, a eficiência do Aura foi posta à prova por meio de *profiling* de código e análise de tempos de resposta da API REST, interface direta entre o *middleware* e os desenvolvedores. Com respostas em menos de um segundo e alternativas de aperfeiçoamento do desempenho semântico, esse aspecto também não deixou a desejar.

Dessa forma, os objetivos do trabalho foram atingidos quanto a oferecer uma alternativa que permita desenvolver aplicações de IoT integrando diferentes tecnologias com pouco esforço.

As análises feitas a partir dos testes apontam um leque de possíveis investigações e aprimoramentos do Aura Middleware. Uma melhoria evidente seria buscar por uma forma de manter o grafo semântico em memória, que tem como possibilidade utilizar um banco de dados baseado em memória, como o Redis [43]. Naturalmente, seria preciso investigar minuciosamente suas formas de armazenamento e impactos de performance.

Outro aspecto a ser explorado é a expansão das ontologias para melhor contemplar as características da miríade de dispositivos da IoT, ou permitir ao *middleware* interpretar outras ontologias já criadas com essa finalidade para oferecer portabilidade com outras formas de integração.

Seria também interessante realizar testes mais ostensivos, inclusive com dispositivos reais, além dos testes com simuladores. Esse tipo de teste tem potencial para revelar casos específicos de problemas e até mesmo apontar melhoramentos em termos de possíveis funcionalidades. Mais especificamente, realizar testes em larga escala para entender o comportamento do *middleware* e a forma de crescimento da quantidade de instâncias de *worker* necessárias para atender a uma demanda crescente.

Por fim, a segurança é um fator que não foi explorado durante esse trabalho e é digno de futuras investigações a fim de evidenciar vulnerabilidades e posteriormente fortalecer o *middleware* nesse quesito.

6. Bibliografia

[1] http://www.rfidjournal.com/articles/view?4986 (acessado em 14/12/15)

[2] http://www.itu.int/ITU-T/recommendations/rec.aspx?rec=y.2060 (acessado em 14/12/15)

[3] Gartner, Inc. “Gartner Says a Thirty-Fold Increase in Internet-Connected Physical Devices by 2020 Will Significantly Alter How the Supply Chain Operates”.<http://www.gartner.com/newsroom/id/2688717> (acessado em 14/12/2015)

[4]http://www.forbes.com/sites/quora/2013/01/07/how-many-things-are-currently-connected-to-the-internet-of-things-iot/ (acessado em 01/10/15)

[5] L. Atzori, A. Iera, G. Morabito. “The Internet of Things: A Survey”, 2010.

[6] Kalil A. Bispo, Nelson S. Rosa, Paulo R. F. Cunha. “Um Middleware Semântico Orientado a Mensagens para Rede de Sensores sem Fio”.

[7] http://web.archive.org/web/20120629211518/http://www.middleware.org/whatis.html (acessado em 15/12/15)

[8] S. Hadim, N. Mohamed. “Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks”.

[9] A. Zaslavsky, C. Perera, D. Georgakopoulos. “Sensing as a Service and Big Data”.

[10] http://www.infoworld.com/article/2611319/computer-hardware/the--internet-of-things--will-mean-really--really-big-data.html (acessado em 14/12/15)

[11] http://www.scaledb.com/internet-of-things-database.php#requirements (acessado em 15/12/15)

[12] http://web.dfc.unibo.it/buzzetti/IUcorso2007-08/mdidattici/ontology-definition-2007.htm

[13] Z. Qin, G. Denker, C. Giannelli, P. Bellavista, N. Venkatasubramanian. “A Software Defined Networking Architecture for the Internet-of-Things”.

[14] Z. Song, Alvaro A. C´ardenas, R. Masuoka. “Semantic Middleware for the Internet of Things”.

[15] Isam Ishaq et al. “IETF Standardization in the Field of the Internet of Things (IoT): A Survey”.

[16] http://www.w3.org/2005/Incubator/ssn/wiki/Main_Page (acessado em 20/12/15)

[17] http://www.hypercat.io/ (acessado em 20/12/15)

[18] http://ogc-iot.github.io/ogc-iot-api/index.html (acessado em 20/12/15)

[19] https://allseenalliance.org/framework/documentation/learn (acessado em 20/12/15)

[20] http://www.service-architecture.com/articles/web-services/service-oriented_architecture_soa_definition.html (acessado em 16/12/15)

[21] http://hydramiddleware.eu/news.php (acessado em 15/12/15)

[22] P. Kostelník, M. Sarnovsky, K. Furdík. “THE SEMANTIC MIDDLEWARE FOR NETWORKED EMBEDDED SYSTEMS APPLIED IN THE INTERNET OF THINGS AND SERVICES DOMAIN”

[23] T. Sabol, J. Hreno. “Semantic middleware in Applications of the Internet of Things”.

[24] Claus Pahl. “Semantic Model-Driven Architecting of Service-based Software Systems”.

[25] Lucas Lira Gomes. “WSN Manager: Uma arquitetura de middleware sensível a contexto para redes de sensores heterogêneas”. Trabalho de Graduação, Centro de Informática da Universidade Federal de Pernambuco, 2014.

[26] https://nodejs.org/en/about/ (acessado em 19/12/15)

[27] https://iotdb.org/pub/ (acessado em 19/12/15)

[28] https://github.com/dpjanes/iotdb-homestar (acessado em 19/12/15)

[29] http://json-ld.org/ (acessado em 07/01/16)

[30] http://www.service-architecture.com/articles/web-services/service-oriented_architecture_soa_definition.html (acessado em 07/01/16)

[31] http://zeromq.org/ (acessado em 07/01/16)

[32] http://mqtt.org/ (acessado em 07/01/16)

[33] http://www.digi.com/products/xbee-rf-solutions/gateways (acessado em 08/01/16)

[34] http://www.json.org/ (acessado em 08/01/16)

[35] https://github.com/mqtt/mqtt.github.io/wiki/brokers (acessado em 08/01/16)

[36] http://learning-0mq-with-pyzmq.readthedocs.org/en/latest/pyzmq/patterns/pushpull.html (acessado em 07/01/16)

[37] http://semanticweb.org/wiki/Ontology (acessado em 09/01/16)

[38] http://www.w3.org/TR/rdf-sparql-query/ (acessado em 09/01/16)

[39] https://www.mongodb.org/ (acessado em 09/01/16)

[40] http://python-eve.org/ (acessado em 09/01/16)

[41] https://www.getpostman.com/ (acessado em 16/01/16)

[42] https://github.com/rkern/line_profiler (acessado em 16/01/16)

[43] http://redis.io/ (acessado em 18/01/16)