Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Programa de Pós-Graduação em Ciência da Computação

Thomás Marques Brandão Reis

FlyIoT: Arquitetura e Linguagem para o desenvolvimento de aplicações genuinamente IoT

Juiz de Fora

2019

Thomás Marques Brandão Reis

FlyIoT: Arquitetura e Linguagem para o desenvolvimento de aplicações genuinamente IoT

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Juiz de Fora, na área de concentração de Sistemas Multimídia em 5 de setembro de 2019, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Orientador: Marcelo Ferreira Moreno

Juiz de Fora

2019

Ficha catalográfica elaborada através do programa de geração
automática da Biblioteca Universitária da UFJF,
com os dados fornecidos pelo(a) autor(a)

Marques Brandão Reis, Thomás.
FlyIoT: Arquitetura e Linguagem para o desenvolvimento de aplicações genuinamente IoT / Thomás Marques Brandão Reis. -- 2019.
85 f.

Orientador: Marcelo Ferreira Moreno
Dissertação (mestrado acadêmico) - Universidade Federal de Juiz de Fora, Instituto de Ciências Exatas. Programa de Pós Graduação em Ciência da Computação, 2019.

1. Arquitetura. 2. IoT. 3. Middleware. 4. Linguagem declarativa. 5. Abstração de recursos. I. Ferreira Moreno, Marcelo, orient. II. Título.

Thomás Marques Brandão Reis

"FlyIoT: Arquitetura e Linguagem para o Desenvolvimento de Aplicações Genuinamente IoT"

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do grau de Mestre.

Aprovada em 05 de setembro de 2019.

BANCA EXAMINADORA

Prof. Dr. Marcelo Ferreira Moreno – Orientador
Universidade Federal de Juiz de Fora

Prof. Dr. Mario Antonio Ribeiro Dantas
Universidade Federal de Juiz de Fora

Prof. Dr. Cássio Vinicius Serafim Prazeres
Universidade Federal da Bahia

Prof. Dr. Romualdo Monteiro de Resende Costa
Centro de Ensino Superior de Juiz de Fora

AGRADECIMENTOS

Agradeço aos meus pais pelo apoio e confiança nas minhas capacidades e empenho em enfrentar os desafios e medos. Agradeço, também, especialmente a minha esposa Katiane, pela paciência, por acreditar, me apoiar e estar sempre ao meu lado durante minha caminhada em busca dos meus objetivos.

Sou grato pelos ensinamentos e dedicação dos professores que encontrei nessa jornada, em especial ao meu orientador, Marcelo Ferreira Moreno, que sempre se mostrou disponível, paciente e presente no desenvolvimento deste trabalho. Agradeço também ao professor Stanley Teixeira pela dedicação em ajudar na revisão do presente trabalho.

Por fim, mas não menos importante, aos meus colegas do laboratório LApIC pelas eventuais ajudas nas resoluções de problemas encontrados por mim no desenvolvimento deste trabalho.

"Faça o teu melhor, na condição que você tem,
enquanto você não tem condições melhores,
para fazer melhor ainda."
Mário Sergio Cortella

RESUMO

A crescente evolução dos dispositivos móveis e a variedade de novos sensores e atuadores permitem que as aplicações alcancem um potencial ainda maior em um ambiente de Internet das Coisas. No entanto, os desenvolvedores de aplicações IoT enfrentam a fragmentação do mercado, soluções proprietárias e falta de interoperabilidade, não apenas em hardware, mas também em ferramentas de desenvolvimento, além da necessidade de possuírem certo conhecimento para configuração manual das "coisas" do ambiente. Além disso, a maioria das plataformas de IoT se concentra principalmente na aquisição e processamento de dados. Com uma visão de que os recursos de IoT devem ser vistos como uma única máquina, independentemente de sua natureza, tecnologias subjacentes e distribuição geográfica, nesta dissertação é proposta uma arquitetura de middleware distribuída e uma linguagem declarativa para a construção de aplicações IoT. Introduzindo conceitos pouco explorados em propostas semelhantes, este esforço representa um passo em direção à concepção de descrições e desenvolvimento de alto nível de aplicações genuinamente IoT.

Palavras-chave: Internet of things. Middleware. Declarative Language. Resource Orchestration. Distributed System.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

IoT	Internet of Things
FlyIoT	Friendly Internet of Things
FlyIoTL	Friendly Internet of Things Language
DSL	Domain-specific Language
SDDL	Scalable Data Distribution Layer
RFID	Radio Frequency Identification
M-Hub	Mobile Hub
FreeRTOS	Real Time Operating System
DevOps	Development Operations
WPAN	Wireless Personal Area Network
6lowpan	IPv6 over Low power Wireless Personal Area Networks
RPL	Routing Protocol for Low power and Lossy Networks
CoAP	Constrained Application Protocol

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos, o crescimento do poder de processamento e a variedade de dispositivos capazes de se comunicarem entre si têm propiciado um ambiente computacional cada vez mais ubíquo. Dispositivos como *smartphones*, sensores e atuadores trazem consigo a possibilidade de tornar os ambientes mais integrados e potencialmente mais inteligentes (PENG; DHAINI; HO, 2018) (SOLUTIONS, 2015). A integração desses dispositivos acaba por viabilizar a extração de conhecimento importante para diversos setores, como por exemplo, a Indústria 4.0 (XU; XU; LI, 2018), *Healthcare* (MIRANDA, 2019), *Smart Cities* (ARASTEH et al., 2016), entre outros. Diante disso, torna-se cada vez mais comum o uso de tais dispositivos quando a finalidade é a extração de conhecimentos a partir da aquisição, monitoramento e processamento de grande quantidade de dados (WOLLSCHLAEGER; SAUTER; JASPERNEITE, 2017).

A Internet das Coisas (IoT) já é um termo utilizado há algum tempo que, segundo (BERGSTRA; MIDDELBURG, 2003), permite que pessoas e "coisas" sejam conectadas a qualquer hora, em qualquer lugar, com qualquer coisa e qualquer um, idealmente através de qualquer rede e qualquer serviço. Devido à grande heterogeneidade de dispositivos disponíveis atualmente, essa comunicação entre quaisquer "coisas" tende a não ser trivial. Diversos problemas relacionados à integração de sensores, atuadores e dispositivos móveis no domínio de aplicações IoT vêm sendo estudados na literatura. Ao passo em que várias plataformas surgem para a criação e implantação de aplicações IoT, abrindo espaço para inovações no meio, surge também uma grande dificuldade para os desenvolvedores de aplicações realmente distribuídas. Os desenvolvedores se veem obrigados a lidar com a fragmentação dessas plataformas e os tipos de dispositivos disponíveis, questões de conectividade, sincronismo, entrega e controle do ambiente, ao invés de focar apenas no desenvolvimento das funcionalidades de suas aplicações.

Algumas plataformas IoT (XIVELY..., s.d.) e linguagens DSL (*Domain Specific Language*) (NEGASH et al., 2017) têm facilitado em parte tais questões, mas ainda há certa dependência de informações de baixo nível a serem providas manualmente. De fato, um dos principais desafios está na possibilidade de oferecer ao desenvolvedor os recursos e as capacidades presentes nos dispositivos de forma transparente, sem aumentar a complexidade do desenvolvimento de aplicações.

Tal ambiente heterogêneo e complexo incentiva o surgimento de soluções de *middlewares* para permitir que as aplicações dos desenvolvedores usufruam dos recursos que os sensores e atuadores oferecem e atendam aos requisitos necessários em ambientes IoT. Um *middleware* é, em sua essência, uma camada de software que provê recursos e abstrações além dos oferecidos pelo sistema operacional para as aplicações de software. Normalmente, um *middleware* oculta boa parte da complexidade inerente ao gerenciamento de recursos

de mais baixo nível. Atualmente, existem diversas soluções de *middleware* para ambientes IoT na literatura (RAZZAQUE et al., 2016), porém, usualmente, são específicas para o domínio de um problema, não sendo tão genéricas ou abrangentes, quanto poderia ser necessário para o ambiente IoT. Além disso, várias soluções requerem que o ambiente envie informações para serviços Web ou nuvem pré-estabelecidos, não permitindo a flexibilização de implantações de Internet das coisas em seu sentido mais amplo, genuinamente distribuído, capazes de incluir sensores, atuadores, e outros dispositivos dispersos livremente em redes de comunicação. Outras propostas porém, utilizam a abordagem de *Fog Computing* (BONOMI et al., 2012), onde se estende o paradigma de nuvem, e a alocação recursos (processamento, armazenamento, entre outros) é trazida para a borda da rede a fim de otimizar o seu acesso de acordo com as decisões que mais fazem sentido para a aplicação.

Neste trabalho, atribui-se o termo "aplicação genuinamente IoT" à aplicação que não pode depender, necessariamente, de recursos em nuvem para sua execução. Ou seja, a aplicação se distribui pelo uso de recursos localizados em qualquer parte alcançável de uma rede de comunicação. Para isso, a aplicação precisa poder ser projetada e desenvolvida sem haver necessariamente uma preocupação com questões intrínsecas ao meio em que será executa, como por exemplo, questões de conectividade, comunicação, protocolos e descoberta de recursos, entre outros. Propostas que se baseiam em *Fog Computing* também podem se aproximar desse conceito se, em sua definição, isentar as aplicações dessas questões mais específicas e inerentes ao meio em que são executadas. Porém essa abordagem mais generalista é pouco explorada na literatura, visto que boa parte das propostas visa solucionar um problema real e geralmente específico.

Com o propósito de contribuir com o processo de concepção e desenvolvimento de aplicações genuinamente IoT, ou seja, aplicações construídas independente, a priori, do meio em que serão implantadas, o presente trabalho introduz uma arquitetura de *middleware* denominado FlyIoT (**F**riend**ly** **I**nternet **o**f **T**hings) e uma linguagem declarativa denominada FlyIoTL (**F**riend**ly** **I**nternet **o**f **T**hings **L**anguage). Com tal proposta é possível elevar o nível de abstração dos recursos para o desenvolvedor, eximindo-o de lidar com questões de preparação do meio, conectividade ou descoberta.

A arquitetura FlyIoT proposta neste trabalho tem como objetivo proporcionar para uma abordagem de construção de *middlewares* que suporte a abstração de complexidades e ofereça para as aplicações, em um nível mais alto, os recursos em um ambiente genuinamente IoT. FlyIoT lida com a descoberta e gerenciamento de recursos, em um processo de orquestração distribuída, no qual os próprios componentes que definem o FlyIoT também se inserem dinamicamente. Para que tal orquestração seja possível, a arquitetura de FlyIoT define componentes fracamente acoplados e instanciáveis em dispositivos locais, servidores Web ou serviços em nuvem, para a provisão de suas respectivas funcionalidades. Além disso, a arquitetura também considera que uma instância de *middleware* deve ser adaptável

a novos componentes de serviços para que seja possível sua aplicação em propósitos mais específicos.

O maior esforço deste trabalho está concentrado na arquitetura FlyIoT, que representa um passo intermediário rumo à definição de uma linguagem capaz de descrever aplicações IoT em alto nível, não restrita apenas a ambientes específicos. A arquitetura parte do pressuposto de que um ambiente IoT pode ser visto como uma máquina, que possui recursos de entrada (sensores), de saída (atuadores) e inclui abstrações para processadores de instruções e volumes de armazenamento. Dessa forma, aplicações genuinamente IoT podem ser mais facilmente concebidas, uma vez que os processos de descoberta e orquestração passam a lidar não só com a distribuição, sensoriamento e atuação da aplicação, mas também com recursos como processamento e armazenamento. Por extensão, instâncias de *middleware* baseadas na arquitetura FlyIoT podem se valer de tais abstrações ao distribuir seus componentes de serviços para gerenciamento. Por fim, é importante destacar que FlyIoT permite também que as aplicações sejam notificadas acerca de eventos observados por FlyIoT de maneira simplificada.

A linguagem FlyIoTL tem o propósito de reduzir a complexidade do desenvolvimento de aplicações IoT nesses ambientes de dispositivos tão heterogêneos aliada aos conceitos especificados pela arquitetura FlyIoT. A máquina de apresentação da linguagem FlyIoTL se apoia nos mecanismos de descoberta, comunicação, gerência de dados, distribuição e orientação a eventos especificados pela arquitetura FlyIoT, de forma a tornar transparente aos desenvolvedores boa parte da complexidade de manipulação típica de ambientes IoT. A linguagem FlyIoTL permite definir as “coisas”, identificando informações obrigatórias e opcionais, além de determinar os requisitos e códigos a serem processados em momentos determinados pela própria linguagem. Além disso, ela permite que o desenvolvedor especifique relacionamentos de eventos e ações encadeadas a fim de estabelecer um fluxo de tarefas a serem executadas pelas ”coisas“ no ambiente em que um *middleware* baseado em FlyIoT é instanciado. Tais eventos podem ser de observação de valores de sensores ou até mesmo gatilhos para outros fluxos de tarefas, por exemplo.

Para a descrição de tal proposta, esta dissertação está estruturada da seguinte forma: O Capítulo 2 apresenta os trabalhos relacionados; O Capítulo 3 descreve a arquitetura de *middleware* proposta seguida da prova de conceito implementada no Capítulo 4; O Capítulo 5 descreve o modelo da linguagem FlyIoTL seguido da prova de conceito do interpretador no Capítulo 6; O Capítulo 7 apresenta os casos de uso e por fim, o Capítulo 8 apresenta a conclusão e trabalhos futuros.

2 FUNDAMENTOS E TRABALHOS RELACIONADOS

2.1 Middlewares e plataformas IoT

Com o surgimento constante de dispositivos cada vez mais inteligentes, já se verifica a necessidade de definir uma futura arquitetura de rede IoT. Diante disso, (YAQOOB et al., 2017) conduziu seu estudo com foco em tais arquiteturas, categorizando-as sob uma taxonomia baseada em aspectos como aplicações, tecnologias capacitadoras, objetivos de negócios, requisitos arquiteturais, tipos de arquitetura (centralizada e decentralizada) de plataforma IoT e topologias de rede. O trabalho destaca a importância da interoperabilidade, escalabilidade, flexibilidade, eficiência energética, gerenciamento de mobilidade e segurança.

Um ambiente IoT geralmente é composto por componentes e dispositivos heterogêneos e, com isso, o seu controle tende a ser árduo. Neste sentido, os *middlewares* IoT são desenvolvidos a fim de abstrair a complexidade ao lidar com essa gama enorme de dispositivos. O estudo realizado por (RAZZAQUE et al., 2016) levanta não só características de diversos *middlewares* IoT já desenvolvidos, mas também necessidades e pré-requisitos importantes que um *middleware* IoT deve atender.

Diversos *middlewares* IoT, com objetivos distintos, que visam facilitar o controle de dispositivos conectados, podem ser encontrados na literatura. De acordo com (RAZZAQUE et al., 2016), existem requisitos funcionais, não-funcionais e arquiteturais para *middlewares* IoT. Os requisitos funcionais elencados são: descoberta de recursos, gerenciamento de dados, gerenciamento de recursos, gerenciamento de código e gerenciamento de eventos. Já nos requisitos não funcionais se enquadram: escalabilidade, segurança, disponibilidade, confiabilidade e privacidade. Por fim, os requisitos arquiteturais são: abstração de programação, interoperabilidade, ciência de contexto, autonomia, adaptabilidade e, além disso, deve ser leve e orientado a serviços distribuídos.

Dentre os requisitos funcionais, a descoberta de recursos deve suportar dispositivos de hardware heterogêneos (e.g., tags RFID, sensores, atuadores, smartphones), obtendo os mesmos metadados que incluem suas capacidades energéticas, de processamento e de memória, tipos de serviços fornecidos, informações do nível de rede e infraestrutura. Os mecanismos de descoberta precisam ser bem dimensionados e deve haver uma distribuição eficiente da carga de descoberta, dada a composição de dispositivos que geralmente são limitados em recursos. Devido à estrutura dinâmica e escalável de infraestruturas IoT, abordagens centralizadas de descoberta de recursos são muitas vezes ineficazes, ao mesmo tempo em que pode ser complexa a decisão entre abordagens puramente distribuídas e híbridas. (TEIXEIRA et al., 2011) e (NEDOS et al., 2009) demonstram como é possível um *middleware* ser escalável, porém suas soluções podem não funcionar bem para aplicações que exigem garantia de descoberta de alta precisão, como aplicações de missão crítica.

Isso se dá devido ao fato de que suas soluções se baseiam em redes oportunistas, ou seja, funcionam bem para aplicações que não necessitam de algum grau de sincronismo de mensagens entre os nós da rede. Cada nó obedece a certo tipo de protocolo para repassar sua mensagem aos demais até atingir o destino. No caso de redes onde existe certa quantidade de nós em movimento, esse tipo de solução escala de maneira satisfatória dependendo do protocolo utilizado, porém é prejudicada na velocidade do envio das informações. Exemplos desse tipo de solução podem ser encontrados em redes de sensores sem fio com foco em coleta de dados para análise postergada, onde não será problemático se as informações chegarem ao destino com certo atraso.

A proposta apresentada por (TALAVERA et al., 2015) demonstra um sistema para descoberta de dispositivos e recursos com foco em ambientes IoMT (Internet of Mobile Things). Em um ambiente desse tipo, os dispositivos não são necessariamente estáticos e, por isso, é necessária uma abordagem mais oportunista de comunicação entre eles. O serviço de *middleware* apresentado por (TALAVERA et al., 2015), chamado M-Hub, é um serviço de propósito geral que descobre, registra e permite comunicação remota em modo unicast, broadcast e em grupo com outros dispositivos. O M-Hub é baseado no *middleware* Scalable Data Distribution Layer (SDDL) (DAVID et al., 2013) e preenche uma lacuna que o SDDL Core deixava entre a conexão com a Internet e conexões sem fio de curto alcance dos dispositivos – que podem ser sensores e atuadores muito restritos, incorporados a outros dispositivos com capacidades significativas de processamento e armazenamento. Tais dispositivos são chamados de M-OBJs (Kind of Mobille Objects) que se conectam a partir de uma API de detecção, presença e atuação de curto alcance chamado S2PA, a qual possui uma interface comum para diferentes tecnologias sem fio de baixo alcance (WPAN - Wireless Personal Area Network) dentro do M-Hub. Apesar da proposta abrangente apresentada por (TALAVERA et al., 2015), em M-Hub cada dispositivo móvel se comporta como os demais e não existe, no *middleware*, distinções de capacidades dos mesmos para alocação de tarefas, ou seja, todos os dispositivos se comportam como agregadores e controladores de outros dispositivos mais simples, como sensores e atuadores.

Os dispositivos IoT, em sua grande maioria, são limitados em questão de processamento, memória, armazenamento e eficiência energética. Diante disso, trabalhos extensos como o de (MUSADDIQ et al., 2018) buscam realizar o levantamento de pontos importantes no uso dos recursos desses dispositivos por sistemas operacionais com foco em IoT. Alguns desses sistemas bastante conhecidos são: *FreeRTOS* (GAY, 2018), *TinyOS* (AMJAD et al., 2016) e *Contiki* (ZIKRIA et al., 2018). O FreeRTOS foi projetado para ser pequeno e simples. Prova disso é seu kernel, que consiste de apenas três arquivos C. Ele possui também métodos para lidar com várias *threads*, *mutex*, semáforos e cronômetros de *software*. Esse tipo de sistema é geralmente utilizado em projetos embarcados, onde a prioridade é ser leve e rápido. O TinyOS também é um sistema muito leve e exige apenas

8KB de memória, além de ser bastante flexível. Possui um foco em redes de sensores sem fio que exigem certo tipo de segurança. Já o Contiki é um sistema operacional de código aberto que conecta minúsculos microcontroladores com foco em baixo custo e baixo consumo de energia, além de dar suporte a padrões de protocolos com IPv4, IPv6, 6lowpan, RPL e CoAP. De fato, percebe-se um grande esforço em estudar as vantagens e facilidades do uso de tais sistemas operacionais dedicados para abstrair características dos recursos mais limitados de ambientes IoT, porém ainda há certa dependência de conhecimentos avançados de programação por parte dos seus usuários. Além disso, desenvolvedores com o propósito de criar novas soluções IoT – não necessariamente atreladas simplesmente a coleta, agregação e análise de grande quantidade de dados na nuvem – enfrentariam um desafio ainda maior em adaptar tais sistemas.

Além de recursos como processamento, memória e armazenamento, é importante ressaltar o gerenciamento dos recursos que os dispositivos podem prover à rede IoT, monitorando e garantindo que eles sejam alocados ou provisionados de maneira justa, além de resolver possíveis conflitos no seu uso. Em arquiteturas IoT, principalmente naquelas orientadas a serviços ou máquinas virtuais, o *middleware* precisa facilitar a composição (ou recomposição) de recursos (ou serviços) de forma espontânea para satisfazer as necessidades da aplicação. O ubiSOAP (CAPORUSCIO; RAVERDY; ISSARNY, 2012) é um *middleware* que se destacou no estudo realizado por (RAZZAQUE et al., 2016), pois explora diversas tecnologias de rede a fim de criar um ambiente multi-radio integrado, oferecendo conectividade independente de rede aos serviços. Ele permite, em sua arquitetura, a abstração de recursos unificada para dispositivos mais simples, além de dar suporte à autenticação, que é uma preocupação de muitas aplicações IoT. Contudo, o *ubiSOAP* não possui suporte a questões de segurança e privacidade. Outros *middlewares* orientados a serviços bem conhecidos em ambientes IoT são: o Sirena (BOHN; BOBEK; GOLATOWSKI, 2006), Cosmos (KIM; LEE et al., 2008), Socrades (DE SOUZA et al., 2008) e Hydra (EISENHAUER; ROSENGREN; ANTOLIN, 2010). Diferentemente do *ubiSOAP*, estes possuem certa preocupação com o quesito segurança, embora ainda sejam deficientes em questões de privacidade (DHAS; JEYANTHI, 2019).

Outra característica importante é o gerenciamento de dados, ponto fundamental para aplicações IoT. Esses dados representam informações que foram adquiridas, processadas e armazenadas a partir dos dispositivos ou infra-estrutura de rede e que atendam a uma demanda requisitada pela aplicação. Uma grande quantidade de dados brutos coletados continuamente precisa ser convertida, por meio de filtragem e agregações, em conhecimento útil para a aplicação. Além disso, ao escolher bancos de dados para o gerenciamento de dados em sistemas IoT, é importante visar escalabilidade, capacidade de lidar com alta taxa de operações, flexibilidade de esquemas do banco de dados, integração com ferramentas de análises e custos (CRUZ HUACARPUMA et al., 2017).

Xively (XIVELY..., s.d.)(SINHA; PUJITHA; ALEX, 2015) é uma plataforma de *middleware* que permite a integração de sensores na nuvem e também permite que se desenvolva mecanismos e regras para interpretação dos dados coletados. A plataforma Xively, adquirida pela empresa Google, tinha como foco auxiliar empresas e usuários na conexão de dispositivos para ambientes IoT. No nível de dispositivo, a plataforma ajuda a gerenciar a comunicação, reforça práticas de segurança e fornece ferramenta para gerenciamento de credenciais e atualizações de firmware. Aplicativos móveis e web podem usar o recurso de mensagens e gerenciamento da plataforma para provisionar e se comunicar com suas "coisas". Apesar de a integração de sensores ser de simples utilização, a interação com sua API (*Application Programming Interfaces*) pode não ser trivial, principalmente se o sensor não for suportado pela plataforma. Além disso, não existe garantia de que os dados e informações fiquem restritos ao ambiente IoT implantado, podendo haver riscos quanto a posse ou privacidade dos dados persistidos na plataforma. O OpenIoT (KIM; LEE, 2014a) é um *framework* e plataforma de código aberto um pouco mais abrangente, que dá suporte ao desenvolvedor hardware e de software, provedor de serviço e usuários desses serviços. Possui plataformas de *Mashup* para o gerenciamento de aplicações registradas e permite que os administradores operem a partir dela. O OpenIoT permite também o uso de sensores virtuais, cunhando o termo Sensoriamento-as-a-Service, pois inclui uma estrutura semântica, valendo-se de ontologias, além de garantir interoperabilidade entre sistemas externos. Dependendo da experiência do desenvolvedor, pode haver dificuldades de implantação, pois tanto o Xively quanto o OpenIoT exigem certo processo manual de configuração para que as aplicações integrem os sensores. O Resthing (QIN et al., 2011) é um *middleware* IoT orientado a serviços. Ele, assim como a solução proposta neste trabalho, também utiliza uma camada que oferece serviços por meio de uma API RESTful para os recursos em uma rede de sensores com suporte a abstração. O *middleware* oferece uma camada de serviços e uma de adaptação com o objetivo de abstrair os recursos IoT. Na descrição do *middleware*, é demonstrado apenas um serviço implementado pelo autor – que provavelmente não suporta implantação distribuída – sem especificar a possibilidade de orquestração de seus serviços.

Em aplicações IoT, segundo (RAZZAQUE et al., 2016) há potencialmente um grande número de eventos que são gerados e devem ser gerenciados pelo *middleware*. O gerenciador de eventos é responsável por mapear eventos simples observados em eventos significativos para a aplicação. O grande número de eventos é gerado de forma proativa e reativa em IoT. Por isso, é de se esperar que os componentes de *middleware* se tornem gargalos no sistema.

Por fim, o último requisito funcional é o gerenciamento de código, cuja implantação em um ambiente IoT é desafiador. Tornar um código portátil entre os nós da rede não é trivial, pois necessita um controle minucioso por parte do *middleware*, que deve estar ciente de quais dispositivos devem receber e executar determinado código, além de, é claro,

ter certeza que o dispositivo é capaz de realizar tal tarefa.

Recentemente, plataformas com abordagens baseadas em microsserviços (THÖNES, 2015) vêm sendo utilizadas para solucionar problemas em IoT. Um microsserviço geralmente é construído com o objetivo de minimizar e isolar a complexidade de cada um deles de acordo com sua função. Quanto menor o microsserviço, mais simples espera-se que ele seja, porém deve-se atentar para que a distribuição desses serviços não gere outros problemas que antes não existiriam em um serviço monolítico. Trabalhos como (SANTANA; MELLO ALENCAR; PRAZERES, 2019), (KRYLOVSKIY; JAHN; PATTI, 2015) e (SUN; LI; MEMON, 2017) utilizam-se de microsserviços implantados para registar e consumir dados gerados a partir de sensores e atuadores. Trazem implementações mais vantajosas se comparadas a implementações monolíticas e puramente centralizadas, pois permitem maior adaptabilidade, escalabilidade e interoperabilidade. O trabalho (SANTANA; MELLO ALENCAR; PRAZERES, 2019) se destaca por usar uma abordagem de microsserviços reativos, ou seja, possui fluxo de comunicação desacoplado no tempo (permitindo a simultaneidade) e espaço (permitindo mobilidade e distribuição). Operações assíncronas e não-bloqueantes reduzem o congestionamento de recursos no sistema e produzem escalabilidade e baixa latência. Além disso, comunicação baseada em mensagens assíncronas caracteriza os sistemas como sendo mais resilientes (capacidade de lidar com falhas) e com mais elasticidade (capacidade de se dimensionar horizontalmente). O autor estendeu a proposta do SOFT-IoT (PRAZERES; SERRANO, 2016) inserindo o conceito de microsserviços reativos o que tornou a sua implementação significativamente melhor que a anterior, além de usar para sua implementação softwares robustos como o *Kubernetes* (HIGHTOWER; BURNS; BEDA, 2017).

A arquitetura FlyIoT proposta nesse trabalho preenche boa parte dos requisitos apontados por (RAZZAQUE et al., 2016). A Tabela 1 apresenta um comparativo resumido de algumas características de *middlewares* estudados que se destacaram, frente à proposta de arquitetura FlyIoT. FlyIoT ainda se encontra em um nível arquitetural com uma prova de conceito para vários recursos importantes implementados. No entanto, estudos quantitativos frente aos outros trabalhos precisam ser conduzidos a fundo no futuro para uma comparação mais objetiva. Na verdade, essa comparação preliminar permite que a especificação da arquitetura seja influenciada por características desejáveis que o middleware deve possuir. Assim, a arquitetura é projetada de modo a favorecer as características apontadas na Tabela 1.

Os requisitos de **Gerenciamento de recursos**, **Descoberta de recursos** e **Gerenciamento de Dados** são contemplados pelos componentes de serviço de FlyIoT. Além disso, pelo fato de os módulos de serviços de FlyIoT possuírem fraco acoplamento entre si e preferencialmente sem estado, eles podem ser instanciados mais facilmente caso alguma falha ocorra ou seja necessário substituir por outras instâncias. Isso faz com

Middlewares	Descoberta de recursos	Gerenciamento de recursos	Gerenciamento de dados	Orquestração de serviços	Abstração de recursos
PRISMA(SILVA; DELICATO et al., 2014)	C,D	M	A,PA	SI	Q
ubiSOAP(CAPORUSCIO; RAVERDY; ISSARNY, 2012)	D,Di,R	A,M,CA,CL	SI	SI	C
MOSDEN(PERERA et al., 2014)	D,Di,S	A,M,CP	A,PA	SI	Q
Echelon(ADESTO, s.d.)	D,Di,S	A,M	A,PA	SI	Q
Xively(XIVELY..., s.d.)	D,Di,S	A,M	A,PA	SI	SI
M-Hub(TALAVERA et al., 2015)	D,Di	A,C	A	S	Q
OpenIoT(KIM; LEE, 2014a)	C,Di,S	A,M,CP	A,PA	SI	SI
Resthing(QIN et al., 2011)	C,Di	A,M,CP	A,PA	N	Q
SOFT-IoT Ext.(SANTANA; MELLO ALENCAR; PRAZERES, 2019)	C,S	M,A	A,PA	SI	SI
FlyIoT	D,C,Di,S	A,M,CA,CP	A,PA,PF	S	C,Q
Legenda	C=centralizada D=distribuída Di=de dispositivos R=de rede S=de serviços	A=alocação M=monitoramento C=composição (A=adaptativa, P=predefinida) CL=gerenc. de conflito	A=armazenamento P=processamento (A=agregação, C=compressão, F=filtragem)	S=sim N=não SI=sem informação	C=computacionais Q=qualitativos SI=sem informação

Tabela 1 – Comparativo entre propostas de *middleware* na literatura

que um *middleware* baseado em FlyIoT atenda mais facilmente também ao requisito de **disponibilidade**. O requisito **abstração de programação** também é atendido, pois o *middleware* baseado em FlyIoT estabelece uma orquestração e descoberta dos seus próprios serviços a fim de isentar a aplicação de lidar com tais peculiaridades. FlyIoT se mostra uma arquitetura híbrida entre **Orientada a Serviços** e **Orientada a eventos**, onde cada um dos componentes é representado por um conjunto de serviços e que, aliado aos controladores dos recursos IoT, pode também permitir a manipulação por meio de monitoramento de eventos. Um destaque a ser mencionado é o requisito de **abstração de recursos**, onde toda a heterogeneidade de recursos em ambientes IoT é reduzida a apenas 3 tipos: **entrada** (*input*), **saída** (*output*) e **processadores** (*processors*). Além disso, segundo (RAZZAQUE et al., 2016), nenhum dos *middlewares* em seu estudo lidou com gerenciamento de código de maneira eficiente. Com isso, o uso da abordagem em que a arquitetura FlyIoT categoriza recursos do tipo *Processors* estabelece um caminho promissor para preencher o requisito de **gerenciamento de código**, visto que é possível um *middleware* identificar as capacidades de um processador antes de ordenar a execução de *scripts* de código ou programas.

Traçando um paralelo em relação à arquitetura proposta por (QIN et al., 2011), FlyIoT também oferece uma camada de serviços e outra de adaptação de recursos IoT. Mas, diferentemente da arquitetura de (QIN et al., 2011), FlyIoT incrementa algumas funcionalidades como a capacidade de **descoberta de recursos e serviços** de maneira distribuída. FlyIoT se vale dessa funcionalidade de descoberta de recursos e suas capacidades para instanciar seus próprios componentes de serviço.

(TALAVERA et al., 2015) apresenta uma solução que permite que dispositivos móveis se tornem agregadores de outros sensores. Diferente da solução proposta por (TALAVERA et al., 2015), FlyIoT permite a **distinção das capacidade dos recursos** descobertos além da especificação de ações e eventos a serem observados pelas aplicações. Isto possibilita desenvolver uma aplicação mais flexível a diferentes necessidades de acordo com o domínio em que FlyIoT for instanciada.

Assim como a solução apresentada por (SANTANA; MELLO ALENCAR; PRA-

ZERES, 2019), FlyIoT se alinha aos conceitos de microsserviços, em que cada componente de serviço deve, preferencialmente, ser leve e fácil de se instanciar. Além de sensores e atuadores, FlyIoT também considera o uso de processadores e volumes de armazenamento. Isso torna possível definir ou, pelo menos exemplificar, **recursos virtuais** como parte dos elementos aptos a serem manipulados pelas aplicações IoT.

Plataformas IoT como (XIVELY. . . , s.d.) e (KIM; LEE, 2014a) geralmente possuem uma certa abstração na configuração de sensores ou atuadores por parte do desenvolvedor. Porém, acabam por exigir certa configuração manual, além de ser quase sempre obrigatório enviar os dados a um serviço de nuvem. Novas integrações de sensores ou atuadores, inicialmente não suportados por tais plataformas, podem se tornar impossíveis devido a sua natureza mais rígida de configuração. Em FlyIoT, **não é obrigatório que os dados sejam enviados a um serviço de nuvem**.

2.2 Linguagens de programação

Quando se trata do desenvolvimento de aplicações, normalmente existem vários paradigmas de linguagem de programação, sendo o imperativo e o declarativo dois dos principais. O uso de uma linguagem de programação imperativa (ou procedural) consiste em especificar de forma estruturada cada passo que um programa deve seguir para alcançar um objetivo ou estado desejado (FREEMAN-BENSON, 1992). Esse paradigma imperativo se destaca pela sua eficiência, abrangência e modelagem mais adequada à diversidade de problemas do mundo real. Porém, se ele não for utilizado seguindo padrões de projeto sólidos à medida que o programa cresce, sua legibilidade e entendimento podem se tornar bem complexos. Muitas linguagens de propósito geral, como C++, Java, Python e Javascript seguem essa abordagem imperativa e são bastante utilizadas para solucionar os mais diversos problemas.

Por outro lado, quando se trata de resolução de problemas específicos, o paradigma imperativo pode não ser o mais adequado, devido a sua alta complexidade de desenvolvimento. Um exemplo disso são soluções para aplicações multimídia que podem ser mais eficientes utilizando uma abordagem declarativa, em que o autor ou desenvolvedor foca principalmente em "o que" precisa ser feito e não em "como" precisa ser feito. O paradigma declarativo é baseado em programação funcional, programação lógica ou programação restritiva. Linguagens declarativas normalmente são mais limitadas, do ponto de vista de lidar com problemas mais gerais, porém possuem um foco maior na solução de problemas para o domínio ao qual foram destinadas. Alguns exemplos de DSL (Domain Specific Language) declarativas conhecidas incluem SQL (*Structured Query Language*), HTML (*Hyper Text Markup Language*) e UML (*Unified Modeling Language*).

Em ambientes multimídia, algumas linguagens se destacam pela simplicidade e a possibilidade de lidarem com objetos de mídia, como SMIL (BULTERMAN; RUTLEDGE,

2008), NCL (SOARES; RODRIGUES, 2006) e STorML (FREESZ JR.; YUNG; MORENO, 2017). Tais linguagens são baseadas em XML (*Extensible Markup Language*) (BRAY et al., 2006) possuem uma estrutura flexível e adaptável. Essas linguagens possuem conceitos bem eficientes no que diz respeito ao relacionamento entre tais objetos e não precisam necessariamente ser apenas declarativas ou híbridas, permitindo que interajam com *scripts* imperativos, como é o caso da linguagem NCL (SOARES; RODRIGUES; MORENO, 2007). A linguagem NCL possui foco em ambientes de TV Digital e IPTV. Também é possível criar ações a partir de eventos gerados de acordo com os estados observados em cada objeto de mídia. Todo elemento que interage dentro de uma aplicação NCL é considerado uma mídia, ou seja, imagem, vídeo, áudio, scripts, etc., permitindo certa homogenização de seus componentes capazes de interagir, o que facilita o controle por parte da máquina de apresentação da linguagem. Já a linguagem STorML possui foco em ambientes de Mídia Digital *out of home* (DOOH), com ênfase em propagandas externas e anúncios publicitários. Sendo baseada em NCL e SMIL, a STorML partilha das mesmas características de relacionamento de eventos e trilhas para exibição de conteúdo multimídia. Apesar dos diversos recursos oferecidos por tais linguagens, elas não suportam ambientes IoT distribuídos. Existem trabalhos, entre os quais se destaca o de (BATISTA, 2013), para complementar linguagens como NCL e dar suporte a multi-dispositivos, mas acabam por inserir mais complexidades com as quais o autor terá que lidar no desenvolvimento das aplicações.

Para ambientes IoT, existem algumas DSL e *frameworks* que abstraem ou facilitam parte do desenvolvimento dos componentes e recursos, como por exemplo (NEGASH et al., 2017) e (GOMES et al., 2017). A DoS-IL apresenta características comuns a um desenvolvedor, como palavras chaves *"while"*, *"if"*, *"else"*, operadores lógicos, elementos como *"device"*, *"sensor"*, *"actuator"*, *"event"* e *"resource"*. A escrita da linguagem segue o paradigma imperativo e é bastante leve para ser executada por sua arquitetura. Mas tal linguagem abstrai apenas parte dessa complexidade e ainda requer que os autores e desenvolvedores lidem com a complexidade relacionada à comunicação, descoberta e configuração desses ambientes, além de exigir que o desenvolvedor já possua experiência com programação, uma vez que há grande proximidade com linguagens como C/C++. A linguagem apresentada por (GOMES et al., 2017) possui uma estrutura híbrida em que parte de composição é escrita num documento XML de maneira declarativa. Alguns componentes de configuração de rede são repassados por meio de uma definição similar a um documento de lista de itens. Também é possível inserir condicionais tradicionais como *"if"* e *"else"*. Assim como na primeiro caso, o desenvolvimento de aplicação usando essa abordagem pode se tornar árduo para quem tem pouca experiência, além de ser necessário lidar com configurações de protocolos de rede pelo seu arquivo de configuração.

Existem também plataformas que facilitam a integração de recursos IoT, como por exemplo: OpenIoT (KIM; LEE, 2014b), Xively (SINHA; PUJITHA; ALEX, 2015),

Ditto (FOUNDATION, 2019a), Hono (FOUNDATION, 2019b). O Ditto e o Hono são ferramentas que estão sob a organização Eclipse, que introduz o conceito de Device-as-a-Service, que expõe as características sobre as capacidades de um dispositivo como sendo um serviço. Porém, a maioria dessas plataformas acaba por limitar o desenvolvimento no que diz respeito à interação entre as "coisas". Muitas delas focam principalmente no cenário de aquisição e processamento de dados oriundos de sensores ou no controle de atuadores baseado no conhecimento desses dados processados na nuvem. São deficientes quando é necessário realizar relacionamentos mais complexos entre suas *"coisas"* e descobrir novos recursos IoT.

Diante dessas questões, a linguagem FlyIoTL se apresenta como uma abordagem para manipulação de dispositivos IoT em um nível de abstração mais elevado. Sem que o desenvolvedor tenha que se preocupar, a priori, com o meio em que está sendo executada. Linguagens como as que (GOMES et al., 2017) e (NEGASH et al., 2017) permitem a abstração de parte da complexidade envolvida com o uso de documentos declarativos XML ou imperativos com elementos mais próximos do vocabulário usado por um desenvolvedor. Porém, tais linguagens dependem de que o ambiente IoT seja controlado e configurado pelo desenvolvedor e, ainda, que se tenha certa experiência com ambientes IoT. A linguagem FlyIoTL, aliada aos conceitos especificados pela arquitetura FlyIoT, possibilita que o ambiente IoT seja preparado automaticamente e então seus *things* se tornem disponíveis para que a aplicação FlyIoTL manipule em um nível mais elevado de abstração e de maneira declarativa. Por ser majoritariamente declarativa, FlyIoTL permite também **inserção de (ou referência) *scripts* de código** para adaptar alguma funcionalidade não suportada em seu código declarativo. Além disso, FlyIoTL possibilita a **manipulação de *things* virtuais** por meio de recursos físicos encontrados por FlyIoT, como processadores que executam *scripts* de código e volumes de armazenamento que manipulam arquivos.

A linguagem FlyIoTL, assim como as linguagens NCL introduzidas por (SOARES; RODRIGUES, 2006) e (FREESZ JR.; YUNG; MORENO, 2017), torna possível o relacionamento de eventos e ações entre suas entidades. Um evento de um objeto de mídia pode ser observado e relacionado com ações em outros objetos de mídia em uma aplicação NCL ou STorML. De maneira similar, eventos em um *thing* podem ser observados e relacionados com ações de outros *things* em uma aplicação FlyIoTL. Além disso, diferentemente de uma aplicação NCL, uma aplicação FlyIoTL possibilita a **inserção de novos eventos e ações** baseados nas capacidades de tipos dos *things* declarados no documento FlyIoTL. Com isso, a linguagem pode ser expandida, do ponto de vista de capacidades, de acordo com os *things* anunciados por meio de suas entidades físicas descobertas por FlyIoT.

3 ARQUITETURA FLYIOT

Com a popularização de dispositivos computacionais cada vez menores e capazes de se comunicarem entre si, abriu-se espaço para a Internet das Coisas (IoT), composta por dispositivos extremamente heterogêneos como sensores e atuadores, simples ou robustos, dispositivos agregadores de sensores e atuadores (*smartwatches*, *smartphones*, *tablets*), computadores ou até mesmo grandes servidores espalhados pelo planeta compondo os serviços de nuvem. Diversos trabalhos e esforços, como os abordados no Capítulo 2, vêm sendo direcionados para solucionar problemas em cenários que visam integrar e controlar a heterogeneidade dos dispositivos.

Normalmente, os *middlewares* buscam solucionar problemas específicos relacionados ao meio em que são implantados, abstraem parte dessa complexidade e oferecem recursos para as aplicações. A especificidade do ambiente em que são implantados faz com que se tornem limitados a determinados cenários e muitas vezes restritos ao meio em que se inserem. Diante disso, as aplicações IoT que usufruem de tais ambientes acabam se tornando também restritas a esse meio. Diante disso, como é possível oferecer recursos em um nível mais elevado para que aplicações IoT sejam desenvolvidas de maneira mais simplificada sem que o desenvolvedor precise lidar com complexidades intrínsecas ao meio?

A arquitetura geral de *middleware* para ambientes IoT proposta neste trabalho, denominada FlyIoT, segue o paradigma distribuído e foi pensada para possibilitar a visão de recursos de IoT como uma grande máquina distribuída a ser gerenciada para facilitar o desenvolvimento de aplicações. A arquitetura FlyIoT é orientada a serviços, ou seja, é decomposta em componentes de serviços menores, e cada um deve ser implantado de forma independente. Com isso, cada componente de serviço pode possuir dependências tecnológicas distintas. Eventuais mudanças em um deles podem ter um impacto reduzido na implantação e funcionamento de outros componentes da arquitetura. Essa característica distribuída e autocontida dos componentes de serviço de FlyIoT beneficia sua substituição na implantação se necessário. Além disso, a arquitetura FlyIoT especifica certo grau de gerenciamento de eventos dos recursos que foram descobertos pelo *middleware* e, portanto, é híbrida entre uma arquitetura de *middleware* orientada a serviços e orientada a eventos.

A arquitetura FlyIoT é especificada com a premissa de que o *middleware* instanciado realize sua inicialização em uma rede local e possa ser migrado dinamicamente para componentes disponíveis em um serviço de nuvem ou rede em outra geolocalização, não obrigando que o *middleware* seja centralizado ou distribuído. Porém, espera-se que a inicialização do *mdidleware* seja centralizada, tendo em vista o objetivo de atender a uma aplicação. Posteriormente seus componentes poderão se tornar distribuídos da maneira mais adequada para o ambiente em questão. O seu comportamento pode se alterar de acordo com as necessidades da aplicação ou políticas de distribuição do *middleware*,

e portanto, não serão tratadas neste trabalho, cujo foco é a arquitetura geral e seus componentes principais. A Figura 1 ilustra a arquitetura FlyIoT.

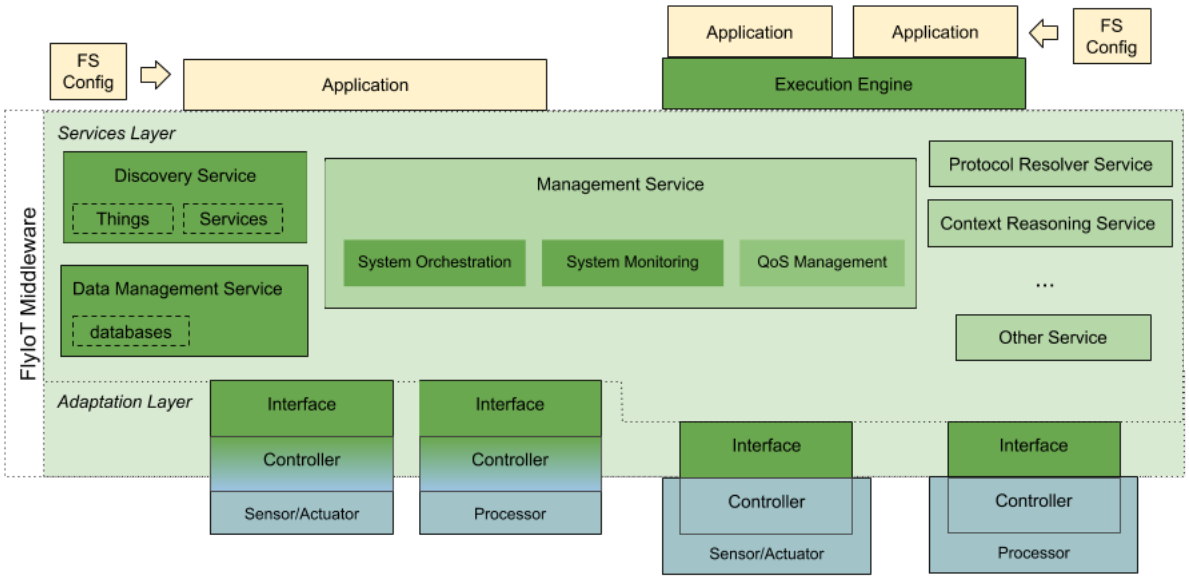

Figura 1 – Arquitetura FlyIoT

FlyIoT oferece uma camada de comunicação abstrata com aplicações que desejam controlar recursos dos dispositivos em um ambiente genuinamente IoT. Essa comunicação deve ser simples e transparente para qualquer aplicação e não deve envolver questões específicas do ambiente onde o *middleware* foi instanciado. Detalhes sobre protocolos de rede, configurações de hardware ou outros quesitos que envolvem configurações de baixo nível não devem ser repassados para a aplicação. Aplicações IoT podem depender de uma ***Execution Engine*** (máquina de execução), a qual, por sua vez, pode usar recursos providos pelo *middleware* baseado em FlyIoT. O ***FS Config*** (Configuração de Serviços Federados) é um conjunto de instruções e permissões que pode ser integrado à aplicação a fim de permitir, por exemplo, que o *middleware* acesse recursos pertencentes ao usuário em um serviço proprietário de nuvem. O termo "recurso" (*Resource* ou *thing*) em FlyIoT é usado para designar sensores, atuadores, processadores e volumes de armazenamento, além de recursos compostos, formados por meio da agregação ou agrupamento de recursos identificados pelo *middleware* durante sua execução.

Um recurso, para FlyIoT, pode ser categorizado como: Entrada (*Input*), Saída (*Output*) ou Processador (*Processor*). Todos os tipos de recursos podem prover ao *middleware* baseado em FlyIoT suas capacidades funcionais no momento de sua descoberta, tornando-o ciente e permitindo lidar com cada recurso no decorrer da execução da aplicação IoT.

***Input*:** Recursos com características de coleta de informações e entrada de dados para FlyIoT, seja via evento, seja por algum tipo de medição, como comumente é utilizado

em ambientes IoT. Enquadram-se nesse tipo de recurso os sensores, os dispositivos de entrada do usuário e os volumes de armazenamento em modo leitura. Alguns exemplos de sensores são: termômetros, sensor sonoro, sensor de presença, sensor de vibração; exemplos de dispositivos de entrada: teclados, telas *touch*, reconhecedores de comandos de voz; exemplos de volumes de armazenamento: conjunto de arquivos em algum recurso como HDs (Hard Disk Drive) ou SSDs (Solid State Drive), entre outros.

***Output*:** Recursos que permitem a saída de dados e ações por parte de FlyIoT, sendo capazes de fornecer informações por algum meio para a aplicação. Nesta categoria estão presentes os atuadores, dispositivos de saída e volumes de armazenamento em modo escrita. Enquadram-se nesse tipo de recurso, por exemplo, lâmpadas, ventiladores, telas, caixas de som, emissores de cheiro e também conjuntos de arquivos armazenados em HDs e SSDs, entre outros.

***Processor*:** Esses recursos têm características distintas do tipo *Input* e *Output*, pois permitem a execução de *scripts* ou sequências de código. Podem ser mais complexos que os recursos anteriores, pois sempre estão ligados a alguma arquitetura de *hardware*, possuindo informações distintas como velocidade de processamento, núcleos e cache, entre várias outras. Um recurso desse tipo é capaz de informar quais linguagens de programação ele suporta como, por exemplo, um processador que executa *scripts* Lua (IERUSALIMSCHY; DE FIGUEIREDO; FILHO, 1996) pode ser descoberto e disponibilizado pelo *middleware* para a aplicação.

Ao traçar um paralelo com uma máquina única, os recursos de entrada, saída, armazenamento e processadores do *middleware* podem ser equiparados ao *hardware* de CPU, memória e dispositivos de um computador. A comunicação entre recursos fundamentais em ambientes IoT equivale à comunicação entre dispositivos de E/S por meio dos barramentos dos computadores, ressalvada a maior escala, heterogeneidade e intermitência observadas em IoT. O conjunto de componentes de FlyIoT busca tratar tais desafios de forma transparente para os desenvolvedores.

3.1 Camada de serviços FlyIoT

De acordo com o estudo conduzido por (RAZZAQUE et al., 2016) e (SILVA; KHAN; HAN, 2018), um *middleware* precisa atender alguns requisitos, dentre eles encontrar, identificar e catalogar os diversos recursos em um ambiente IoT, além de controlar as informações geradas por tais recursos. Diante disso, na arquitetura FlyIoT, existe uma camada de serviços, os quais podem ser distribuídos e executados de forma independente dos demais. No entanto, a arquitetura FlyIoT não especifica as políticas de distribuição pois ela depende do meio em que o *middleware* é instanciado. Elementos como latência,

capacidade dos dispositivos, conexão, entre outros, idealmente devem ser considerados para que um determinado componente de serviço seja instanciado em um ponto ou outro. Tal procedimento pode requerer um estudo minucioso do meio executado e, portanto, uma arquitetura é apenas um primeiro passo para que essa distribuição seja executada posteriormente. Na camada de serviços de FlyIoT, cada tipo de serviço é tratado como um módulo da arquitetura. O módulo ***Discovery Service*** (Serviço de descoberta), uma vez instanciado, é responsável por realizar a descoberta dos recursos em um ambiente e dos próprios serviços do *middleware*.

Todos os módulos de serviço do *middleware* devem possuir um componente de descoberta para que eles possam se anunciar quando forem instanciados como serviços na rede. Quando o *middleware* baseado em FlyIoT é instanciado de forma distribuída com componentes presentes na nuvem, o *Discovery Service* também deve ser capaz de identificar esses serviços e torná-los disponíveis para serem utilizados pelo *middleware*. A partir do momento em que um serviço é instanciado em uma rede local ou na nuvem, um ou mais *Discovery Services* devem identificar as propriedades (tipo, identificador, endereços, protocolos de comunicação suportados) desse serviço e torná-las acessíveis para os demais componentes. Quando se trata de uma descoberta em uma rede local, na prática seria necessário um *Discovery Service* para reconhecer todos os componentes do *middleware* que foram inicializados. Entretanto, quando há mais de uma rede, pode ser necessário mais de um *Discovery Service* para auxiliar nessa descoberta.

O módulo ***Data Management Service*** é responsável por armazenar e gerenciar dados úteis para o funcionamento do próprio *middleware*. Podem ser armazenados e gerenciados dados referentes a recursos e outros componentes descobertos por um *Discovery Service*, bem como dados para controle do próprio *middleware*. Na arquitetura FlyIoT, não é proibitivo haver mais de uma instância de *Data Management Service*, ou seja, pode haver mais de um *Data Management Service*, um para cada tipo de controle de dados desejado. Porém, se houver mais de um *Data Management Service* com informações duplicadas, elas devem estar sincronizadas, de modo que o acesso às informações seja consistente para a aplicação IoT independentemente da política de acesso e distribuição. As instâncias dos *Data Management Service* podem fazer uso dos recursos de armazenamento descobertos para manter os dados de interesse se necessário.

O ***Management Service*** pode ser o módulo mais complexo em FlyIoT, dependendo da gama de serviços incluídos em sua implementação. A função deste módulo é controlar, monitorar e orquestrar os demais serviços e componentes do *middleware*. O ideal é que exista uma instância desse módulo em cada ambiente incluído em um cenário como, por exemplo, um na rede local e outro na nuvem. Essas várias instâncias devem se comunicar entre si, a fim de informar sobre a disponibilidade ou não de algum serviço e, se necessário, eleger um novo recurso de processamento capaz de prover uma nova instância

de um serviço que deixou de existir. Nota-se, neste ponto, que recursos descobertos ficam disponíveis não somente para as aplicações, mas também para os próprios módulos do *middleware*. O *Management Service* possui três subcomponentes: *System Orchestration*, responsável por manter, controlar e ordenar a instanciação de outros serviços do *middleware*; *System Monitoring*, responsável por observar o estado e disponibilidade dos demais componentes do *middleware*; e *QoS Management*, que é um componente opcional para monitorar e gerenciar a qualidade de transmissão de informações entre os componentes do *middleware*, de acordo com alguma exigência de qualidade de serviço (QoS) advinda da aplicação. O *Management Service* também deve ser responsável por autorizar o acesso através de algum método seguro de autenticação para uso das informações do *middleware* por usuário e aplicação.

(ALVI et al., 2015) e (YAQOOB et al., 2017) enfatizam a importância de protocolos e da ciência de contexto por parte de *middlewares* IoT, que são específicos a certos conjuntos de aplicações. Por exemplo, aplicações distribuídas envolvendo conexões e troca de informações com dispositivos móveis (como *smartphones*), requerem protocolos específicos para manter a qualidade e disponibilidade de informações processadas em determinadas circunstâncias intrínsecas ao meio, bem como a resiliência a falhas, evitando perda de dados e permitindo que se recuperem automaticamente. Embora a proposta de FlyIoT não contemple protocolos específicos para algum ambiente e métodos para aquisição de dados e análise de contexto, tal necessidade é considerada na arquitetura. O módulo ***Protocol Resolver Service*** é responsável por manter conhecimento do *middleware* sobre protocolos de forma geral. Uma instância desse módulo pode estar disponível para consulta pelo *middleware* ao tentar se comunicar com algum recurso que inicialmente ele não reconheça. O *Protocol Resolver Service* pode fornecer as dependências (código ou binário) necessárias para interagir com o novo recurso. Com isso, um *middleware* baseado em FlyIoT pode se tornar mutável e adaptável de acordo com necessidades de alguma aplicação ou ambiente específico.

O módulo ***Context Reasoning Service*** é responsável pelo suporte à ciência de contexto em FlyIoT, podendo haver mais de uma instância desse módulo. A função dele é analisar os dados relacionados a um ou mais recursos, extraindo informações importantes para a aplicação sobre o contexto de execução. Diversas ações que uma aplicação realiza podem depender desse tipo de informação para trazer mais adaptabilidade na execução. O *Context Reasoning Service* pode processar informações sobre o ambiente em que o *middleware* está sendo executado, armazenadas em uma instância de *Data Management Service*. Ele pode, então, usar tais informações como apoio para tomadas de decisão do *middleware*. O foco da presente proposta de arquitetura não é tratar especificamente de análise e ciência de contexto, e sim evidenciar a necessidade e a importância de adaptar serviços de análise de contexto e complementar as capacidades do mesmo. Trabalhos como (VEIGA et al., 2017), (MAARALA; SU; RIEKKI, 2016) (GIL et al., 2016) têm

foco direto na resolução de contexto e análise de grande quantidade de dados para extrair conhecimentos importantes dessas bases.

Além disso, outros módulos podem ser incorporados a um *middleware* baseado em FlyIoT, desde que obedeçam às regras e protocolos de comunicação entre os seus serviços. Assim, um *middleware* pode ser adaptado a novos recursos que possam surgir futuramente, visto que esse meio está em constante mudança e evolução. Do ponto de vista "IoT como uma máquina", os módulos de serviços do *middleware* apresentados podem ser comparados a componentes de *software* que um sistema operacional precisa acionar para controlar seus dispositivos – tanto *hardwares* quanto outros *softwares*. FlyIoT considera que novos componentes de serviços podem ser "instalados" (similar ao conceito usado para computadores) para lidar com alguma nova tarefa específica.

3.2 Camada de adaptação FlyIoT

Fazem parte de FlyIoT, em um nível mais baixo, *Interfaces* e *Controllers* de sensores, atuadores e processadores. Esses componentes também compõem a camada de adaptação do *middleware* FlyIoT.

Como já citado anteriormente, em ambientes IoT, é comum que sensores e atuadores sejam recursos mais simples e restritos, sem meios sofisticados de interação entre eles. Geralmente podem se comunicar apenas via algum protocolo específico de baixo nível ou, até mesmo, são tão restritos que não possuem um protocolo de comunicação. Um exemplo disso é a utilização de um recurso que inicialmente não foi projetado para ambiente IoT, tornando-o acessível para outros componentes em uma rede IoT. Para isso, seria necessário o uso de um *Controller* específico para esse sensor ou atuador. Os ***Controllers*** são responsáveis por controlar e manipular o comportamento desses recursos (*Resources*) e podem ser categorizados em dois tipos.

***Interno ao Resource*:** Esses *Controllers* já vêm incorporados ao próprio *Resource*. Eles normalmente expõem chamadas de API de sistema para que os interessados se comuniquem com o respectivo *Resource*. Exemplos podem ser encontrados em *smartphones* e *tablets* que oferecem chamadas de API para que aplicativos móveis tenham acesso a determinados recursos, como câmera, microfone, armazenamento, etc.

***Externo ao Resource*:** Diferentemente da categoria anterior, esse tipo de *Controller* não está presente no *Resource*, geralmente devido a limitações de *hardware* ou simplesmente pelo fato de o *Resource* não ter sido projetado para ambientes IoT. Nesse caso é necessário que um *Controller* externo ao *Resource* seja implementado para controlá-lo. Existem alguns sensores e atuadores que são *Resources*, por exemplo,

projetados para comunicação apenas via GPIO (General Purpose Input/Output) com microcontroladores e microprocessadores.

Os *Controllers*, na arquitetura FlyIoT, são responsáveis por repassar ações do *middleware*, advindas de aplicações IoT, para os seus respectivos *Resources*, além de fazer o registro de eventos a serem observados pela aplicação, enviar notificações ao *middleware* quando um determinado evento de interesse ocorre e repassar algum *script* para execução em *Resources* do tipo *Processor*.

A ***Interface*** é um componente intermediário de software que contém as regras de comunicação com qualquer componente de FlyIoT. Essas interfaces basicamente são a forma pela qual os *Controllers* devem expor os *Resources* para a camada de serviços do *middleware*. Por meio da *Interface* também são fornecidas informações para que o *Discovery Service* identifique tal *Resource* quando ele se tornar disponível. Tais informações são providas pelo *Controller*, que são repassadas para a *Interface*. Traçando um paralelo com uma máquina única, um *Controller*, juntamente com sua *Interface*, se equipara aos *drivers* que os sistemas operacionais necessitam para entender e interagir com os dispositivos de *hardware* que, por sua vez, se assemelham aos *Resources* (ou *things*) com os quais o *middleware* interage.

Um destaque importante em FlyIoT é que, por meio de *Controllers* e *Intefaces*, e partindo da premissa de que existam *Resources* que se anunciam como processadores e armazenamento, é possível identificar o quão aptos esses *Resources* estão para receberem uma instância de um componente de módulo de serviço que compõe FlyIoT. Os impactos das dificuldades de implantação de código e serviços apontados por (RAZZAQUE et al., 2016) podem, dessa forma, ser reduzidos consideravelmente, até mesmo ao se determinar se o *middleware* baseado em FlyIoT é capaz ou não de atender a alguma necessidade de uma aplicação IoT.

Em resumo, é possível observar a relação da arquitetura FlyIoT diante do conceito de máquina única. Os *Input*, *Outputs* e *Processors* são similares aos dispositivos de *hardware* de uma máquina, que possuem controladores de recursos e interfaces a serem gerenciados pelo sistema operacional (conjunto de serviços em FlyIoT). Além disso, a composição desses serviços de FlyIoT pode oferecer às aplicações os recursos de um ambiente IoT em um nível de abstração mais elevado, simplificando a maneira pela qual essa aplicação lida com tais recursos. FlyIoT permite o acesso a esses recursos eximindo a aplicação de lidar com questões específicas do ambiente em que está sendo executada como questões de conexão, configuração e preparação desses recursos.

4 PROVA DE CONCEITO DA ARQUITETURA FLYIOT

Neste capítulo é descrita uma implementação dos componentes e comunicação de um *middleware* baseado na arquitetura FlyIoT a título de prova de conceito. FlyIoT segue o paradigma distribuído baseado em serviços, em que cada um de seus componentes deve ser instanciado em uma rede local, na nuvem ou outros ambientes de rede. Os componentes de serviço desta implementação são capazes de comunicar por meio chamadas em API's RESTful Cada um dos componentes de serviço oferece essas API's para que os demais componentes na rede tenham acesso.

4.1 Containers

Uma grande evolução no processo de virtualização se deu, nos últimos anos, por meio do uso de *containers* (BERNSTEIN, 2014), os quais possuem uma imagem completa de todos os seus requisitos de software e são executados de maneira nativa, compartilhando o mesmo *kernel* do sistema operacional, mas de forma a isolar os processos da aplicação em relação ao restante do sistema. Por ser uma abordagem mais leve e com vantagens análogas à da virtualização, o uso de *containers* para executar aplicações e serviços em rede já é uma realidade no mercado. Alguns exemplos de gerenciadores de containers são: *Docker* (durante o desenvolvimento), *Kubernetes* (HIGHTOWER; BURNS; BEDA, 2017), *Mesosphere* (MESOS, s.d.) e *Docker Swarm* (NAIK, 2016).

Na instanciação do *middleware* FlyIoT, sempre que possível, cada módulo de serviço e controlador é instanciado via *container*, de forma a permitir uma orquestração mais simples por meio de algum gerenciador de *containers*. No entanto, alguns dispositivos de hardware não permitem implantação de *containers* em razão de limitações de *kernel*, como é o caso de *smartphones*. Nesse caso é necessária a instanciação de forma nativa, por meio de aplicativo (.apk para o sistema operacional Android).

Para cada serviço do FlyIoT, há um arquivo de configuração ".dockerfile", que contém as informações necessárias sobre a imagem do sistema operacional, arquivos e dependências para a construção de uma imagem e execução de um *container* com o respectivo componente.

Cada *container* executando um serviço ou um controlador possui um identificador único no momento da inicialização. Portanto, cada instância tem apenas um ciclo de vida e não deve ser retomada após finalizada, ou seja, sempre que uma instância iniciar sua execução, esta será uma nova instância em um novo *container*. Essa abordagem simplifica o processo de orquestração dos serviços e evita a necessidade de monitorá-los para que retornem ao estado atual em caso de falha, bastando simplesmente finalizar e inicializar um novo *container*.

4.2 Descrição da implementação

Os relacionamentos e funcionamento dos componentes de FlyIoT podem ser vistos no diagrama de classes da Figura 2. O diagrama apresenta os graus de cardinalidade de relacionamento entre os componentes, além de algumas das operações principais implementadas e heranças de componentes. Inicialmente, pode-se visualizar duas entidades bases no *middleware* implementado como um todo: *Services* e *Resources*.

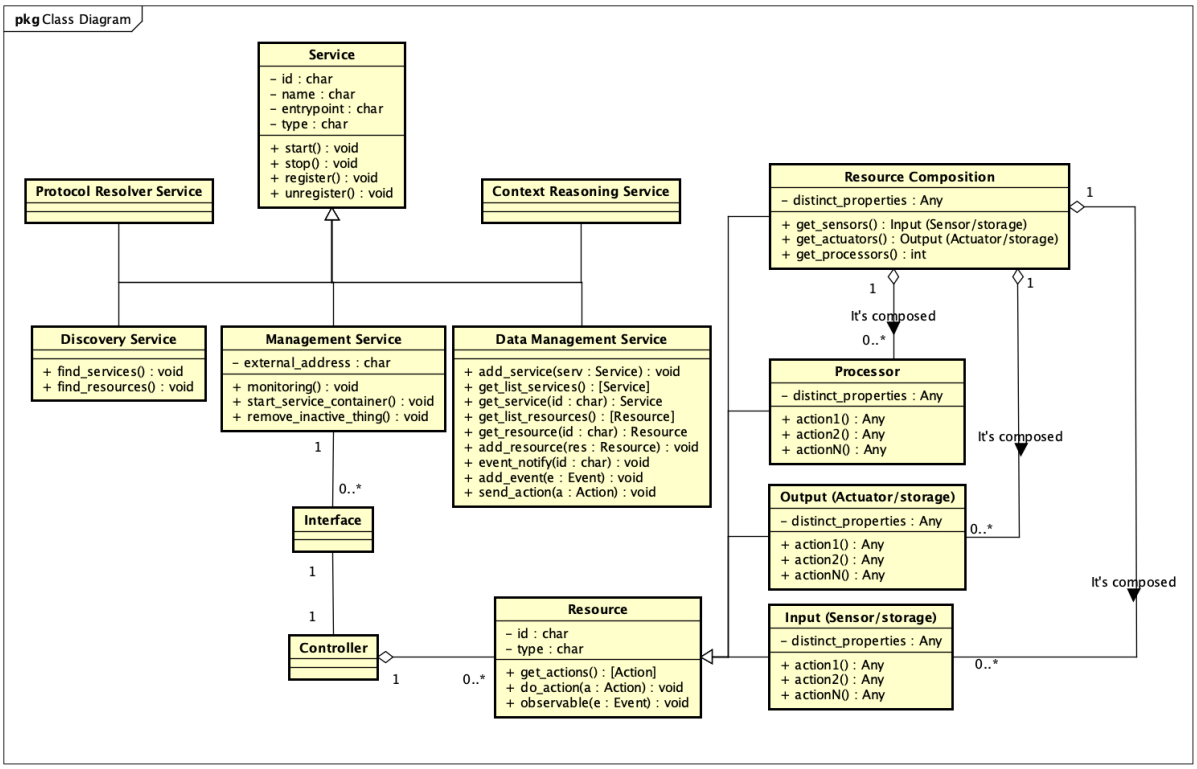

Figura 2 – Diagrama de classes do *middleware* FlyIoT

Cada entidade no diagrama possui atributos e funções relevantes para o seu funcionamento. No caso de um *Service*, pode-se observar alguns atributos como o *id*, *name*, *entrypoint* e *type*. Todo *Service* também possui métodos de *start()* e *stop()*, *register()* e *unregister()*.

***id*:** Identificador único, do tipo *uuid* (LEACH; MEALLING; SALZ, 2005), gerado no momento da instanciação do *Service*. Esse identificador será utilizado pelo *middleware* para manipular e referenciar os seus componentes de serviço.

***name*:** Usado para diferenciar *Services* do mesmo tipo de acordo com sua finalidade. Por exemplo, pode existir em uma instanciação do middleware dois *Data Management Service*: um chamado *Data Management Home* e outro *Data Management Work*. Apesar disso, esse atributo *name* é opcional, pois a referência de fato ao componente será o *uuid*.

***entrypoint*:** O atributo que contem o endereço (*address:port*) base para que outros componentes do middleware interajam com o respectivo *Service*. Essa informação será anunciada pelos *Services* no momento da instanciação.

***type*:** Atributo que representa o tipo do *Service* propriamente dito. Pode ser *Discovery Service*, *Management Service*, *Data Management Service*, etc.

***start()*:** Inicializa a preparação do *Service* até que ele esteja apto a interagir com os demais componentes.

***stop()*:** Finaliza os processos do *Service* de modo não forçado, para evitar vazamento de memória, permitindo que seja encerrado posteriormente.

***register()*:** Responsável por registrar o *Service* como um serviço na rede. Em seguida ele se anuncia com suas informações essenciais para interação e se torna visível para que o *Discovery Service* o identifique.

***unregister()*:** Prepara o *Service* para cancelar seu registro na rede como um serviço, o qual em seguida deixa de existir e de se anunciar.

Um *Service* deve ser herdado para os demais tipos de serviços mencionados anteriormente: *Discovery Service*, *Management Service*, *Data Management Service*, *Context Reasoning Service*, *Protocol Resolver Service*, ou até mesmo outros não implementados ou abordados nessa prova de conceito, pois o *middleware* pode ser adaptado futuramente para suportar novos tipos de *Services*. O *Discovery Service* possui dois métodos distintos *find_services()* e *find_resources()*.

***find_services()*:** É executado logo após o *Discovery Service* se registrar como um serviço na rede. A operação fica sempre em execução enquanto o *Discovery Service* está ativo, e é responsável por buscar e encontrar outros *Services* que são registrados na rede. Quando o *Discovery Service* deixa de existir, o método em questão também é finalizado.

***find_resources()*:** De maneira similar, é responsável por buscar e encontrar *Resources* na rede, mais especificamente *Controllers* que anunciam tais *Resources* por meio de sua *Interface*.

O *Data Management Service* possui algumas funções necessárias para o seu funcionamento, como *add_service(s: Service)*, *get_list_service()*, *get_service(id: Char)*, *add_resource(r: Resource)*, *get_list_resource()*, *get_resource(id: Char)*, além de outros métodos de CRUD (*Create, Read, Update, Delete*) tanto para *Services* quanto para *thing*.

O *Data Management Service* intermedeia o acesso à base de dados pelos demais componentes do *middleware*, além de prover funções como envio de ações ou notificações por parte dos *Resources* e aplicação.

***add_service(s: Service)*:** Adiciona um novo *Service* no *Data Management Service*.

***update_service(s: Service)*:** Altera as informações de um determinado *Service* no *Data Management Service*.

***get_list_services(): [Service]*:** Retorna uma lista de todos os *Services* que estão registrados no *Data Management Service*.

***get_service(id: char): Service*:** Retorna um *Service* que possua o id passado por parâmetro, se o mesmo existir.

***get_services(type: char): [Service]*:** Retorna uma lista de *Services* que seja do tipo passado por parâmetro.

***remove_service(id: char): Service*:** Remove o *Service* que possua o *id* passado por parâmetro.

***add_resource(t: Resource)*:** Adiciona um novo *Resource* no *Data Management Service*

***get_list_resources(): [Resource]*:** Retorna uma lista de todos os *Resources* que estão registrados no *Data Management Service*.

***get_resource(id: char): Resource*:** Retorna um *Resource* que possua o id passado por parâmetro, se o mesmo existir.

***get_resource(type: char): [Resource]*:** Retorna um lista de *Resources* que possua o tipo passado por parâmetro.

***update_resource(t: Resource)*:** Altera as informações de um determinado *Resource* no *Data Management Service*.

***remove_resource(id: char)*:** Remove o *Resource* que possua o *id* passado por parâmetro.

***get_types(): [String]*:** Retorna uma lista de tipos de *Resources* que já foram descobertos pelo *middleware*.

***add_event(e: Event)*:** Realiza o registro de um novo evento a ser observado pelo *middleware*. Cada evento contém um *id* único e parâmetros de acordo com sua natureza (expressão lógica, intervalo ou outras informações específicas).

***event_notify(id: Char)*:** Aciona o *middleware* informando o *id* do evento que ocorreu. Em seguida o *Data Management Service* notifica a *Execution Engine* ou aplicação sobre a ocorrência do evento.

***add_action(a: Action)*:** Registra uma nova ação que o *middleware* deve repassar ao respectivo *Controller* do *Resource*.

Todo *Resource* possui um *id* único e *type*, além de métodos de *get_actions()*, *do_action(a: Action)* e *observable(e: Event)*.

***id*:** Identificador único do tipo *uuid* gerado no momento em que o *Controller* do *Resource* é inicializado.

***type*:** Refere-se ao tipo do *Resource* em questão. Esse tipo estará disponível para o desenvolvedor via aplicação FlyIoTL.

***get_actions(): [Action]*:** Retorna a lista de *actions* suportadas por determinado *Resource*. Essas informações serão expostas para o *middleware* FlyIoT via *Controller* e *Interface*.

***do_action(a: Action)*:** Executa uma *action* a partir de uma notificação realizada pelo *middleware* para o *Resource* em questão.

***observable(e: Event)*:** Registra um determinado *observable* de evento para o *Resource*. Quando esse evento ocorrer, o *middleware* deverá ser notificado pelo *Controller* do *Resource*.

Um recurso *Processor* é todo aquele capaz de executar algum código demandado pela aplicação FlyIoT ou por serviços do próprio *middleware*. *Processors* podem prover esse tipo de capacidade para o *middleware* por meio de seus respectivos *Controllers* que são responsáveis pela carga do ambiente de execução suportado e do código que porventura lhe seja atribuído. *Processors* podem possuir propriedades bem distintas, como tipos de linguagens de programação suportadas, arquitetura do *chip*, velocidade de processamento, quantidade de núcleos, entre outros.

Os recursos do tipo *Input* e *Output* abrangem sensores e atuadores respectivamente. Em geral, como esses tipos de recurso são bem limitados, cada um possui seu próprio controlador ou faz parte de um *Resource Composition* para expor suas capacidades ao *middleware* via um controlador em comum. Ambos os tipos podem possuir ações e propriedades distintas de acordo com a natureza de cada *Resource*.

Uma peculiaridade importante é a possibilidade de oferecer um recurso de *Storage* às aplicações e aos serviços do próprio *middleware*. Dependendo da finalidade, ele pode

ser considerado um *Input* ou *Output*. Exemplo disso seria uma aplicação que precisa persistir dados coletados em algum local. Esse tipo de *Resource* seria disponibilizado como *Output*. Na outra ponta, estaria uma aplicação que precisa iniciar um procedimento a partir da leitura de um arquivo gerado por outra aplicação/processo. Nesse caso, o arquivo se encontraria em um *Resource* do tipo *Input*.

É importante destacar mais alguns detalhes sobre a implementação dos módulos da arquitetura FlyIoT no *middleware* de prova de conceito. A linguagem de programação utilizada foi o *Python 3*, mais especificamente o *Python 3.6* (VAN ROSSUM; DRAKE JR, 2014), para todos os *Services* (*Discovery Service*, *Data Management Service* e *Management Service*). As dependências básicas, do ponto de vista de bibliotecas utilizadas nos *Services* do *middleware*, são bem similares, salvo exceção para o *Data Management Service*, que contém também algumas dependências para comunicação com o banco de dados, cujo escolhido foi o *MongoDB* (CHODOROW, 2013). Toda mensagem trocada entre a *Execution Engine* e demais componentes desta prova de conceito de FlyIoT possui autenticação por meio de JSON Web Token (JWT)(JONES; BRADLEY; SAKIMURA, 2015).

Todo *Service* da prova de conceito contém um script *"register.py"* responsável por anunciá-lo na rede com suas respectivas capacidades. Além disso, cada *Service* contém um *script "app.py"* que implementa as funções REST de comunicação com o próprio *Service*. Outros arquivos adicionais, *"const.py"* e *"util.py"*, também estão presentes com informações sobre valores de configuração (como por exemplo, nome e tipo do *Service*) e funções úteis comuns utilizadas na implementação. Os dois *scripts* (*"register.py"* e *"app.py"*) são executados como subprocessos de um *"script"* principal chamado *"main.py"* que é executado para inicializar a instância do *Service*. Sempre que um *script* do *Service* é parado, os demais scripts também são encerrados, evitando vazamento de memória (*Memory Leak*) no *Resource*. Maiores informações sobre os arquivos referentes a implementação da prova de conceito podem ser encontrados no repositório (*https://github.com/thomasmarquesbr/middleware_flyiot*).

O conceito seguido e implementado é que todo *Service* pode ser obtido por meio de *containers*, quando o *Resource* em questão tem capacidade para tal. Diante disso, todo *Service* contém um *"script"* de descrição de instanciação de *containers* com informações sobre qual imagem base de virtualização deve ser construída, bem como as dependências de projeto e componentes que compõem o *Service*. Outra alternativa seria o *container* ser executado a partir de uma imagem construída para o respectivo *Service* e disponibilizada para download em um repositório na nuvem (como Docker Hub, por exemplo). Vale ressaltar que, para a segunda alternativa, não significa que informações estarão sempre sendo enviadas para a nuvem, ou seja, a conexão seria apenas para download da imagem pronta para execução de um *Service* do *middleware* em questão.

Na implementação dos *Controllers* e *Interfaces* para os *Resources* (ou *things* em

FlyIoTL), foi utilizada uma estrutura de projeto equivalente. A diferença aqui é que um *script "app.py"* implementa a *Interface* de comunicação do *middleware* com o *Resource*. Existe também um *"controller.py"*, que implementa o controlador do *Resource*. Há ainda um *script "register.py"* que é responsável por anunciar o *"Resource"*, provendo as informações essenciais de descoberta para o *middleware*. Devido a limitações do sistema operacional *Android*, as implementações dos *Controllers* e *Interfaces* de *Resources* foram realizadas por meio de aplicativo para o sistema (*.apk*). Nesse caso, boa parte do *Controller* de seus *Resources* é acessível por meio de API's disponíveis no sistema *Android* e, por isso, foi utilizada a linguagem *Kotlin* (SAMUEL; BOCUTIU, 2017) para seu desenvolvimento. Adaptações para atender às necessidades da arquitetura FlyIoT foram desenvolvidas complementando seus *Controllers* e inserindo a *Interface* de comunicação com o *middleware*.

4.3 Comunicação entre os componentes FlyIoT

Recentemente, no âmbito de serviços IoT, é muito comum a especificação de APIs baseadas em princípios REST (RICHARDSON; AMUNDSEN; RUBY, 2013). REST consiste em operações, regras e restrições que permitem a criação de soluções com interfaces bem definidas e inerentemente remotas. Uma API REST aproveita as definições do HTTP (FIELDING et al., 1999), usando a operação GET para recuperar um recurso; PUT para alterar o estado ou atualizar um recurso; POST para criar um recurso; e DELETE para removê-lo.

Como as chamadas são, em sua essência, *stateless* (sem estado e não dependente de requisições anteriores), REST é útil em aplicações na nuvem, móveis e IoT. Os componentes sem estado podem ser livremente reimplantados se algo falhar e podem ser redimensionados para acomodar as alterações de carga. Isso ocorre porque qualquer solicitação pode ser direcionada para qualquer instância de um componente. Isso torna o REST útil em arquiteturas orientadas a serviços porque a vinculação a um serviço por meio de uma API é apenas uma questão de controlar como o localizador de recurso uniforme (URL) é decodificado.

Para esta prova de conceito da arquitetura FlyIoT, é proposta uma implementação em que a comunicação entre os seus serviços seja via API's RESTful. O corpo da mensagem HTTP passada entre os componentes de FlyIoT contém objetos simples, a fim de minimizar os impactos de processamento pelos componentes envolvidos. A estrutura utilizada para a troca de informação segue o formato JSON (OSKARSSON et al., 1996), por permitir o intercâmbio de dados de forma leve, baseado em texto e independente de linguagem.

Sempre que um novo *Resource* (sensor, atuador, armazenamento ou processador) se tornar disponível em uma rede em que FlyIoT foi instanciado, o seu *Controller* irá se anunciar juntamente com suas chamadas REST por meio da *Interface*. Com isso, uma

instância de *Discovery Service* descobre o novo recurso junto a uma instância do *Data Management Service*, por meio de uma chamada REST também provida por ele. Esse processo de descoberta pode ser observado na Figura 3, como um diagrama de sequência. Na figura, observa-se que uma aplicação, quando necessário, pode realizar requisições REST ao *Data Management Serivice* para obter informações sobre os *Resources* disponíveis descobertos anteriormente pelo *middleware*. Quando um novo *Resource* é inserido na rede, seu *Controller* irá anunciá-lo e, então, o componente *Discovery Service* instanciado o detecta. O *Discovery Service* repassará essas informações sobre o *Resource* via requisição POST para o *Data Management Service*. Posteriormente, as informações sobre o novo *Resource* estarão disponíveis para a aplicação.

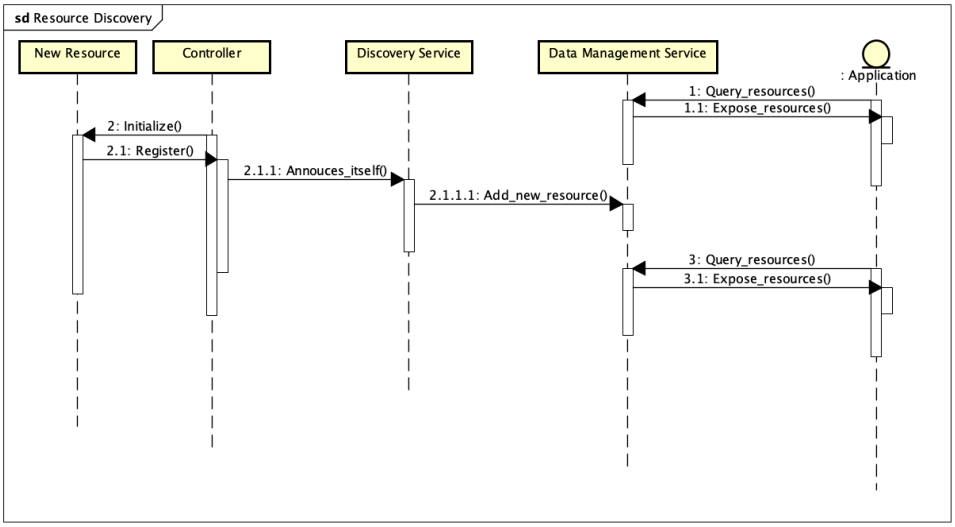

Figura 3 – Procedimento de descoberta de recursos

No trecho **??** abaixo é possível observar um exemplo de como um *Resource* é anunciado com suas informações essenciais para que o *middleware* o registre e o torne disponível para a aplicação. Cada *Resource* possui informações individuais e outras essenciais para o *middleware*, como *entrypoint*, *"uuid"*, *"type"*, *"observables"*, por exemplo. Essas informações serão importantes posteriormente para o controle de ações e eventos por parte do *"middleware"* na execução da aplicação.


```
{
    "entrypoint": "http://10.0.1.26:11000/",
    "type": "touchSurface",
    "model": "Xiaomi_Redmi 5",
    "timestamp": "2019-08-23 20:02:34.335185",
    "addr": "04:B1:67:36:1E:CC",
    "multitouch": true,
    "name": "Xiaomi_Redmi 5",
    "observables": ["swipeUp", "swipeDown", "swipeLeft","swipeRight"],
```

```
    "uuid": "4f23e577-c511-4f3c-a346-dd3081fbef8a"
}
```

Quando um recurso ou serviço deixa de existir na rede, ou seja, o controlador ou serviço deixa de se anunciar, o *Management Service* detecta que tal componente ficou indisponível. Então o *Management Service* notifica o *Data Management Service*, remove esses componentes e, quando necessário – para o caso de algum *Service* essencial ao *middleware* –, inicia o processo de instanciação de um novo *Service*.

A Figura 4 ilustra esse comportamento para quando um *Resource* se torna indisponível. Durante a execução de um *middleware* baseado em FlyIoT, um componente de *Management Service* está sempre monitorando, de tempos em tempos, a disponibilidade de um determinado *Resource* por meio de chamadas ao seu *Controller*. Quando esse *Resource* em questão se torna indisponível, seja qual for o motivo (nesse caso o encerramento por parte do próprio *Controller*), o *Controller* desliga (ou finaliza) o *Resource* e encerra seu registro de anúncio na rede. Logo em seguida o *Management Service* detecta a indisponibilidade desse *Resource*. Para essa detecção, o *Management Service* realiza uma requisição de DELETE para remoção do respectivo *Resource* da base de dados do *Data Management Service*.

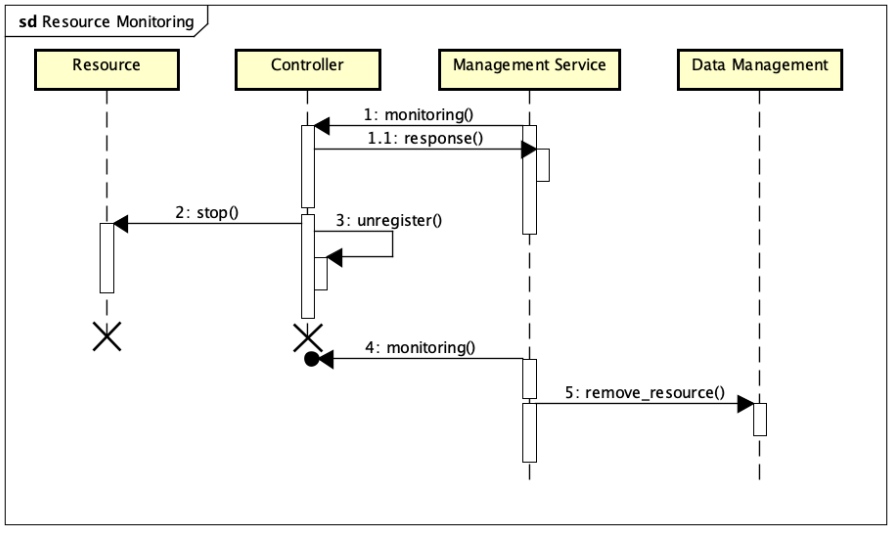

Figura 4 – Procedimento de monitoramento quando recurso se torna indisponível

Já a Figura 5 ilustra o que acontece quando um *Service* essencial se torna indisponível. Assim como no caso anterior, o *Management Service* monitora de tempos em tempos a disponibilidade dos *Services* essenciais. Quando o *Discovery Service* em questão se torna indisponível, o *Management Service* detecta a indisponibilidade e realiza a requisição de DELETE no *Data Management Service*.

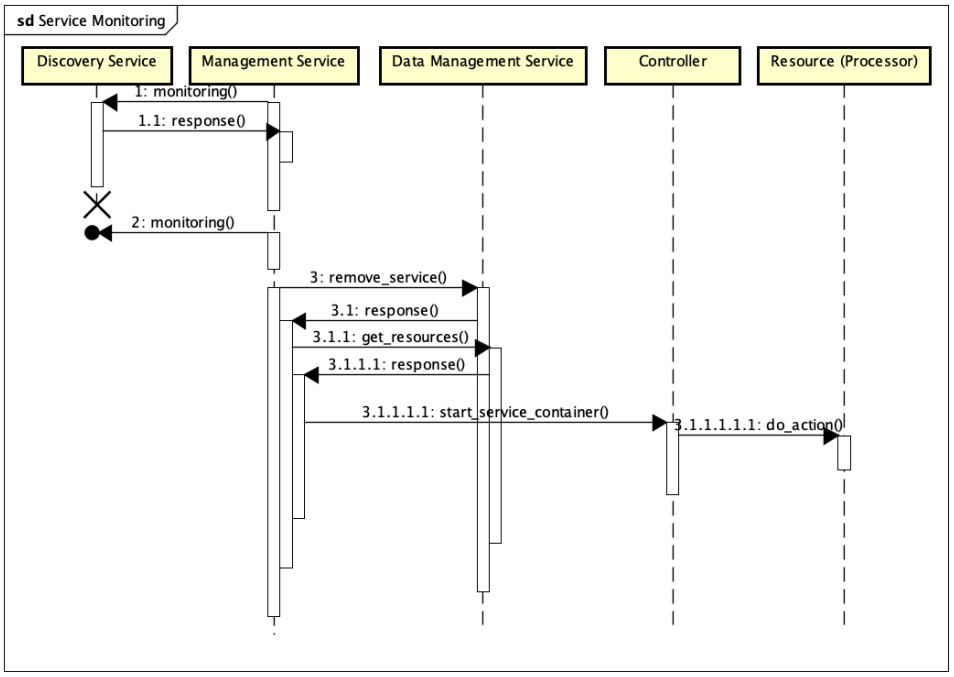

Figura 5 – Procedimento de monitoramento de serviço essencial indisponível

Como nesse caso foi um *Service* que se tornou indisponível, o *Management Service* verifica se esse *Service* é essencial ao funcionamento do *middleware*. Caso seja verdadeiro, ele realiza uma requisição GET ao *Data Management Service* para recuperar a lista de *Resources* do tipo *Processor* disponíveis e verifica quais deles estão aptos a instanciarem um novo *Discovery Service*. Tendo encontrado tal *Resource*, o *Management Service* realiza uma requisição PUT para o *Resource* em questão, enviando um *"script"* para que tal *Processor* o execute a fim de instanciar um novo *Discovery Service* naquela rede.

Quando uma aplicação IoT necessita realizar uma ação sobre algum *Resource* (*thing*), a *Execution Engine* notifica o *Data Management Service*, que, por sua vez, identifica quais os *Resources* do tipo especificado estão acessíveis e dispara a ação para o *controller* desse *Resource*. A Figura 6 ilustra tal comportamento. Uma aplicação que deseja enviar uma ação para algum *Resource* irá fazer uma requisição PUT ao *Data Management Service* informando o *type* ou *id* desse *Resource*. O *Data Management Service* então realiza uma consulta em sua base de dados sobre os *Resources* disponíveis. Se houver tal *Resource*, o *Data Management Service* irá disparar a ação para o seu respectivo *Controller*, que acionará o *Resource*. Uma ação para um *type* de *Resource* resultará em disparos da ação a todos os *Resources* desse mesmo *type* que estejam disponíveis em *Data Management Service*. Caso contrário, uma ação para um *id*, resultará em ação disparada apenas para aquele *Resource* em questão.

Uma aplicação IoT também pode registrar *observables* com o intuito de monitorar eventos e ser notificada quando esses eventos ocorrerem. A primeira parte do registro

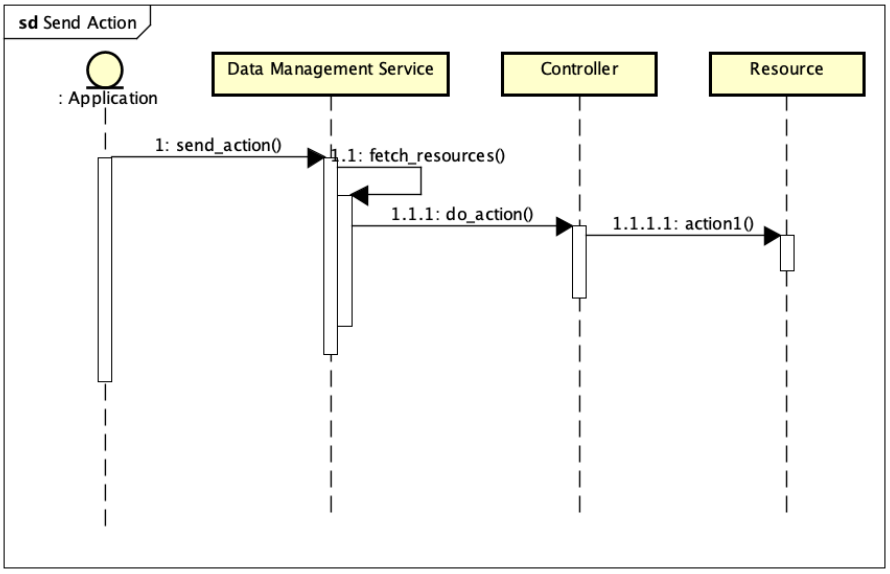

Figura 6 – Procedimento de envio de ações da aplicação para o *Resource*

de um *observable* funciona de maneira similar ao realizar uma ação, porém nesse caso o *Controller* fica responsável por monitorar quando esse evento ocorrer e então notifica o *middleware*, que repassa a mensagem de notificação para a aplicação. A Figura 7 descreve esse tipo de comportamento. Uma aplicação adiciona um evento via requisição POST ao *Data Management Service*, que registra esse evento a ser observado no respectivo *Controller* do *Resource*. O *Controller* irá monitorar esse evento a partir de agora e, quando ele ocorrer, irá enviar uma requisição PUT ao *Data Management Service* para que ele notifique a aplicação sobre a ocorrência de tal evento.

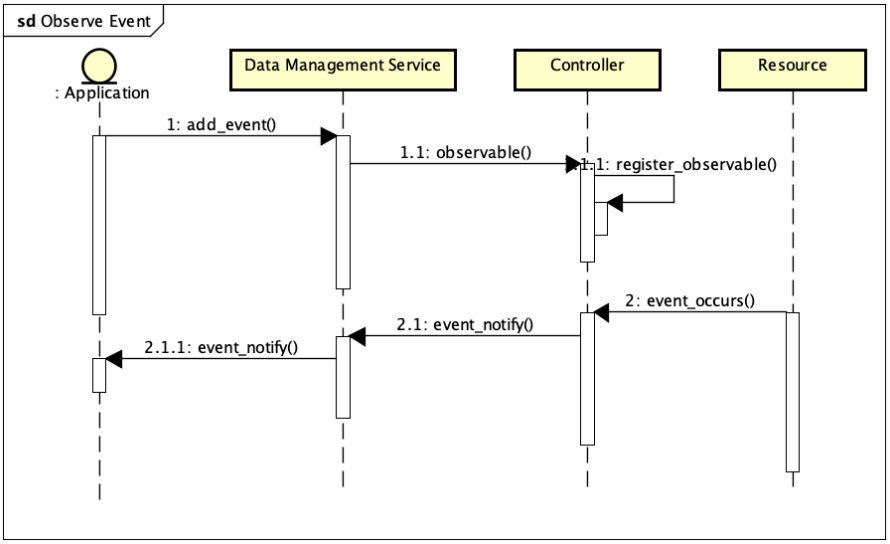

Figura 7 – Procedimento de registro de monitoramento de eventos em um *Resource*

O *middlware* desenvolvido nessa prova de conceito não implementa todos os módulos descritos na arquitetura FlyIoT, porém contempla seus principais módulos para atingir

seu objetivo principal, que é possibilitar a construção de uma linguagem com abordagem de controle de recursos em um nível de abstração mais elevado. Ao se beneficiar das facilidades do *middleware* baseado em FlyIoT, uma aplicação ou máquina de execução lidará com abstração de programação e de recursos, ou seja, toda a comunicação e o controle dos recursos IoT se darão por meio de requisições "http". O próprio *middleware* ficará responsável por repassar essas ações aos recursos descobertos. Isso torna opcional que a aplicação precise de detalhes sobre os recursos com os quais ela tem que lidar, pois cada recurso disponível será repassado para a aplicação em um nível mais elevado, bastando apenas uma requisição "http" de ação ou evento a ser observado informando o seu tipo.

Além disso, o controle de descoberta de recursos, monitoramento e distinção de capacidades para instanciação de componentes de serviço do *middleware* ficará a cargo do próprio *middleware*. A aplicação lidará com o comportamento de apenas 3 categorias de *Resources* especificados na arquitetura FlyIoT (*Input*, *Output* e *Processor*), e suas capacidades estarão sendo anunciadas pelos próprios controladores desses recursos, eximindo a aplicação da necessidade de conhecer toda a heterogeneidade normalmente presente em ambientes IoT.

Os serviços do *middleware* e informações sobre os recursos não precisam necessariamente estar presentes em um serviço de nuvem. Inicialmente uma instanciação de um *middleware* baseado em FlyIoT pode se estabelecer em uma rede local. Seguindo alguma política de distribuição e havendo permissão de acesso do usuário configurada na aplicação, o *middleware* poderá instanciar seus componentes na nuvem caso necessário.

5 LINGUAGEM FLYIOTL

5.1 Modelo da Linguagem

A Friendly Internet of Things Language (FlyIoTL) foi desenvolvida com o objetivo de abstrair ainda mais a forma com a qual aplicações devem lidar com os recursos em um ambiente IoT. Como apresentado no Capítulo 2, a maioria das soluções para IoT focam principalmente na aquisição, agregação e análise de dados, quase sempre exigindo que essas informações sejam enviadas para a nuvem. Além disso, a necessidade de provisão de informações de baixo nível também está presente em propostas que visam abstrair a complexidade ao controlar recursos IoT, como mencionado na Seção 2.2. Diante disso, a primeira versão do modelo da linguagem FlyIoTL foi projetada para permitir que o desenvolvedor tenha acesso e controle esses recursos de forma simplificada sem restringir suas funcionalidades.

O modelo da linguagem FlyIoTL permite representar *things* (*Resources*) de um ambiente IoT isentando o desenvolvedor da aplicação de lidar com questões intrínsecas ao meio, como comunicação, descoberta, disponibilidade e controle especifico de cada tipo de *thing*. Tal abordagem é possível ao adotar, sob a máquina de execução do modelo, a solução de *middleware* baseado na arquitetura FlyIoT. A Figura 8 ilustra o modelo da linguagem.

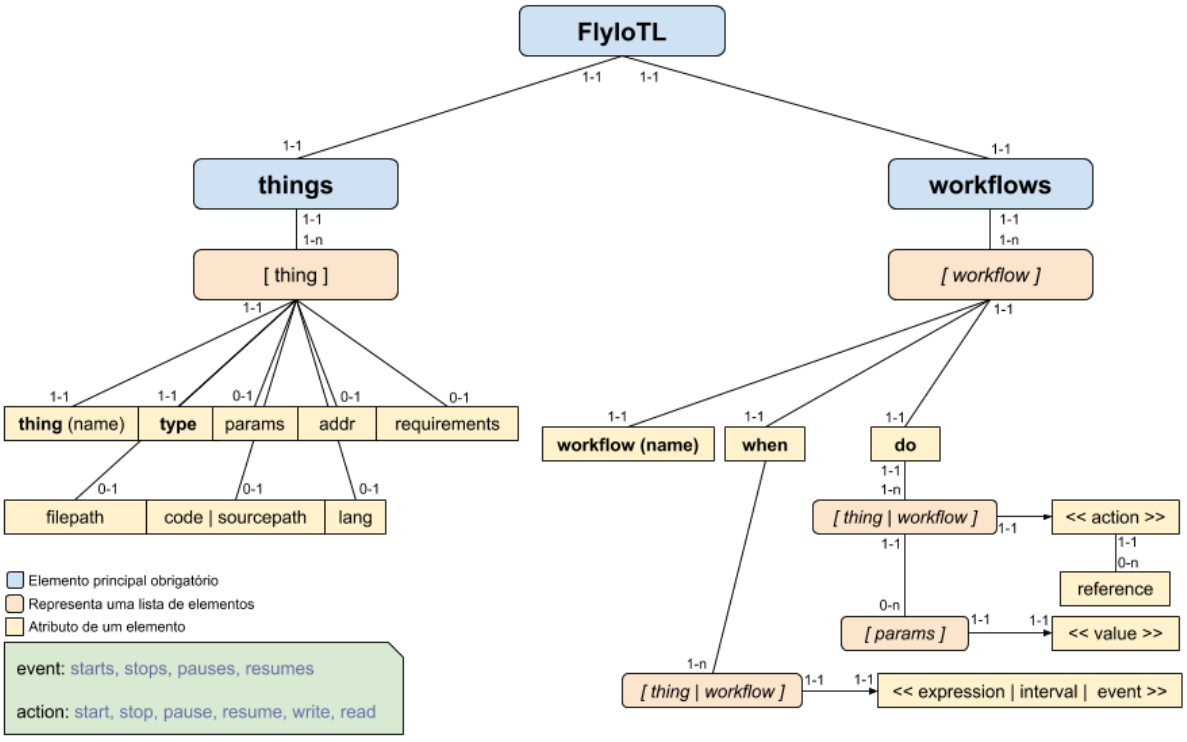

Figura 8 – Modelo da FlyIoT Language - FlyIoTL

Na Figura 8, os elementos representados por *"[elemento]"* fazem referência a listas do respectivo "elemento". Cada um desses "elementos" possui atributos obrigatórios, opcionais e até mesmo outros elementos filhos. Em **FlyIoTL**, existem dois elementos principais: *things* e *workflows*, além do elemento raiz *FlyIoTL*. O elemento *things* abrange uma lista com as definições referentes a cada *thing* que o desenvolvedor deseja utilizar em sua aplicação. As definições variam de acordo com as características de cada *thing*.

Do ponto de vista da máquina de execução que vai suportar a linguagem, existem basicamente três tipos de *things*: *Input*, *Output* e *Processor*. Para *Input*, tem-se qualquer recurso de entrada, incluindo sensores e dispositivos de interação. Alguns exemplos de *Inputs* são termômetros, sensores de presença, telas sensíveis ao toque, sensores de gestos, sensores ultrassônicos e todo recurso capaz de dar entrada com informações para o sistema (*Middleware* FlyIoT). Para *Output* estão quaisquer recursos de saída, como lâmpadas, *coolers*, *displays*, emissores de cheiro, emissor de efeitos sonoros e até *media players* (que é uma composição de atuadores, como *display*, caixas de som etc.). Por fim, para *Processor* estão os recursos capazes de executar sequências de código em alguma linguagem de programação, seja de forma interpretada ou compilada. Recursos de armazenamento podem ser considerados como *Input* ou *Output*, dependendo das necessidades da aplicação.

Na Figura 8, o elemento ***things*** é uma lista de elementos *thing*, os quais possuem propriedades em comum e propriedades específicas de acordo com o seu tipo. Tais propriedades são expressas por meio dos elementos *thing*, *type*, *params*, *addr*, *requirements*, *sourcepath*, *code* ou *filepath* e *lang*. Os elementos *thing* e *type* são obrigatórios.

***thing(name)*:** Representa o identificador na aplicação para cada *thing*. É similar a um identificador de variável definido pelo desenvolvedor em outras linguagens de propósito geral. Com esse identificador, o desenvolvedor pode associar esse *thing* com os demais dentro do documento da aplicação FlyIoTL.

***type*:** Referencia uma categoria ou um tipo de *Resource* descoberto pelo *middleware* FlyIoT. Um tipo nem sempre pode ser descoberto pelo *middleware*, porém poderá ser relacionado a outros tipos identificados no decorrer da execução da aplicação, como por exemplo, um *thermometer* ser associado a um *heatSensor*.

***params*:** Enumera parâmetros que o *middleware* possui sobre um determinado *type*. Esses parâmetros podem ser referenciados e ter seus valores alterados no decorrer da execução da aplicação. Por exemplo, um valor pode ser obtido a partir de um *thing* e passado como parâmetro de entrada a outro *thing* (Exemplo: Seção 6.2).

***addr*:** representa endereço individual útil de identificação do *thing* para o caso de ambientes pré configurados ou conhecidos. Esse endereço pode ser um identificador de *beacon* do dispositivo, endereço MAC, endereço IP, entre outros.

***requirements*:** Permite especificar requisitos para um determinado tipo de *thing*. O desenvolvedor pode exigir que, dentre os *things* descobertos, seja utilizado apenas o *thing* que possui tais *requirements*. Um exemplo seria um *thing* do tipo *media player* que suporte certos *codecs*, ou *thing* do tipo *touchscreen* com um conjunto específico de gestos multitoque.

***filepath*:** Essa propriedade está presente apenas em um *thing* do tipo armazenamento em modo leitura ou escrita. Ela representa a localização desse arquivo nesse recurso de armazenamento. Por exemplo, um arquivo que se encontra no *filepath* "/app/file.txt" em um determinado *thing* de armazenamento.

***sourcepath*:** É usado para que o interpretador da linguagem referencie um arquivo de programa, ou seja, um *script* (código executável, *bytecode*). Por exemplo, referenciar a localização de um *"script"* que será associado a um *thing* do tipo *Processor*.

***code*:** É usado para listar no próprio corpo do documento da aplicação FlyIoTL um código de *scripting*. Os elementos *sourcepath* e *code* são mutuamente exclusivos e, tanto para um quanto para o outro, o código será executado em um *thing* do tipo *Processor*.

***lang*:** Esse elemento deve ser usado em conjunto com o *sourcepath* ou *code*. Ele especifica em qual linguagem está escrito o *script*. Por exemplo, se for um *script Python*, o valor do elemento *lang* deve dizer qual ou quais versões do *Python* o *Processor* deve suportar.

O segundo elemento principal, *workflows*, é composto de uma lista de tarefas (*workflow*) a ser executada, condicionada à ocorrência de algum(s) evento(s). Cada elemento *workflow* possui 3 componentes expressos pelos elementos: *workflow*, que é o nome do próprio *workflow* e que poderá ser referenciado em outros *workflows*; *when*, no qual serão descritas as condições a serem levadas em consideração pelo presente *workflow*; e *do*, que conterá as ações que serão disparadas logo que as condições observadas em *when* forem verdadeiras.

***workflow(name)*:** Esse elemento, de forma análoga ao elemento *thing*, representa o identificador do *workflow* no documento da aplicação FlyIoTL e poderá ser referenciado em outros locais do próprio documento FlyIoTL.

***when*:** O elemento é composto por condições a serem observadas pela máquina de execução da linguagem. Tais condições podem ser observadas a partir de referências a *things* ou a outros *workflows*. As condições podem ser representadas a partir de expressões lógicas ou intervalos de valores observados. Existe também um conjunto de eventos

padrão que é observável: *starts*, *stops*, *pauses*, *resumes*. Existem também eventos que podem ser especificados de acordo com algum tipo de *thing*.

do*:** No elemento *do*, é especificado o conjunto de ações a serem tomadas pelo *workflow*. Tais ações podem ser realizadas tendo *things* ou *workflows* como alvos, assim como nas condições. Porém, nesse caso, pode haver um elemento *reference* que cria um identificador para um valor oriundo de alguma ação do *workflow*, que pode ser passado para outra ação dentro do mesmo *workflow*, como um parâmetro. O escopo de ***reference (Exemplo: Seção 6.2) se limita apenas ao próprio *workflow* no qual foi declarado. As ações padrão do modelo são *start*, *stop*, *pause*, *resume*, *write*, *read*. Cada *thing* possui parâmetros que são definidos a priori sob o elemento *things*. As ações podem variar de acordo com a origem, características e capacidades de cada *thing*.

5.2 Comportamento dos *things*

Normalmente é esperado que cada *thing* em um ambiente de Internet das Coisas possua um comportamento distinto devido a sua natureza heterogênea. No entanto, como FlyIoTL considera 3 tipos básicos de *thing* (*Input*, *Output*, *Processor*), obtém-se uma uniformização que simplifica substancialmente o processo de criação. Assim, tem-se comportamentos similares para todo atuador ou recurso de saída (*Output*) e todo sensor ou recurso de entrada (*Input*). Na Figura 9, pode ser observado um diagrama de estados referente aos *things* da linguagem FlyIoTL.

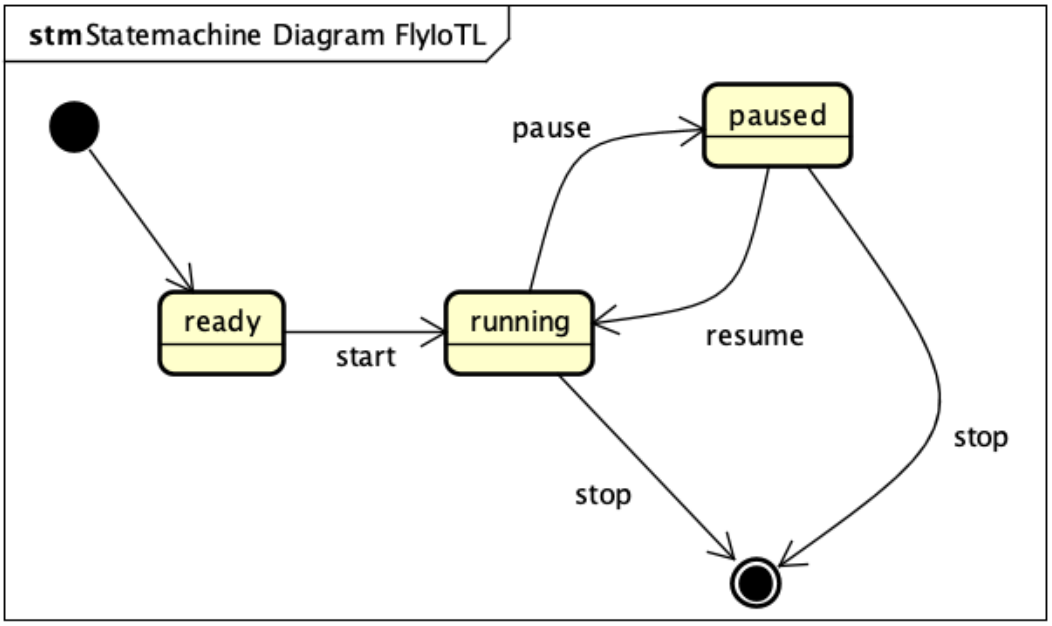

Figura 9 – Máquina de estados de things em FlyIoTL

A partir do momento em que um controlador de um *thing* é inicializado e o mesmo realiza todas as condições e preparações necessárias para que o *thing* funcione corretamente, ele entrará no estado de ***ready***. Após isso, o *thing* está pronto para ser inicializado a partir do momento em que receber um *start*, que então entrará no estado de ***running***. Enquanto estiver no estado de *running*, ele poderá ser pausado ao receber uma ação de *pause*, quando então entrará no estado de *pause*. Uma vez pausado, ele poderá retornar ao estado de ***running*** caso receba uma ação de *resume*. Tanto no estado ***paused*** quanto no estado *running*, ao receber uma ação de *stop*, o *thing* terá encerrado o seu ciclo de vida. No caso de sensores e atuadores, os seus comportamentos na máquina de estados são similares. Quando um sensor ou atuador estiver no estado de *ready*, poderá ter seu estado alterado conforme a figura 9 descreve.

Dentre *Inputs* e *Outputs*, existem os *things* do tipo *Storage*, que irão lidar com *things* arquivos virtuais. *Processors* também lidam com *things* virtuais, que são os códigos ou *scripts* que poderão ser acionados e executados a partir de aplicações escritas em FlyIoTL. Quando um controlador de um *thing* do tipo *Storage* entra no estado ***ready***, ele estará pronto para criar ou manipular um arquivo. Assim que recebe uma ação para manipular um arquivo, realiza as operações com o mesmo entrando no estado de ***running***. Em seguida é automaticamente parado após as operações de escrita. Ao final, o controlador do *Storage* pode se preparar novamente para entrar no modo *ready* de forma automática.

No caso de *things* do tipo *Processor*, a máquina de estados funciona de forma similar ao *Storage*. O controlador do *Processor* o prepara para entrar no modo ***ready***. Ao receber uma ação de *start*, ele executa o código referenciado na aplicação FlyIoTL, entrando no estado ***running***. Após finalizar a execução do código, ele é encerrado. Na sequência, o controlador do *Processor* pode voltar automaticamente ao estado de ***ready***.

Tanto para *Storages* ou *Processors* não faz sentido, a princípio, que eles possuam um estado de ***paused*** durante a execução da aplicação FlyIoTL. Portanto, as ações (*pause* e *resume*), que fazem referência a esse estado, podem ser desconsideradas por seus controladores. Essa abordagem se dá em função de que, ao se executar um *script* ou manipular um arquivo como escrita ou leitura, é incomum pausar a execução desse tipo de tarefa.

Em resumo, os três tipos de *thing* (*Input*, *Output* e *Processor*) passam a se comportar de maneira similar, baseados em uma mesma máquina de estados, porém com um pequeno detalhe: a não obrigatoriedade de *Processors* e *Storages* possuírem o estado de ***paused***. Do ponto de vista da gerência da aplicação FlyIoTL, isso facilita o controle do comportamento dos *things*, que anteriormente se mostravam muito mais heterogêneos e distintos.

6 PROVA DE CONCEITO DO INTERPRETADOR DE FLYIOTL

6.1 Implementação da *Execution Engine*

A *Execution Engine* (Máquina de Execução) é responsável por ler um arquivo contendo a aplicação FlyIoTL e então executá-la. Como é possível ver na Figura 10, a *Execution Engine* acaba se tornando bem simples devido a sua utilização apoiada sobre o *middleware* baseado em FlyIoT, que já abrange boa parte das tarefas mais complexas.

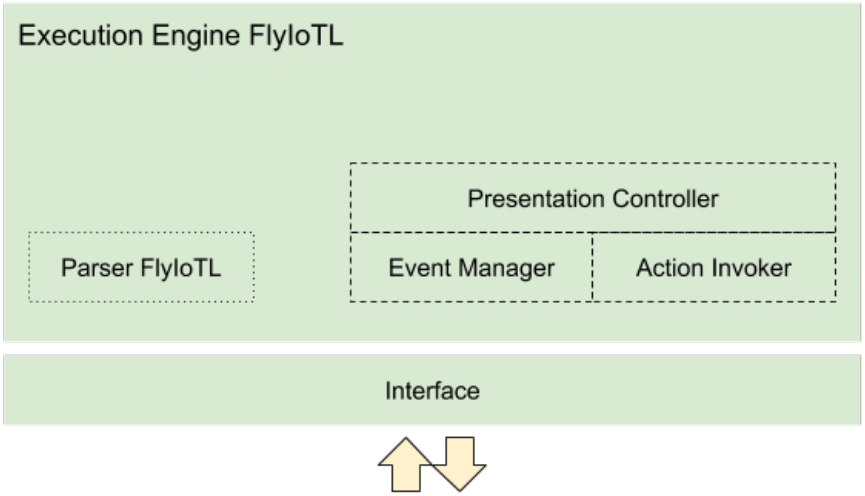

Figura 10 – Arquitetura da máquina de execução FlyIoTL

Na *Execution Engine* estão presentes alguns componentes: *Parser FlyIoTL*, *Presentation controller*, *Event Manager*, *Action Invoker* e *Interface*. O ***Parser FlyIoTL*** é o módulo responsável por analisar o arquivo contendo a aplicação *FlyIoTL* verificando sua sintaxe e identificando se está de acordo com as regras de elaboração do documento. Nessa etapa são analisados os parâmetros e eventos específicos utilizados nos *workflows* da aplicação, que só são permitidos caso tenham sido declarados no trecho de definição dos *things* da aplicação.

Após a etapa de validação do documento FlyIoTL, o ***Presentation Controller*** entra em ação acionando o ***Event Manager***, que elenca a lista de eventos a serem observados nos *things*. Nessa etapa é construída uma lista de eventos agrupada, na qual cada conjunto de eventos está organizada de acordo com seu respectivo *workflow*. Todo evento adicionado a essa lista recebe um *id* único composto por *uuid*, identificador do *workflow* a que pertence (elemento *"workflow"*) e também o identificador do respectivo *thing* (elemento *"thing"*). A lista de eventos é registrada no *Data Management Service*, que repassa esse registro de *observables* de eventos para os respectivos *things*.

Quando uma notificação de algum evento é recebida pelo *Execution Engine*, o *Event Manager* identifica a qual evento se refere (associação direta, pois o *id* do evento é

facilmente mapeável em *workflow* e *thing*) e repassa ao *Presentation Controller*, que irá analisar se as condições especificadas no *workflow* foram atendidas. Se essas condições forem atendidas, o *Presentation Controller* aciona o *Action Invoker*.

O ***Action Invoker*** vai montar a estrutura da mensagem de ação – com os respectivos parâmetros que foram especificados no documento da aplicação FlyIoTL – e encaminhar a mensagem para o *Data Management Service* do *middleware* por meio da *Interface*. O destino é direcionado para os grupos de *things* de acordo com o seu *type* especificado, exceto para definições de *things* individuais (por meio do atributo *addr*).

A ***Interface*** é responsável por intermediar a comunicação entre os componentes do próprio *middleware* e a *Execution Engine*. Esta, por sua vez, pode se comportar como um *Service* que expõe suas funções de comunicação ao *middleware* via API REST e também se anuncia como os demais componentes do *middleware*. Dessa maneira, ao iniciar sua execução, o *middleware* pode identificá-la e já prover uma prévia de *things* (*Resources*) já descobertos, tornando tais informações disponíveis para a aplicação FlyIoTL.

6.2 Modelo de aplicação em FlyIoTL

Para uma primeira instanciação da linguagem FlyIoTL, YAML (BEN-KIKI; EVANS; INGERSON, 2005) foi utilizado como base, devido à facilidade de estruturação e à natureza "limpa" do código, se comparado a alternativas como XML ou JSON. Além disso, o YAML é diretamente mapeável para a estrutura JSON, que é mais simples de ser representada computacionalmente e analisada do que uma estrutura de XML.

Conforme abordado no modelo da linguagem, um arquivo de documento da aplicação FlyIoTL possui duas partes principais: definição de *things* e definição de *workflows*. A seguir, serão apresentados os segmentos de um documento da aplicação FlyIoTL baseado em YAML a título de exemplo. O documento completo desta aplicação FlyIoTL pode ser encontrado no Anexo A.

No trecho inicial do documento, é declarado o elemento raiz que define o inicio da aplicação *"FlyIoTL"*. Em seguida vem o primeiro elemento básico (*"things"*), que é usado para definir a lista de *things*. Na sequência, o primeiro *thing* é definindo, atribuindo um nome identificador *"myThermometer"*, especificando seu tipo *"thermometer"* e um determinado parâmetro presente na lista de parâmetros, chamado *"temperature"*. Assim como o *"type"*, os *"params"* são providos pelo *middleware* para a *Execution Engine* no início da execução, para que possam ser validados.

```
FlyIoTL:
    things:
      - thing: myThermometer
```

```
    type: thermometer
    params: [ 'temperature' ]
```

Em seguida, é declarado um *thing* identificado como *"myUmiditySensor"*, que é do tipo *"umiditySensor"* e não possui parâmetros na aplicação.

```
  - thing: myUmiditySensor
    type: umiditySensor
```

Após o *"myUmiditySensor"*, é definido um *thing* do tipo *"fan"* e nome identificador *"eastFan"*. Diferente dos outros dois *things*, o *eastFan* é um *thing* conhecido e possui um identificador *beacon*. Esse campo também poderia ser usado para referenciar um *MAC address* ou um endereço *IP*, por exemplo.

```
  - thing: eastFan
    type: fan
    addr: 55AC567
```

O *thing* do tipo *"ledRGB"* a seguir é definido com nome identificador *"myLed"* e possui dois parâmetros: *"color"* e *"intensity?"*. Nesse caso, um parâmetro terminado em interrogação (?) significa que ele é desejável, mas opcional.

```
  - thing: myLed
    type: ledRGB
    params: [ 'color', 'intensity?' ]
```

Na sequência, é definido um *thing* do tipo *"touchSurface"*, identificado na aplicação pelo nome *"mySmartphone"*, que possui dois requisitos. Um requisito é uma especificação feita pelo desenvolvedor para que o *thing* atenda às necessidades informadas. São especificados dois gestos, *"swipeUp"* e *"swipeDown"* e que o *thing* deve suportar *multitouch*.

```
  - thing: mySmartphone
    type: touchSurface
    requirements:
        gestures: [ 'swipeUp', 'swipeDown' ]
        multitouch: true
```

Também é possível referenciar um *thing* com um perfil de *player* para apresentar conteúdo multimídia. É especificado um *thing* do tipo *"mediaPlayer"*, identificado por *"myPlayer"*, que possui um conjunto de requisitos de *codecs*, formatos e protocolos, sendo que alguns deles podem também ser opcionais, definidos com (?) ao final.

```
- thing: myPlayer
  type: mediaPlayer
  params: [ 'mediaurl', 'duration' ]
  requirements:
      codecs: [ 'h264', 'h265?', 'aac', 'mp3?' ]
      formats: [ 'isobmff', 'ts?' ]
      protocols: [ 'dash' ]
```

O *thing* do tipo *"inputStorage"* é especificado com a finalidade de referenciar arquivos a serem usados na aplicação. Nesse caso, o *thing* é definido com o nome de *"myFile"* e define um *"filepath"* referenciando a localização de volume e *path* do arquivo, além de um requisito no qual ele já deve existir a priori.

```
- thing: myFile
  type: inputStorage
  filepath: 'myApp:/myFile'
  requirements:
      mustExist: true
```

Para o *thing* do tipo *"outputStorage"* e nome de identificação *"myFile2"*, é especificado um *"filepath"* de localização do arquivo no *thing*. Além disso, aqui é definido um conjunto de requisitos de eventos a serem observados na aplicação FlyIoTL e um parâmetro como o nome *"data"*.

```
- thing: myFile2
  type: outputStorage
  filepath: 'myApp:/newFile'
  requirements:
      events: [ 'isCreated', 'isModified', 'isDeleted' ]
  params: [ 'data' ]
```

Por fim, a última definição de *thing* é do tipo *"processor"* e nome identificador *"myMonitoring"*. Nesse tipo de *thing* é especificada uma referência de *script* a ser executado pelo *thing*, passando o seu caminho como *"sourcepath"*, além do tipo de linguagem a ser suportada para execução (em *"lang"*). Como alternativa para não utilizar *"sourcepath"*, é possível implementar o trecho de código imperativo diretamente no documento FlyIoTL, atribuindo-o como valor ao elemento *"code"* seguido de "|" (que, em um documento YAML, reconhece todo o trecho de texto posterior e identado como valor do elemento definido anteriormente, preservando sua formatação).

```
    - thing: myMonitoring
      type: processor
      lang: 'python3'
      sourcepath: './path/to/file.py'
#     code: |
#       import time
#       ts = time.time()
#       print(ts)
      params: [ 'temperature' ]
```

Na segunda parte de uma aplicação FlyIoTL, define-se a lista de *workflows*. Nela estão presentes os eventos a serem observados e as ações que devem ser realizadas de acordo com o acontecimento de cada tipo de evento.

O primeiro *workflow*, com identificador *"w1"*, especifica que, quando um termômetro (*"myThermometer"*) marcar entre 33 e 44 graus e o sensor de umidade (*"myUmiditySensor"*) marcar menos que 50 %, a temperatura do termômetro deve ser lida, referenciando-a como *"temp1"*. O valor será passado como parâmetro ao *script* de monitoramento (*"myMonitoring"*), que será inicializado junto com um ventilador específico *"eastFan"*. No caso do uso de intervalos ao observar um evento, toda leitura dentro desse intervalo será notificada. Ainda que ocorra apenas uma leitura, isto já será suficiente para acionar o *workflow* caso as demais condições sejam verdadeiras.

```
workflows:
  - workflow: w1
    when:
      - myThermometer: '[33-40]'
      - myUmiditySensor: '<50%'
    do:
      - myThermometer: read
        reference: temp1
      - myMonitoring: start
        temperature: temp1
      - eastFan: start
```

O *workflow* seguinte, que possui identificador *"w2"*, especifica que, quando for detectado um evento de *"swipeUp"* de uma tela *touchScreen* (*"mySmartphone"* no caso), uma lâmpada led (*"myLed"*) será acionada e configurada para a cor branca (*"#ffffff"*), com intensidade ajustada para 80 % (*"0.8"*), caso o dispositivo possua esse ajuste.

```
    - workflow: w2
      when:
```

```
      - mySmartphone: 'swipeUp'
    do:
      - myLed: start
        color: '#ffffff'
        intensity: '0.8'
```

O terceiro exemplo de *workflow* é o *"w3"*. De modo similar ao *workflow "w2"*, quando for detectado um evento de *"swipeDown"* na tela *"touchScreen"* do *"mySmarphone"*, a lâmpada será ajustada para a cor travertino (*"#fffde7"*) com intensidade de 30 % (caso suporte esse tipo de ação).

```
- workflow: w3
  when:
    - mySmartphone: 'swipeDown'
  do:
    - myLed: start
      color: '#fffde7'
      intensity: '0.3'
```

No *workflow "w4"*, quando os *workflows "w2"* e *"w3"* detectarem um evento de parada (*"stop"*), seja por fim natural de seu fluxo ou por alguma ação de *"stop"*, o *thing "myMonitoring"* deverá ser inicializado juntamente com o *"myFile2"*, resultando na criação do arquivo definido no *thing "myFile2"*.

```
- workflow: w4
  when:
    - w2: stops
    - w3: stops
  do:
    - myMonitoring: start
    - myFile2: start
```

No quinto *workflow*, com identificador *"w5"*, quando um arquivo for criado em *"myFile2"*, o conteúdo de *"myFile"* deverá ser lido, referenciado via *"nf"* e passado como parâmetro para ser escrito no *"myFile2"*. Além disso, o valor de *"myThermometer"* deverá ser lido e passado como parâmetro para o *"myMonitoring"* via referência *"temp5"*.

```
- workflow: w5
  when:
    - myFile2: isCreated
  do:
    - myFile: read
```

```
        reference: nf
      - myFile2: write
        data: nf
      - myThermometer: read
        reference: temp5
      - myMonitoring: start
        temperature: temp5
```

No *workflow* seguinte, nomeado como *"w6"*, quando o *workflow* definido como *"w1"* ocorrer, ou seja, suas condições de execução forem acionadas, o *workflow "w6"* irá inicializar o *thing "myPlayer"*, passando os valores de parâmetro *"mediaUrl"* contendo a *url* do local onde o player deve buscar o conteúdo a ser exibido. Determina-se ainda um tempo de duração máxima, neste caso pré-fixado em 50 segundos.

```
- workflow: w6
  when:
    - w1: starts
  do:
    - myPlayer: start
      mediaurl: 'file://myApp/path/to/file'
      duration: '50s'
```

Por fim, no *workflow "w7"*, quando for detectado que o *"myPlayer"* atingiu 10 segundos de reprodução, será acionado o *thing "eastFan"* e desligado o *thing "myLed"*.

```
- workflow: w7
  when:
    - myPlayer: '=10s'
  do:
    - eastFan: start
    - myLed: stop
```

Com a linguagem FlyIoTL, aliada às facilidades do *middleware* baseado na arquitetura FlyIoT, é possível elevar o nível de abstração para o controle de *things* presentes em um ambiente IoT. O desenvolvedor pode se preocupar apenas em definir e criar os relacionamentos entre os diversos tipos de *things* desejáveis de maneira declarativa e o *middleware*, juntamente com a máquina de execução da linguagem FlyIoTL, se encarregará de prover, sempre que possível, os recursos para a aplicação. Problemas relacionados a conexão, preparação e execução em cada um dos *things* ficará a cargo dos seus próprios controladores e serviços do *middleware*. Além disso, o controle feito para identificar quais ações e eventos deverão ser acionados com base no que está disponível para a aplicação, será de responsabilidade da máquina de execução FlyIoTL.

7 CASOS DE USO

Para demonstrar a solução proposta, foram implementados dois cenários de caso de uso: um para ambiente preparado e outro para ambiente não preparado. Os ambientes descritos a seguir são simples, e uma validação mais precisa requer testes com maior nível de aprofundamento em trabalhos futuros, dialogando com outras propostas de *middleware* na literatura. Nos cenários aqui apresentados, o objetivo se limita a demonstrar como os requisitos de abstração de programação e de recursos são cumpridos, ou seja, como a implementação do *middleware* baseado na arquitetura e na linguagem propostas eleva o nível de abstração no controle e acesso aos recursos IoT.

Um ambiente preparado é aquele em que os módulos de serviço do *middleware* já estão previamente instanciados e não há novas instanciações de componentes de serviço durante a execução da aplicação FlyIoTL. Além disso, os *things* a serem descobertos são conhecidos *a priori* e podem ser referenciados por meio do parâmetro *"addr"* no momento de sua definição no documento FlyIoTL. Um ambiente preparado se assemelha a muitas soluções IoT utilizadas atualmente, nas quais cada dispositivo deve ser configurado para se comunicar com um serviço central agregador (geralmente na nuvem). Em geral, este trabalho de configuração manual tende a se tornar árduo e repetitivo.

No segundo caso de uso, um ambiente não preparado, pode não haver informações sobre os componentes do *middleware* ou *things* que estão ativos no momento da instanciação da máquina de execução. Os demais módulos essenciais do *middleware* e *things* devem ser descobertos a partir de um ponto inicial. À medida que os *things* são descobertos, as instâncias dos módulos do *middleware* podem ser inicializadas para que, uma vez satisfeitas, tenha início a execução da aplicação FlyIoTL.

A título de casos de uso para a prova de conceito, os dois tipos de ambientes serão apresentados na seção a seguir, com o detalhamento sobre implementações e destaques sobre o potencial da solução proposta. O foco neste capítulo é demonstrar como as funcionalidades especificadas na arquitetura FlyIoT, aliadas à linguagem FlyIoTL, beneficiam o processo de criação de aplicações IoT. Facilidades no quesito de abstração de programação, controle de recursos descobertos e distribuição da instanciação dos componentes do *middleware* ganham destaque, pois tais processos são transparentes para o desenvolvedor da aplicação. Para demonstração dos casos de uso, foram utilizados os seguintes equipamentos de *hardware*:

Computador: *Imac 21" Late 2013*, *Intel Core I5-2.7GHz*, *8GB* de memória *RAM*, *480GB* de *SSD*, *MacOS 10.14.6 Mojave*

Computador: *Notebook Dell Latitude 3440*, *Intel Core I7-1.8GHz*, *8GB* de memória *RAM*, *1000GB* de *HD*, *Linux Mint 19.2 Tina*

***Single Board Computer*:** *Raspberry Pi 3 Model B*, *BCM2837 1.2GHz*, *1GB* de memória *RAM*, 16GB *microSD*, *Raspbian (Debian 10 Buster)*

***Smartphone*:** *Xiaomi Redmi 5*, *Snapdragon 450*, *3GB* de memória *RAM*, *32GB* de armazenamento

***Smartphone*:** *Xiaomi Mi 8 Lite* , *Snapdragon 660*, *6GB* de memória *RAM*, *64GB* de armazenamento

Sensor Módulo *Arduino switch button KY-004*

Atuador Módulo *Arduino LED RGB KY-009*

7.1 Ambiente preparado

Nesse caso de uso, todos os componentes do *middleware* e recursos a serem descobertos já são esperados, ou seja, o desenvolvedor já está ciente sobre as capacidades do ambiente IoT. Portanto, o objetivo é demonstrar como FlyIoT facilita o controle de eventos e ações dos *things* ao elevar o nível de abstração. Para montar esse cenário, cada componente do *middleware* foi executado a partir da instanciação da *Execution Engine* juntamente com o documento da aplicação FlyIoTL. Os demais componentes do *middleware* foram instanciados seguindo uma ordem de execução. O ambiente está representado conforme a Figura 11.

O componente de *Data Management Service* foi configurado previamente no *middleware* (arquivo de configuração do *Management Service* "config.py") para que seja executado no serviço de nuvem (Google Cloud Platform) com base nas credenciais do usuário (desenvolvedor). Os demais componentes do middleware são instanciados na rede local (*Discovery Service* e *Management Service*). Nesse caso de uso, os componentes do *middleware* não serão instanciados novamente no decorrer da execução, pois o ambiente já é conhecido pelo desenvolvedor.

O documento completo da aplicação FlyIoTL pode ser visualizado no Anexo B. Nesse documento, todos os *things* possuem uma chave *"addr"*, que contém um endereço IP ou um endereço MAC. A aplicação FlyIoTL descreve o comportamento dos *things* conforme a ocorrência dos eventos monitorados e validação em seus *workflows*. Ela é executada por uma *Execution Engine* que também será instanciada. O documento da aplicação FlyIoTL é inicializado juntamente com a *Execution Engine*, que vai validá-lo de acordo com o modelo da linguagem apresentado no Capítulo 5. A *Execution Engine* , durante sua inicialização, é descoberta pelo *Discovery Service* e passa a ter acesso ao *Data Management Service*, bem como às informações dos *Resources* (*things*) já descobertos pelo *middleware*.

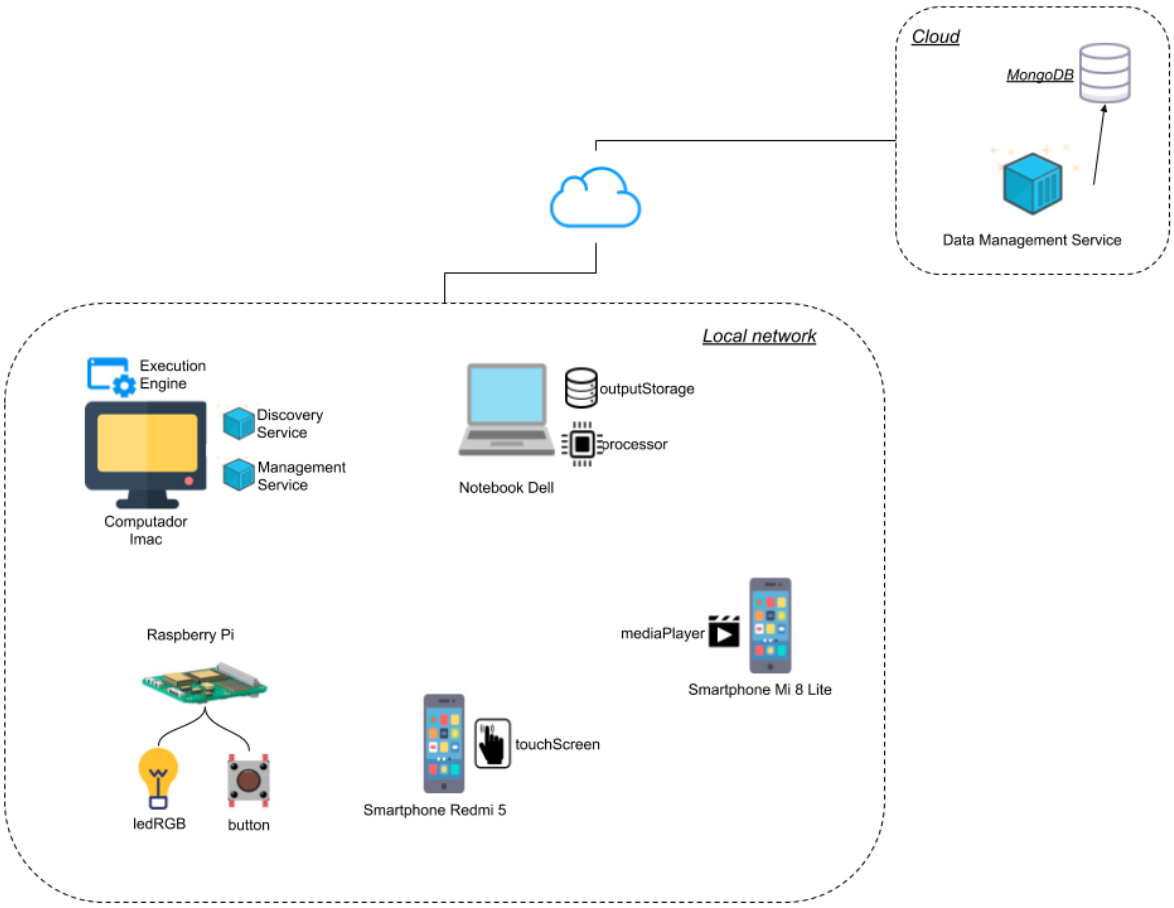

Figura 11 – Caso de uso para ambiente preparado

Nesse caso de uso, foram instanciados alguns *things*: um *mediaPlayer*, *touchSurface*, *ledRGB*, *button*, *storageOutput* e um *processor*. A aplicação FlyIoTL descreve os relacionamentos de eventos entre os *things* e por meio do *Presentation Controller* e do *Event Manager* da *Execution Engine* elenca quais eventos devem ser observados e registrados no *Data Management Service*.

```
- workflow: w1
  when:
    - myButton: starts
  do:
    - myPlayer: start
      mediaurl: 'rtsp://10.10.10.207:5544/'
    - myLed: start
      color: '#ff0000'
```

O primeiro *workflow* (*"w1"*), conforme mostrado acima, descreve que quando o botão for pressionado, deverá iniciar a execução do *mediaPlayer* consumindo o conteúdo presente na *url* passada como parâmetro e deverá também acender o LED RGB com uma cor vermelha. Um exemplo de evento gerado e registrado no *Data Management*

Service pode ser visualizado abaixo. Uma lista desses eventos é elencada e gerada, sendo registrados no *Data Management Service* que, por sua vez, repassa os eventos para o respectivo *thing*, que no caso referencia apenas um *Resource* para cada *type*, pois estão identificados individualmente (por meio do valor de *addr*) no cenário de ambiente preparado atual. O evento registrado referente ao *thing* button pode ser visto a seguir (Demais eventos podem ser conferidos no Anexo B.1).

```
[{
    "id": "572033aa-c3ba-11e9-8d35-685b357d8a1a_w1_when_0_button",
    "condition": "starts",
    "thing": "button",
    "addr": "51-DA-94-58-D1-1E"
}]
```

Cada *Controller* dos respectivos *things* possui eventos *observáveis* que são registrados conforme solicitados pela *Execution Engine*. Quando um determinado evento ocorre, o *Controller* do *thing* notifica o *Data Management Service*, por meio da *Interface*, informando o *id* do evento ocorrido. Cabe ao *Data Management Service* repassar essa informação para a *Execution Engine*, a qual vai analisar os *things* envolvidos no respectivo *workflow*, verificando quais eventos já foram satisfeitos para dar início ao gatilho da(s) ação(es), por meio do seu *Action Invoker*. A troca de mensagens entre *Execution Engine* e *things* poderia ocorrer diretamente, a partir do momento que a *Execution Engine* já possui as informações dos *things* providas pelo *middleware*. Mas, para manter a estrutura definida de arquitetura *Application/ExecutionEngine->middleware* FlyIoT->*things* e vice-versa, tal abordagem foi mantida.

Todos os *Controllers* dos *things* implementados para o caso de uso reconhecem por padrão os eventos de *"starts"*, *"stops"*, *"resumes"*, *"ends"*, além das ações padrão da linguagem FlyIoTL *"start"*, *"stop"*, *"pause"*, *"resume"*, *"write"* e *"read"*. Porém, para alguns *things*, eles podem aceitar ou simplesmente ignorar determinado tipo de ação ou evento que não lhes dizem respeito, como é o caso do *thing* do tipo *button* ao receber uma ação de *write*. Além disso, alguns *things* como o *myPlayer*, *MyTouchSurface* e *MyLed* possuem parâmetros adicionais, definidos no documento FlyIoTL, que reagem de forma distinta um do outro de acordo com os eventos.

```
  - thing: myPlayer
    type: mediaPlayer
    params: ['mediaurl']
    addr: '10-FB-75-65-9A-8B'
    requirements:
      codecs: ['h264', 'aac', 'mp3?']
```

```
      formats: ['ts']
      protocols: ['dash']
```

```
  - thing: myTouchSurface
    type: touchSurface
    addr: 'E7-39-AB-C5-E6-F7'
    requirements:
      gestures: [ 'swipeUp', 'swipeDown', 'swipeLeft', 'swipeRight' ]
      multitouch: true
```

```
  - thing: myLed
    type: ledRGB
    addr: 'E4-DF-60-B9-02-1F'
    params: ['color', 'intensity']
```

Quando um evento é registrado com sucesso em um *thing*, seu *Controller* começa a monitorá-lo para notificar ao *middelware* sempre que tal evento ocorrer. Apesar de exigir um pouco mais de esforço por parte dos *Controllers* de alguns *things*, como o *mediaPlayer*, esta abordagem economiza no uso da rede ao evitar sobrecarga no fluxo de mensagens, as quais são enviadas somente quando determinada condição é atendida.

Os *things* do tipo *storage* e *processor* têm comportamentos distintos. Um *storage*, mais especificamente o *outputStorage* – como na aplicação FlyIoTL em questão –, possui uma definição de *"filepath"* que faz referência a um arquivo. Também são declarados nesse caso os eventos de criação, modificação e remoção do arquivo. Tal *thing* é referenciado no *workflow "w5"*, no qual seu valor é escrito adicionando uma nova linha no arquivo declarado com o valor atual do tempo de reprodução do *mediaPlayer* sempre que ele for pausado. Além disso, o *workflow* também altera a cor do *ledRGB*.

```
  - thing: myLed
    type: ledRGB
    addr: 'E4-DF-60-B9-02-1F'
    params: ['color', 'intensity']
```

```
  - workflow: w5
    when:
      - myPlayer: pauses
    do:
      - myLed: 'start'
        color: '#ffff00'
      - myPlayer: read
        reference: currenttime
```

```
      - myFile: 'write'
        data: 'currenttime'
```

Por fim, o *thing* do tipo *processor* é declarado tendo um *sourcepath* como base. Nesse caso, o *script* realiza uma simples requisição *web* a um servidor externo (Firebase) ao ambiente IoT em questão. O servidor envia uma notificação para o smartphone do usuário nele configurado, sempre que ocorrer o evento descrito no *workflow "w6"*, ou seja, quando as ações referenciadas no *workflow "w2"* também ocorrerem.

```
- thing: myScript
  type: processor
  lang: 'python3'
  sourcepath: './scripts/start.py'
```

```
- workflow: w6
  when:
    - w2: stops
  do:
    - myLed: stop
    - myTouchSurface: stop
    - myScript: start
```

```
- workflow: w2
  when:
    - myButton: starts
    - w1: starts
  do:
    - myPlayer: stop
    - myLed: stop
```

Também é possível, a partir do último exemplo de *workflow* do caso de uso descrito, integrar um ambiente instanciado com o *middleware* baseado em FlyIoT a outros ambientes não necessariamente IoT. Apesar de simples, tal integração permite demonstrar como é possivel complementar funcionalidades ainda não suportadas pela arquitetura FlyIoT e a linguagem FlyIoTL, sem recorrer a implementações e alterações complexas no *middleware*. Assim como em outras propostas na literatura (XIVELY..., s.d.), (KIM; LEE, 2014a), (SALEME et al., 2019), o FlyIoT também suporta ambientes mais estáticos e pode atender a públicos mais específicos.

A abordagem utilizada em um ambiente preparado é bastante comum em aplicações IoT. Normalmente elas são desenvolvidas a fim de controlar os recursos de dispositivos IoT para solucionar um problema específico. Para montar esse tipo de cenário sem o uso de

FlyIoT, o desenvolvedor deveria se preocupar em configurar cada um dos *things* ou recursos presentes nesse ambiente como, por exemplo, o protocolo de comunicação suportado por cada um, suas capacidades, se estão acessíveis ou não. Alguns desses recursos nem possuem suporte a redes e conexões externas, como é o caso de processadores e volumes de armazenamento. Alguns sensores e atuadores mais flexíveis até oferecem API's para acesso e controle, porém cada um deles tem padrões e funcionamentos distintos e geralmente é limitado ao que os fabricantes determinam. Dispositivos mais simples e de propósito geral necessitariam de uma implementação em nível mais baixo como programação de placas microcontroladoras integradas. E mesmo dispositivos mais robustos, como *smartphones*, necessitariam de certo preparo – como aplicativos móveis que se comuniquem com a rede ou serviço de nuvem – para estarem aptos a realizar alguma função integrada a redes IoT.

Outra opção seria utilizar algum dos *middlewares* e plataformas estudados no Capítulo 2. Plataformas como *Xively* ou *OpenIoT* permitiriam certa facilidade na integração de sensores e atuadores por parte do desenvolvedor. Porém, todos os sensores e atuadores teriam que possuir acesso a Internet e provavelmente serem certificados junto à plataforma para só então serem integrados, como é o caso do *Xively*. Mesmo assim, tais plataformas não suportam dispositivos como processadores e volumes de armazenamento como FlyIoT, o que tornaria esse tipo de funcionalidade ainda mais difícil. O controle de processadores e dispositivos de armazenamento também precisaria ser feito pela aplicação do desenvolvedor. De fato, essas plataformas IoT são mais robustas do ponto de vista de segurança dos dados se comparado à FlyIoT, porém todo o fluxo de dados deve ser armazenado na nuvem e tal abordagem pode não ser interessante para o desenvolvedor.

Com a utilização da linguagem FlyIoTL – aliada ao *middleware* baseado na arquitetura FlyIoTL – nesse cenário de ambiente preparado, o desenvolvedor precisa se preocupar apenas em desenvolver o documento declarativo da linguagem, configurar o componente que deve ser executado no serviço de nuvem juntamente com suas credenciais e instanciar a *Execution Engine* passando o documento FlyIoTL como parâmetro. A partir disso, o *middleware* e a *Execution Engine* se encarregam de preparar e controlar os recursos na rede de acordo com o que foi especificado pelo desenvolvedor no documento FlyIoTL. Com isso, os requisitos de abstração de programação e de recursos elencados nos capítulos anteriores são atendidos por FlyIoT ao homogeneizá-los e categorizá-los em apenas três tipos (entrada, saída e processadores) para que o desenvolvedor tenha acesso e controle no documento da aplicação FlyIoTL.

7.2 Ambiente não preparado

Ao contrário do caso de uso anterior, neste foi utilizado um cenário onde o ambiente não estava inicialmente preparado. O objetivo aqui é demonstrar como o *middleware* se comporta ao ser instanciado em um ambiente sem informações sobre os *things*, processo de

instanciação e distribuição dos seus componentes, para permitir a execução da aplicação FlyIoTL. A aplicação utilizada é similar ao cenário anterior, do ponto de vista de alguns *things* e *workflows* declarados no documento FlyIoTL. Importante destacar que os *things* declarados no documento não possuem o elemento *"addr"*, pois não há informações sobre a existência e capacidades do *Resources*. O documento completo da aplicação FlyIoT utilizada nesse caso de uso pode ser conferido no Anexo C.

Para inicialização do cenário, uma aplicação FlyIoTL é executada juntamente com a *Execution Engine* e, diferente do caso de uso anterior, não existem informações sobre *Discovery Service* ou demais componentes do *middleware* FlyIoT que já foram instanciados. Logo o documento FlyIoTL é apenas validado em um primeiro momento. Com exceção desse caso específico, a *Execution Engine* executa um *"micro script"* de descoberta (que ela carrega para esses casos) em seu *"register.py"*, que irá buscar na rede local apenas por *processors* capazes executar os componentes do *middleware* FlyIoT, mais especificamente uma instância do *Management Service*. Nesse caso, existia um computador (*Notebook Dell*) se anunciando como um *processor* capaz de executar *scripts Python 3* e também capaz de instanciar e executar *containers Docker*. Então o *"micro script"* da *Execution Engine* envia uma mensagem, por meio de sua *Interface* REST, contendo um procedimento *Python* que será executado no *Processor* em questão, que, por sua vez, irá inicializar o download da imagem referente ao *Management Service*. Logo em seguida, o *container* com o respectivo *Service* é instanciado.

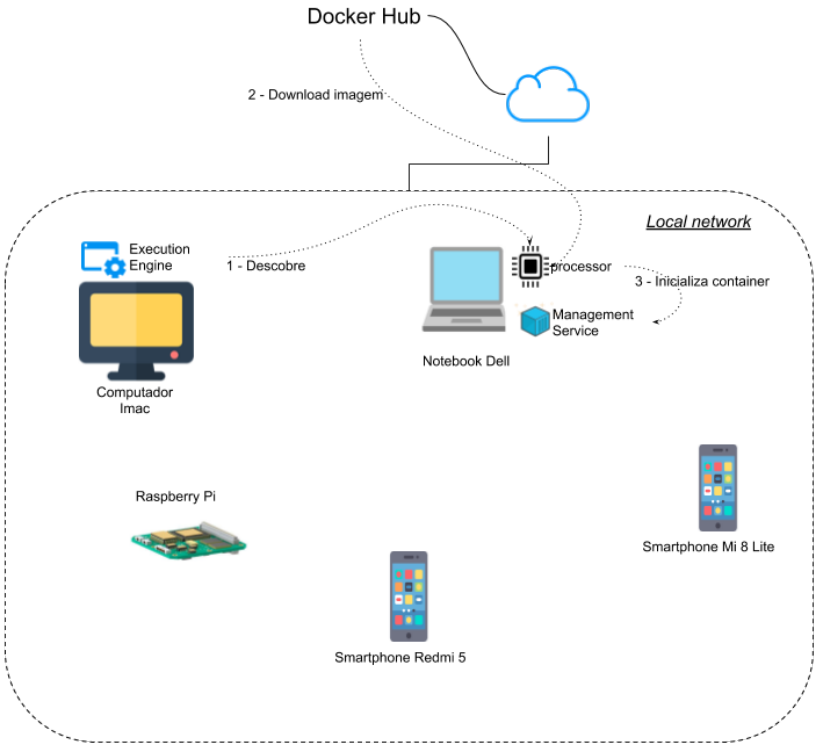

Figura 12 – Caso de uso para ambiente não preparado - Fase 1

Quando o *Management Service* iniciar sua execução e identificar que não foi localizado nenhum *Data Management Service* ou *Discovery Service*, o processo de instanciação dos mesmos será iniciado no próprio *Processor*, visto que as dependências entre os componentes de um *middleware* baseado em FlyIoT são semelhantes. Além disso, aproveitando-se dessa questão, as imagens dos *Services* do *middleware* de prova de conceito construídas previamente e disponibilizadas na nuvem (Docker Hub), são especificadas sobre uma mesma base de dependências, ou seja, utilizando-se a tecnologia de virtualização via *containers*, em que uma imagem pode ser construída por meio de camadas. Dessa forma, toda vez que for necessário instanciar um *container* com base em imagens semelhantes, apenas as camadas distintas são baixadas e construídas ao invés realizar o *download* e construção de toda a imagem novamente. As figuras 12 e 13 ilustram tal comportamento inicial do *middleware*.

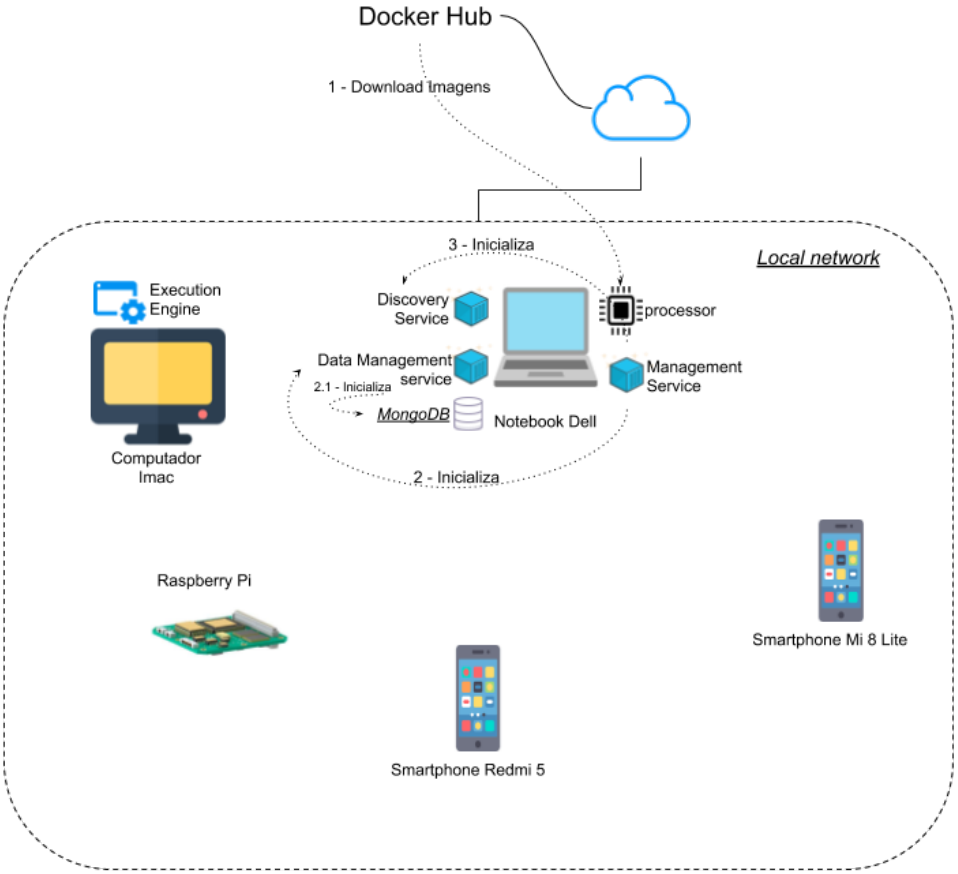

Figura 13 – Caso de uso para ambiente não preparado - Fase 2

Como já mencionado no Capítulo 3, o propósito desse trabalho não é determinar políticas de distribuição, mas adotar uma que permita demonstrar o funcionamento de um *middleware* baseado em FlyIoT. Quando um *Processor* é descoberto e escolhido para instanciar componentes do *middleware*, o limite de instâncias ativas simultaneamente está condicionado à quantidade de núcleos desse *Processor* (informação disponibilizada quando

ele se anuncia na rede). Se este limite for excido, o *middleware* tentará realocar instâncias de *Service* para algum outro *Processor* que venha a ser encontrado.

Uma vez instanciado, o *Discovery Service* buscará por *Resources* de todos os tipos: *Inputs* (sensores), *Outputs* (atuadores) e *Processors*. Sempre que um *Resource* for encontrado, ele será registrado no *Data Management Service*. Diferente do caso anterior, aqui alguns *things* foram inicializados em um mesmo dispositivo: um *mediaPlayer* e um *touchSurface* foram instanciados em dois *smartphones*; e um *storage* tanto no *Imac*, quanto no *Notebook*. Como o *uuid* é gerado no momento da inicialização do *thing*, os recursos se anunciam como *things* distintos, porém de mesmo tipo para alguns deles. Mais detalhes podem ser visualizados na Figura 14.

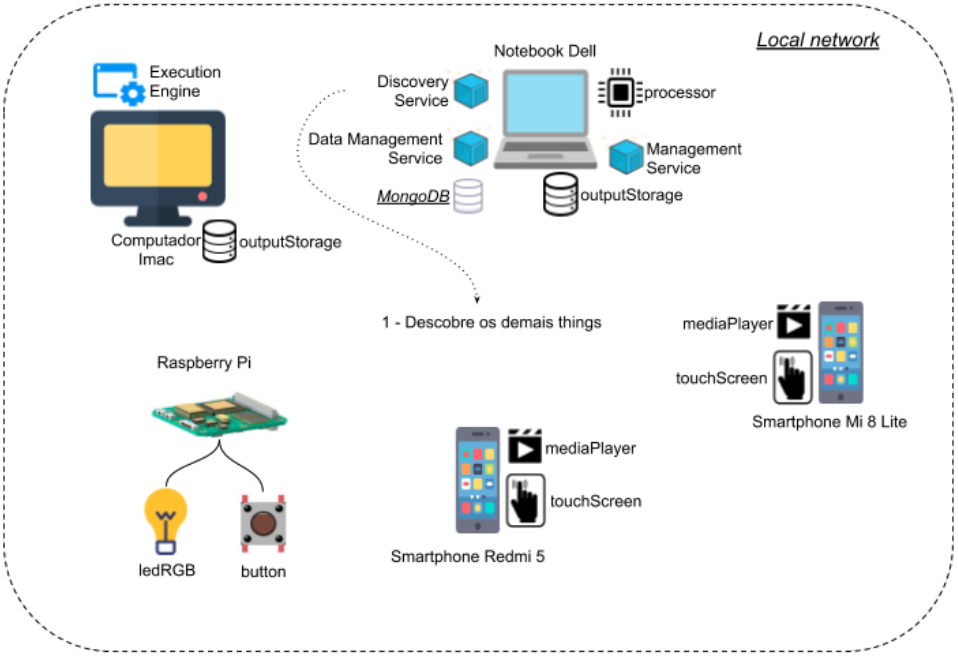

Figura 14 – Caso de uso para ambiente não preparado - Fase 3

Como descrito no Capítulo 3, o *Management Service* possui um módulo *System Monitoring*, que monitora os *things* e *Services* essenciais de FlyIoT. Quando um *thing* se torna indisponível, o *Management Service* remove o respectivo *thing* do *Data Management Service*. Porém, quando o *System Monitoring* detecta a indisponibilidade de um *Service*, ele aciona o *System Orchestration*, que elege um novo *thing* do tipo *Processor* capaz de instanciar um novo *container* do respectivo *Service*. No caso de uso em questão, após a instanciação dos *containers* de *Data Management Service* e *Discovery Service*, foi injetado um comando via terminal no *Notebook* para "matar" e remover o *container*, com a finalidade de testar essa função do *Management Service*. Um novo *container* do *Data Management Service* foi inicializado, as informações foram atualizadas nos demais *Services* e nenhum dado do banco de dados foi perdido, pois existe um segundo *container* – especificamente destinado ao banco *MongoDB* – executando em paralelo ao *Data Management Service*.

Tal abordagem é uma boa prática, comumente adotada em ambiente *DevOps*, (TOSATTO; RUIU; ATTANASIO, 2015) em que cada serviço é implantado em um *container* específico, com suas próprias dependências distintas, minimizando a complexidade de cada serviço. Além disso, até mesmo se o *container* do banco de dados fosse removido, nenhum dado seria perdido, pois o *container* mapeia os dados salvos para uma área de armazenamento do dispositivo, ou seja, somente o sistema de gestão do banco de dados seria afetado.

Nesse cenário de ambiente não preparado, como a *Execution Engine* não possuía previamente informações sobre um *Data Management Service*, ela executou apenas a validação do documento FlyIoTL. Ficou a cargo do *Management Service* notificar a *Execution Engine* sobre a disponibilidade do *Data Management Service* e *Discovery Service*. Nesse ponto, os *things* já foram descobertos e estão acessíveis para que a *Execution Engine* inicie a construção da lista de eventos a serem observados, de maneira similar ao caso de uso da seção anterior. Essa etapa pode ser visualizada na Figura 15.

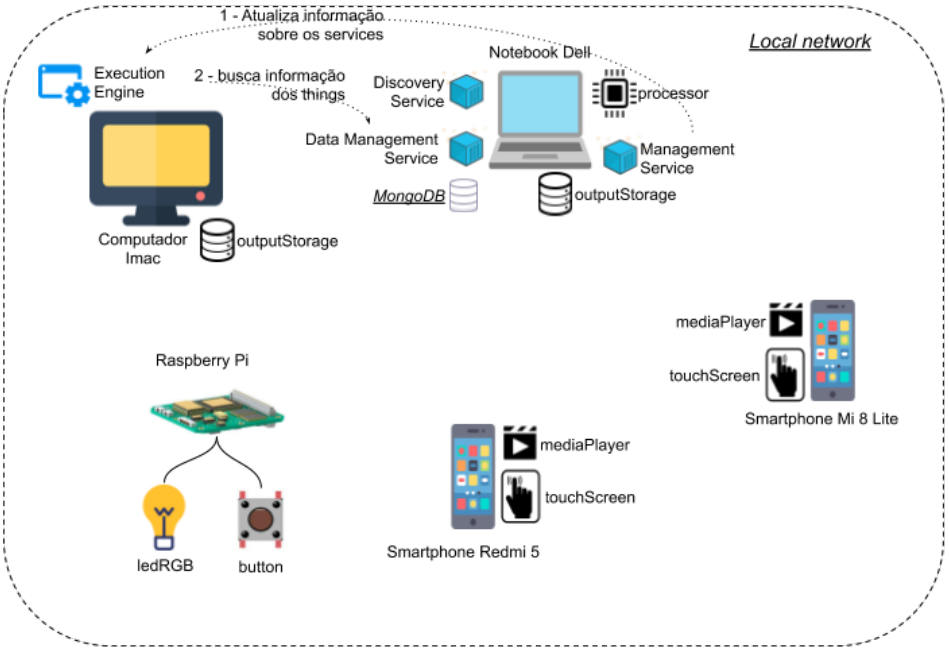

Figura 15 – Caso de uso para ambiente não preparado - Fase 4

Vale destacar que os *things* se anunciam de maneira similar ao caso anterior, porém eles não são referenciados (elemento *"addr"*) individualmente no documento da aplicação FlyIoTL. Por isso, quando um determinado evento é registrado no *Data Management Service*, o registro de um evento é disparado para todos os *things* daquele mesmo *type*. Quando um desses eventos ocorrer, um gatilho de ação referente a tal evento pode ser acionado no respectivo *workflow* da aplicação. Este comportamento permite criar grupos de *things* referenciados, baseando-se em seu tipo, para um propósito mais genérico. Algumas das informações sobre os metadados de descoberta expostos ao *Discovery Service* pelos *things* podem ser observadas abaixo.

```
{
    "id": "932cc81e-c5f8-11e9-a20c-685b357d8a1a",
    "type": "processor",
    "entrypoint": "http://10.0.1.3:8000/",
    "timestamp": "2019-08-23 19:51:48.075966",
    "cores": "4",
    "language": ["Python3", "Python2"],
    "support": "Docker-container"
}
```

```
{
    "id": "965080da-c5f8-11e9-92a4-685b357d8a1a",
    "type": "outputStorage",
    "entrypoint": "http://10.0.1.3:9000/",
    "timestamp": "2019-08-23 19:51:53.185732",
    "events": ["isCreated", "isModified", "isDeleted"]
}
```

```
{
    "entrypoint": "http://10.0.1.26:10500/",
    "id": "52e0f19a-587e-4968-970e-56711f811f00",
    "type": "mediaPlayer",
    "model": "Xiaomi_Redmi 5",
    "timestamp": "2019-08-23 20:02:28.287902",
    "codecs": [ "h264", "aac", "mp3", "mp4"],
    "formats": ["ts"],
    "name": "Xiaomi_Redmi 5",
    "observables": ["current"],
    "protocols": ["dash", "rtsp"]
}
```

```
{
    "entrypoint": "http://10.0.1.26:11000/",
    "id": "a1b60b76-69f2-419b-aa44-d760677e736c",
    "type": "touchSurface",
    "model": "Xiaomi_Redmi 5",
    "timestamp": "2019-08-23 20:02:34.335185",
    "addr": "04:B1:67:36:1E:CC",
    "multitouch": true,
    "name": "Xiaomi_Redmi 5",
    "observables": ["swipeUp", "swipeDown", "swipeLeft","swipeRight"],
    "uuid": "4f23e577-c511-4f3c-a346-dd3081fbef8a"
}
```

Na próxima etapa desse caso de uso, os eventos são registrados e encaminhados da mesma maneira que no caso de uso anterior. Quando algum evento registrado em algum dos grupos de *things* for detectado e todas as condições de um *workflow* for satisfeita, o *Presentation Controller* da *Execution Engine*, por meio do *Action Invoker*, envia as ações para o *Data Management Service* que por sua vez encaminha para os respectivos grupos de *things* de acordo com seu *type*. A lista de eventos registrados pode ser conferida no Anexo C.1

O *Management Service* pode ser configurado também para monitorar, por meio do *System Orchestration*, o número máximo de instâncias de *Services* a serem executadas em um mesmo *Processor*, a fim de garantir um certo balanceamento dos *Resources* na rede. Quando tal limite é excedido, o *Management Service* irá localizar outros *Processors* com a(s) mesma(s) característica(s) para executar um *Service* FlyIoT. Quando isso é possível, e tal *Resource* está disponível, o *Service* em questão é instanciado no segundo *Processor*. Caso seja um *Data Management Service*, as informações dos dados referentes aos *things* e eventos são transferidas para o novo *Data Management Service* enquanto as informações referentes ao *Data Management Service* antigo são atualizadas nos demais *Services* e na *Execution Engine*. Por fim, o antigo *Service* é encerrado a partir de uma mensagem enviada pelo *Management Service*. Se o *Service* a ser migrado não for um *Data Management Service*, a transferência pode ser imediata – mantendo-se o procedimento de primeiro inicializar, atualizar e depois encerrar – pois, com exceção do *Data Management Service*, os demais não possuem dados a serem restaurados.

No caso de uso atual, essa migração para outro *Processor* não gerou impacto na execução da aplicação FlyIoTL, mesmo quando era necessário realizar o download e construção completa da imagem referente ao *Service* no segundo *Processor*. Isso porque o processo ocorre em paralelo ao componente de FlyIoT que já está em execução. Porém, quando um *Service* é encerrado abruptamente em razão de fatores externos, pode haver impacto e perda de informação referente a algum evento nesse intervalo de tempo entre o *Management Service* detectar a falha e inicializar o novo *Service*. Quando essa inicialização ocorre no mesmo *Processor* em que o anterior foi encerrado, o tempo de processo pode ser bem reduzido, pois a imagem do *container* já existe ali, bastando apenas inicializar um novo *container*. Como a quantidade de *things* presentes na aplicação FlyIoTL apresentada nos casos de uso é pequena, essa abordagem de migração de *Services* FlyIoT abre espaço para novos estudos com foco específico nesse tipo de análise.

O documento da aplicação FlyIoTL deste caso de uso é similar ao anterior. Um destaque a ser observado aqui seria a definição de propriedades dentro de *requirements* no corpo do documento. Por exemplo, no caso do trecho de código a seguir, o *thing* em questão deve possuir alguns eventos de gestos que serão associados a ele posteriormente nos *worlflows* do documento FlyIoTL. Demais características podem ser especificadas no

documento para que *things* possam executar melhor suas funções distintas. Com base nessas informações, o *middleware* baseado em FlyIoT tentará endereçar as ações e eventos específicos aos respectivos *things*.

```
- thing: myTouchSurface
  type: touchSurface
  requirements:
    gestures: ['swipeUp', 'swipeDown', 'swipeLeft', 'swipeRight']
    multitouch: true
```

```
- thing: MyFile
  type: outputStorage
  filepath: 'app:/log.txt'
  requirements:
    events: ['isCreated', 'isModified', 'isDeleted']
```

Esse cenário para ambiente não preparado ainda é pouco explorado na literatura por *middlewares* de modo geral. Dentre os *middlewares* estudados nesse trabalho, talvez o que mais se aproxime desse tipo de cenário seja o M-Hub de (TALAVERA et al., 2015). Para adequá-lo ao cenário, como o caso de uso proposto nessa seção, seria necessário que o desenvolvedor adaptasse cada um dos sensores e atuadores que o M-Hub agrega. Este *middleware* já traz certa facilidade no processo de descoberta e gerenciamento dos dados. Porém, como cada componente M-Hub é representado pro smartphones/tablets, seria necessário que o desenvolvedor implementasse um meio de comunicação com esses dispositivos para que possa acessar as informações dos sensores/atuadores agregados a eles por meio de sua aplicação. Só então o desenvolvedor teria acesso a informações sobre as capacidades do mesmo. Para soluções de problemas que não envolvam principalmente sensoriamento, adaptar o M-Hub poderia ser custoso para o desenvolvedor de uma aplicação IoT, pois ele teria que alterar o direcionamento dado pelo autor da proposta de *middleware* M-Hub.

Por outro lado, do ponto de vista da programação, para adaptar linguagens já existentes, como a proposta por (GOMES et al., 2017), seria necessário recorrer a outras linguagens mais flexíveis como, por exemplo, a linguagem Lua (IERUSALIMSCHY; DE FIGUEIREDO; FILHO, 1996). Linguagens procedurais para representar e lidar com eventos e ações descritos nesse trabalho gerariam um grande esforço por parte do desenvolvedor, que, para cada aplicação ou pequena alteração de comportamento, teria um retrabalho muito elevado. Além disso, usar diversas linguagens como "cola" entre as demais torna ainda mais árdua a integração desses dispositivos IoT.

Com a utilização da linguagem FlyIoTL e o *middleware* baseado em FlyIoT, foi possível observar o comportamento em um ambiente inicialmente desconhecido. Uma

primeira abordagem de orquestração, descoberta e gerenciamento de serviços e recursos pôde ser observada conforme os requisitos destacados no Capítulo 2. FlyIoTL foi especificada para que seja uma linguagem de construção de aplicações IoT adaptáveis aos *things*, capaz de permitir que novos elementos da linguagem sejam referenciados e relacionados a eventos baseados nas distinções das capacidades de cada recurso (*thing*). Por fim, o fato de o *middleware* baseado em FlyIoT se beneficiar da própria descoberta de recursos para instanciar os seus componentes preenche uma lacuna importante no quesito de gerenciamento de código, o que faz da arquitetura FlyIoT uma abordagem promissora na concepção de aplicações IoT sob o conceito de máquina única.

8 CONCLUSÃO

Neste trabalho, foi descrita uma arquitetura de *middleware* distribuída orientada a serviços, denominada FlyIoT, que visa abstrair boa parte da complexidade de um ambiente IoT e prover recursos a aplicações em um nível mais alto em uma visão que reúne recursos IoT como uma máquina a ser gerenciada. Para isso, a proposta usa conceitos e abordagens modulares para generalização de componentes a fim de homogeneizar a fragmentação presente em ambientes IoT. Também foi descrito um modelo e instanciação inicial para uma linguagem voltada ao desenvolvimento de aplicações IoT, denominada FlyIoTL. Foram apresentados exemplos de utilização da linguagem para manipulação de diversos tipos de "coisas" em ambientes IoT, explorando as facilidades trazidas pelo *middleware* baseado na arquitetura FlyIoT. Foram abordados também alguns conceitos para instanciação dos componentes do *middleware* baseado em princípios REST e com uso de estruturas de comunicação simples de troca de mensagens JSON para provisão de recursos a uma aplicação.

Foi demonstrado um exemplo de como o *middleware* FlyIoT tem um princípio "regenerativo" ao ser afetado por alguma falha em um de seus componentes de serviço e como esse processo de recuperação se dá através da inicialização automática de *containers* pré-configurados e disponíveis publicamente em serviços de nuvem. Isso foi possível graças à abordagem de que um recurso do tipo processador se encontrar apto e disponível para tal tarefa.

A linguagem apresentada ainda não abrange toda a especificidade e heterogeneidade de *things* disponíveis em ambientes IoT, porém toma como base uma harmonização a fim de minimizar a complexidade de manipulação dos mesmos, reduzindo-os apenas a três categorias básicas: *Inputs*, *Outputs* e *Processors*. A linguagem também suporta características de definição baseadas tanto em ambientes preparados, nos quais cada dispositivo pode ser identificado a partir de um endereçamento único. Para esses dispositivos, é realizada uma validação de acordo com os requisitos especificados pelo desenvolvedor da aplicação IoT para cada *thing* de interesse. Com o *middleware* baseado em FlyIoT servindo de apoio à linguagem FlyIoTL, permitiu-se a construção de uma aplicação IoT, partindo do princípio de que o IoT é uma máquina única. Isso privou o desenvolvedor de se preocupar com questões de mais baixo nível. A máquina de execução da aplicação FlyIoTL se tornou mais simples, pois boa parte da carga de trabalho foi distribuída entre os componentes especificados na arquitetura *FlyIoT*.

Espera-se, com esse trabalho, que o processo de construção de aplicações IoT seja mais simples, já que o nível desenvolvimento se torna mais elevado, eliminando a obrigação de lidar com complexidades relacionadas ao meio sem limitar a capacidade dessas aplicações IoT. Ao mesmo tempo, aplicações genuinamente IoT são possíveis graças

à distribuição inerente à arquitetura FlyIoT. Tal arquitetura especifica a instanciação e integração dos próprios serviços a partir de suas características de descoberta de recursos destinados a tais tarefas, aliando a isso as facilidades da linguagem declarativa FlyIoTL proposta.

8.1 Trabalhos futuros

Diversos cenários de estudo podem se desdobrar a partir da solução apresentada nesse projeto. Partindo daqui, espera-se como trabalhos futuros:

Aprimorar o *middleware* baseado em FlyIoT por meio de integrações com outros serviços com foco em análise de contexto, conferindo à FlyIoT maior adaptabilidade e inteligência. Novos componentes de serviço poderiam ser acoplados, por meio de *Context Reasoning Services*, de tal forma que o *middleware* identifique possíveis recursos com informações incompletas ou até mesmo similaridades entre esses recursos, permitindo que a aplicação IoT se adapte automaticamente sem prejuízo à execução.

Integração com serviços externos para complementar as funcionalidades do próprio *middleware* baseado em FlyIoT. Tais integrações podem servir de apoio para o uso de Inteligência Artificial ou ambientes de *Machine Learning* com o propósito de realizar predições sobre um determinado ambiente IoT onde o *middlware* for instanciado.

Realizar análises quantitativas do *middleware* baseado em FlyIoT frente a outras propostas de *middleware* na literatura. Fazer comparativos com o propósito de testar e aprimorar sua escalabilidade e políticas de distribuição frente a essas outras propostas. Tais políticas, se bem especificadas, podem trazer maior otimização quando o *middleware* instanciar seus componentes nos *Resources* mais adequados de acordo com as necessidades das aplicações IoT.

Realizar análises de desempenho no *middleware* baseado em FlyIoT. Trocas de mensagens na rede e latência na execução de uma mesma aplicação IoT devem ser analisadas para identificar possíveis pontos deficientes no desempenho da proposta, aprimorando sua evolução como um *middleware*.

Aprimoramentos da linguagem FlyIoTL podem ser pesquisados, de modo a oferecer maior abrangência na especificação de grupos de *things* para execução de tarefas. Além disso, testar a implantação de uma aplicação FlyIoTL em outros ambientes ou sobre outro *middleware* a fim de analisar possíveis dificuldades.

Apresentar a proposta para desenvolvedores não especialistas em IoT. Identificar o quão simples é a linguagem FlyIoTL para esse público na comparação com outras soluções IoT existentes e elencar possíveis melhorias ou adaptações.

A partir da concepção de uma versão estável e sólida de FlyIoTL, estudar a possibilidade de construção de ferramentas gráficas visuais de autoria para apoio a não desenvolvedores (sem experiência em programação), permitindo que experimentem a integração de aplicações IoT em suas respectivas áreas de estudo.

Referências

ADESTO, Echelon by. **Echelon Platform**. [S.l.: s.n.]. https://www.echelon.com/. Accessed: 2018-11-29.

ALVI, Sheeraz A et al. Internet of multimedia things: Vision and challenges. **Ad Hoc Networks**, Elsevier, v. 33, p. 87–111, 2015.

AMJAD, Muhammad et al. TinyOS-new trends, comparative views, and supported sensing applications: A review. **IEEE Sensors Journal**, IEEE, v. 16, n. 9, p. 2865–2889, 2016.

ARASTEH, H et al. Iot-based smart cities: a survey. In: IEEE. 2016 IEEE 16th International Conference on Environment and Electrical Engineering (EEEIC). [S.l.: s.n.], 2016. p. 1–6.

BATISTA, CECF. GINGA-MD: Uma Plataforma para Suporte à execução de Aplicações Hipermídia Multi-Dispositivo Baseada em NCL. **Phd Theses. Pontifícia Universidade Católica do Rio de Janeiro**, 2013.

BEN-KIKI, Oren; EVANS, Clark; INGERSON, Brian. Yaml ain't markup language (yaml™) version 1.1. **yaml. org, Tech. Rep**, p. 23, 2005.

BERGSTRA, J. A.; MIDDELBURG, C. A. **ITU-T Recommendation G.107 : The E-Model, a computational model for use in transmission planning**. [S.l.], 2003.

BERNSTEIN, David. Containers and cloud: From lxc to docker to kubernetes. **IEEE Cloud Computing**, IEEE, v. 1, n. 3, p. 81–84, 2014.

BOHN, Hendrik; BOBEK, Andreas; GOLATOWSKI, Frank. SIRENA-service infrastructure for real-time embedded networked devices: A service oriented framework for different domains. In: ICN/ICONS/MCL. [S.l.: s.n.], 2006. p. 43.

BONOMI, Flavio et al. Fog Computing and Its Role in the Internet of Things. In: PROCEEDINGS of the First Edition of the MCC Workshop on Mobile Cloud Computing. Helsinki, Finland: ACM, 2012. (MCC '12), p. 13–16. ISBN 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513. Disponível em: <http://doi.acm.org/10.1145/2342509.2342513>.

BRAY, Tim et al. (Ed.). **Extensible Markup Language (XML) 1.1 (Second Edition)**. Second. [S.l.], jul. 2006. Disponível em: <http://www.w3.org/TR/2006/REC-xml11-20060816/>.

BULTERMAN, Dick CA; RUTLEDGE, Lloyd W. **SMIL 3.0: Flexible Multimedia for Web, Mobile Devices and Daisy Talking Books**. [S.l.]: Springer Publishing Company, Incorporated, 2008.

CAPORUSCIO, M.; RAVERDY, P.; ISSARNY, V. ubiSOAP: A Service-Oriented Middleware for Ubiquitous Networking. **IEEE Transactions on Services Computing**, v. 5, n. 1, p. 86–98, jan. 2012. ISSN 1939-1374. DOI: 10.1109/TSC.2010.60.

CHODOROW, Kristina. **MongoDB: the definitive guide: powerful and scalable data storage**. [S.l.]: "O'Reilly Media, Inc.", 2013.

CRUZ HUACARPUMA, Ruben et al. Distributed data service for data management in internet of things middleware. **Sensors**, Multidisciplinary Digital Publishing Institute, v. 17, n. 5, p. 977, 2017.

DAVID, Lincoln et al. A DDS-based middleware for scalable tracking, communication and collaboration of mobile nodes. **Journal of Internet Services and Applications**, v. 4, n. 1, p. 16, 2013.

DE SOUZA, Luciana Moreira Sá et al. Socrades: A web service based shop floor integration infrastructure. In: THE internet of things. [S.l.]: Springer, 2008. p. 50–67.

DHAS, Y Justin; JEYANTHI, P. A Review on Internet of Things Protocol and Service Oriented Middleware. In: IEEE. 2019 International Conference on Communication and Signal Processing (ICCSP). [S.l.: s.n.], 2019. p. 0104–0108.

EISENHAUER, Markus; ROSENGREN, Peter; ANTOLIN, Pablo. Hydra: A development platform for integrating wireless devices and sensors into ambient intelligence systems. In: THE Internet of Things. [S.l.]: Springer, 2010. p. 367–373.

FIELDING, Roy et al. **RFC 2616: Hypertext transfer protocol–HTTP/1.1**. [S.l.]: June, 1999.

FOUNDATION, Research Eclipse. **Eclipse Ditto**. Accessed: 2019-05-13. 2019.

_____. **Eclipse HONO**. Accessed: 2019-05-13. 2019.

FREEMAN-BENSON, Bjorn Nathan. Constraint imperative programming. University of Washington, 1992.

FREESZ JR., Marco A.; YUNG, Ludmila; MORENO, Marcelo. STorM: A Hypermedia Authoring Model for Interactive Digital Out-of-Home Media. In: PROCEEDINGS of the 23rd Brazillian Symposium on Multimedia and the Web. Gramado, RS, Brazil: ACM, 2017. (WebMedia '17), p. 41–48. ISBN 978-1-4503-5096-9. DOI: 10.1145/3126858.3126889. Disponível em: <http://doi.acm.org/10.1145/3126858.3126889>.

GAY, Warren. **Beginning STM32: Developing with FreeRTOS, libopencm3 and GCC**. [S.l.]: Apress, 2018.

GIL, David et al. Internet of things: A review of surveys based on context aware intelligent services. **Sensors**, Multidisciplinary Digital Publishing Institute, v. 16, n. 7, p. 1069, 2016.

GOMES, Tiago et al. A modeling domain-specific language for IoT-enabled operating systems. In: IEEE. IECON 2017-43rd Annual Conference of the IEEE Industrial Electronics Society. [S.l.: s.n.], 2017. p. 3945–3950.

HIGHTOWER, Kelsey; BURNS, Brendan; BEDA, Joe. **Kubernetes: Up and Running: Dive Into the Future of Infrastructure**. [S.l.]: "O'Reilly Media, Inc.", 2017.

IERUSALIMSCHY, Roberto; DE FIGUEIREDO, Luiz Henrique; FILHO, Waldemar Celes. Lua—an extensible extension language. **Software: Practice and Experience**, Wiley Online Library, v. 26, n. 6, p. 635–652, 1996.

JONES, Michael; BRADLEY, John; SAKIMURA, Nat. **Json web token (jwt)**. [S.l.], 2015.

KIM, Jaeho; LEE, Jang-Won. OpenIoT: An open service framework for the Internet of Things. In: IEEE. 2014 IEEE world forum on internet of things (WF-IoT). [S.l.: s.n.], 2014. p. 89–93.

_____._____. In: IEEE. 2014 IEEE world forum on internet of things (WF-IoT). [S.l.: s.n.], 2014. p. 89–93.

KIM, Marie; LEE, Jun Wook et al. Cosmos: A middleware for integrated data processing over heterogeneous sensor networks. **ETRI journal**, Wiley Online Library, v. 30, n. 5, p. 696–706, 2008.

KRYLOVSKIY, Alexandr; JAHN, Marco; PATTI, Edoardo. Designing a smart city internet of things platform with microservice architecture. In: IEEE. 2015 3rd International Conference on Future Internet of Things and Cloud. [S.l.: s.n.], 2015. p. 25–30.

LEACH, Paul J; MEALLING, Michael; SALZ, Rich. A universally unique identifier (uuid) urn namespace, 2005.

MAARALA, Altti Ilari; SU, Xiang; RIEKKI, Jukka. Semantic reasoning for context-aware Internet of Things applications. **IEEE Internet of Things Journal**, IEEE, v. 4, n. 2, p. 461–473, 2016.

MESOS, Apache. **Mesosphere Platform**. [S.l.: s.n.]. `https://mesosphere.com`. Accessed: 2018-12-23.

MIRANDA, Jorge Miguel Pereira Coutada. Sensing technologies in pervasive healthcare: Evaluation and design for senior citizens and continuous care, 2019.

MUSADDIQ, Arslan et al. A survey on resource management in IoT operating systems. **IEEE Access**, IEEE, v. 6, p. 8459–8482, 2018.

NAIK, Nitin. Building a virtual system of systems using Docker Swarm in multiple clouds. In: IEEE. 2016 IEEE International Symposium on Systems Engineering (ISSE). [S.l.: s.n.], 2016. p. 1–3.

NEDOS, A. et al. Probabilistic Discovery of Semantically Diverse Content in MANETs. **IEEE Transactions on Mobile Computing**, v. 8, n. 4, p. 544–557, abr. 2009. ISSN 1536-1233. DOI: 10.1109/TMC.2008.133.

NEGASH, Behailu et al. DoS-IL: A domain specific Internet of Things language for resource constrained devices. **Procedia Computer Science**, Elsevier, v. 109, p. 416–423, 2017.

OSKARSSON, Agneta et al. Total and inorganic mercury in breast milk and blood in relation to fish consumption and amalgam fillings in lactating women. **Archives of Environmental Health: An International Journal**, Taylor & Francis, v. 51, n. 3, p. 234–241, 1996.

PENG, Limei; DHAINI, Ahmad R; HO, Pin-Han. Toward integrated Cloud–Fog networks for efficient IoT provisioning: Key challenges and solutions. **Future Generation Computer Systems**, Elsevier, v. 88, p. 606–613, 2018.

PERERA, Charith et al. Mosden: An internet of things middleware for resource constrained mobile devices. In: IEEE. 2014 47th Hawaii International Conference on System Sciences. [S.l.: s.n.], 2014. p. 1053–1062.

PRAZERES, Cássio; SERRANO, Martin. Soft-iot: Self-organizing fog of things. In: IEEE. 2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA). [S.l.: s.n.], 2016. p. 803–808.

QIN, Weijun et al. RestThing: A Restful Web service infrastructure for mash-up physical and Web resources. In: IEEE. 2011 IFIP 9th International Conference on Embedded and Ubiquitous Computing. [S.l.: s.n.], 2011. p. 197–204.

RAZZAQUE, M. A. et al. Middleware for Internet of Things: A Survey. **IEEE Internet of Things Journal**, v. 3, n. 1, p. 70–95, fev. 2016. ISSN 2327-4662. DOI: 10.1109/JIOT.2015.2498900.

RICHARDSON, Leonard; AMUNDSEN, Mike; RUBY, Sam. **RESTful Web APIs: Services for a Changing World**. [S.l.]: "O'Reilly Media, Inc.", 2013.

SALEME, Estêvão B. et al. Mulsemedia DIY: A Survey of Devices and a Tutorial for Building Your Own Mulsemedia Environment. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 52, n. 3, 58:1–58:29, jun. 2019. ISSN 0360-0300. DOI: 10.1145/3319853. Disponível em: <http://doi.acm.org/10.1145/3319853>.

SAMUEL, Stephen; BOCUTIU, Stefan. **Programming Kotlin**. [S.l.]: Packt Publishing Ltd, 2017.

SANTANA, Cleber Jorge Lira de; MELLO ALENCAR, Brenno de; PRAZERES, Cássio V. Serafim. Reactive Microservices for the Internet of Things: A Case Study in Fog Computing. In: PROCEEDINGS of the 34th ACM/SIGAPP Symposium on Applied Computing. Limassol, Cyprus: ACM, 2019. (SAC '19), p. 1243–1251. ISBN 978-1-4503-5933-7. DOI: 10.1145/3297280.3297402. Disponível em: <http://doi.acm.org/10.1145/3297280.3297402>.

SILVA, Bhagya Nathali; KHAN, Murad; HAN, Kijun. Internet of things: A comprehensive review of enabling technologies, architecture, and challenges. **IETE Technical review**, Taylor & Francis, v. 35, n. 2, p. 205–220, 2018.

SILVA, José R; DELICATO, Flávia C et al. PRISMA: A publish-subscribe and resource-oriented middleware for wireless sensor networks. In: CITESEER. PROCEEDINGS of the Tenth Advanced International Conference on Telecommunications, Paris, France. [S.l.: s.n.], 2014. v. 2024, p. 8797.

SINHA, Nitin; PUJITHA, Korrapati Eswari; ALEX, John Sahaya Rani. Xively based sensing and monitoring system for IoT. In: IEEE. 2015 International Conference on Computer Communication and Informatics (ICCCI). [S.l.: s.n.], 2015. p. 1–6.

SOARES, Luiz Fernando Gomes; RODRIGUES, Rogério Ferreira. Nested context language 3.0 part 8–ncl digital tv profiles. **Monografias em Ciência da Computação do Departamento de Informática da PUC-Rio**, v. 1200, n. 35, p. 06, 2006.

SOARES, Luiz Fernando Gomes; RODRIGUES, Rogério Ferreira; MORENO, Márcio Ferreira. Ginga-NCL: the declarative environment of the Brazilian digital TV system. **Journal of the Brazilian Computer Society**, Springer, v. 13, n. 1, p. 37–46, 2007.

SOLUTIONS, Cisco Fog Computing. Unleash the power of the Internet of Things. **Cisco Systems Inc**, 2015.

SUN, Long; LI, Yan; MEMON, Raheel Ahmed. An open IoT framework based on microservices architecture. **China Communications**, IEEE, v. 14, n. 2, p. 154–162, 2017.

TALAVERA, Luis E et al. The mobile hub concept: Enabling applications for the internet of mobile things. In: IEEE. 2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops). [S.l.: s.n.], 2015. p. 123–128.

TEIXEIRA, Thiago et al. Service Oriented Middleware for the Internet of Things: A Perspective. In: PROCEEDINGS of the 4th European Conference on Towards a Service-based Internet. Poznan, Poland: Springer-Verlag, 2011. (ServiceWave'11), p. 220–229. ISBN 978-3-642-24754-5. Disponível em: <http://dl.acm.org/citation.cfm?id=2050869.2050893>.

THÖNES, Johannes. Microservices. **IEEE software**, IEEE, v. 32, n. 1, p. 116–116, 2015.

TOSATTO, Andrea; RUIU, Pietro; ATTANASIO, Antonio. Container-based orchestration in cloud: state of the art and challenges. In: IEEE. 2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems. [S.l.: s.n.], 2015. p. 70–75.

VAN ROSSUM, Guido; DRAKE JR, Fred L. The Python language reference. **Python software foundation**, 2014.

VEIGA, Ernesto Fonseca et al. An Ontology-based Representation Service of Context Information for the Internet of Things. In: ACM. PROCEEDINGS of the 23rd Brazillian Symposium on Multimedia and the Web. [S.l.: s.n.], 2017. p. 301–308.

WOLLSCHLAEGER, Martin; SAUTER, Thilo; JASPERNEITE, Juergen. The future of industrial communication: Automation networks in the era of the internet of things and industry 4.0. **IEEE industrial electronics magazine**, IEEE, v. 11, n. 1, p. 17–27, 2017.

XIVELY Platform by Google Cloud IoT. [S.l.: s.n.]. `https://xively.com`. Accessed: 2018-12-27.

XU, Li Da; XU, Eric L; LI, Ling. Industry 4.0: state of the art and future trends. **International Journal of Production Research**, Taylor & Francis, v. 56, n. 8, p. 2941–2962, 2018.

YAQOOB, Ibrar et al. Internet of things architecture: Recent advances, taxonomy, requirements, and open challenges. **IEEE wireless communications**, IEEE, v. 24, n. 3, p. 10–16, 2017.

ZIKRIA, Yousaf Bin et al. A survey on routing protocols supported by the Contiki Internet of things operating system. **Future Generation Computer Systems**, Elsevier, v. 82, p. 200–219, 2018.

ANEXO A – Documento de exemplo da aplicação FlyIoTL

```
FlyIoTL:
    things:
      - thing: myThermometer
        type: thermometer
        params: [ 'temperature' ]

      - thing: myUmiditySensor
        type: umiditySensor

      - thing: eastFan
        type: fan
        addr: 55AC567

      - thing: myLed
        type: ledRGB
        params: [ 'color', 'intensity?' ]

      - thing: mySmartphone
        type: touchSurface
        requirements:
            gestures: [ 'swipeUp', 'swipeDown' ]
            multitouch: true

      - thing: myPlayer
        type: mediaPlayer
        params: [ 'mediaurl', 'duration' ]
        requirements:
            codecs: [ 'h264', 'h265?', 'aac', 'mp3?' ]
            formats: [ 'isobmff', 'ts?' ]
            protocols: [ 'dash' ]

      - thing: myFile
        type: inputStorage
        filepath: 'myApp:/myFile'
        requirements:
            mustExist: true

      - thing: myFile2
        type: outputStorage
        filepath: 'myApp:/newFile'
        requirements:
            events: [ 'isCreated', 'isModified', 'isDeleted' ]
        params: [ 'data' ]

      - thing: myMonitoring
```

```
46          type: processor
47          lang: 'python3'
48          sourcepath: './path/to/file.py'
49      #     code: |
50      #       import time
51      #       ts = time.time()
52      #       print(ts)
53          params: [ 'temperature' ]
54
55      workflows:
56
57        - workflow: w1
58          when:
59            - myThermometer: '[33-40]'
60            - myUmiditySensor: '<50%'
61          do:
62            - myThermometer: read
63              reference: temp1
64            - myMonitoring: start
65              temperature: temp1
66            - eastFan: start
67
68        - workflow: w2
69          when:
70            - mySmartphone: 'swipeUp'
71          do:
72            - myLed: start
73              color: '#ffffff'
74              intensity: '0.8'
75
76        - workflow: w3
77          when:
78            - mySmartphone: 'swipeDown'
79          do:
80            - myLed: start
81              color: '#fffde7'
82              intensity: '0.3'
83
84        - workflow: w4
85          when:
86            - w2: stops
87            - w3: stops
88          do:
89            - myMonitoring: start
90            - myFile2: start
91
92        - workflow: w5
```

```
93          when:
94            - myFile2: isCreated
95          do:
96            - myFile: read
97              reference: nf
98            - myFile2: write
99              data: nf
100           - myThermometer: read
101             reference: temp5
102           - myMonitoring: start
103             temperature: temp5
104
105       - workflow: w6
106         when:
107           - w1: starts
108         do:
109           - myPlayer: start
110             mediaurl: 'file://myApp/path/to/file'
111             duration: '50s'
112
113       - workflow: w7
114         when:
115           - myPlayer: '=10s'
116         do:
117           - eastFan: start
118           - myLed: stop
```

ANEXO B – Aplicação FlyIoTL - Caso de uso em ambiente preparado

```
FlyIoTL:

  things:

    - thing: myPlayer
      type: mediaPlayer
      params: ['mediaurl']
      addr: '10-FB-75-65-9A-8B'
      requirements:
        codecs: ['h264', 'aac', 'mp3?']
        formats: ['ts']
        protocols: ['dash', 'rtsp']

    - thing: myTouchSurface
      type: touchSurface
      addr: 'E7-39-AB-C5-E6-F7'
      requirements:
        gestures: [ 'swipeUp', 'swipeDown', 'swipeLeft', 'swipeRight' ]
        multitouch: true

    - thing: myLed
      type: ledRGB
      params: ['color', 'intensity']
      addr: 'E4-DF-60-B9-02-1F'

    - thing: myButton
      type: button
      addr: '51-DA-94-58-D1-1E'

    - thing: myFile
      type: outputStorage
      filepath: 'app:/log.txt'
      addr: '20-23-59-3E-51-C6'
      requirements:
        events: [ 'isCreated', 'isModified', 'isDeleted' ]
      params: [ 'data' ]

    - thing: myScript
      type: processor
      addr: '68-1C-DC-03-38-30'
      lang: 'Python3'
      sourcepath: './scripts/start.py'

  workflows:

```

```
46    - workflow: w1
47      when:
48        - myButton: starts
49      do:
50        - myPlayer: start
51          mediaurl: 'rtsp://10.10.10.207:5544/'
52        - myLed: start
53          color: '#ff0000'
54
55    - workflow: w2
56      when:
57        - myButton: starts
58        - w1: starts
59      do:
60        - myPlayer: stop
61        - myLed: stop
62
63    - workflow: w3
64      when:
65        - w1: starts
66        - myTouchSurface: swipeUp
67      do:
68        - myLed: 'start'
69          intensity: '1.0'
70        - myScript: start
71
72    - workflow: w4
73      when:
74        - w1: starts
75        - myTouchSurface: swipeDown
76      do:
77        - myLed: 'start'
78          intensity: '0.2'
79        - myScript: start
80
81    - workflow: w5
82      when:
83        - myPlayer: pauses
84      do:
85        - myLed: 'start'
86          color: '#ffff00'
87        - myPlayer: read
88          reference: currenttime
89        - myFile: 'write'
90          data: 'currenttime'
91
92    - workflow: w6
```

```
      when:
        - w2: stops
      do:
        - myLed: stop
        - myTouchSurface: stop
        - myScript: start
```

B.1 Eventos registrados no caso de uso em ambiente preparado


```
[
    {
        "_id": "5d6068c4ddc27b052e938097",
        "id": "75917abe-c5f5-11e9-84c3-685b357d8a1a_w1_when_0_myButton",
        "condition": "starts",
        "thing": "button",
        "addr": "51-DA-94-58-D1-1E"
    },
    {
        "_id": "5d6068c4ddc27b052e938098",
        "id": "75917c4e-c5f5-11e9-84c3-685b357d8a1a_w2_when_0_myButton",
        "condition": "starts",
        "thing": "button",
        "addr": "51-DA-94-58-D1-1E"
    },
    {
        "_id": "5d6068c4ddc27b052e93809b",
        "id": "75917d98-c5f5-11e9-84c3-685b357d8a1a_w3_when_1_myTouchSurface",
        "condition": "swipeUp",
        "thing": "touchSurface",
        "addr": "E7-39-AB-C5-E6-F7"
    },
    {
        "_id": "5d6068c4ddc27b052e93809d",
        "id": "75917e60-c5f5-11e9-84c3-685b357d8a1a_w4_when_1_myTouchSurface",
        "condition": "swipeDown",
        "thing": "touchSurface",
        "addr": "E7-39-AB-C5-E6-F7"
    },
    {
        "_id": "5d6068c4ddc27b052e93809e",
        "id": "75917eba-c5f5-11e9-84c3-685b357d8a1a_w5_when_0_myPlayer",
        "condition": "pauses",
        "thing": "mediaPlayer",
        "addr": "10-FB-75-65-9A-8B"
    }
]
```

ANEXO C – Aplicação FlyIoTL - Caso de uso em ambiente não preparado

```
FlyIoTL:

  things:

    - thing: myPlayer
      type: mediaPlayer
      params: ['mediaurl']
      requirements:
        codecs: ['h264', 'aac', 'mp3?']
        formats: ['ts']
        protocols: ['dash', 'rtsp']

    - thing: myTouchSurface
      type: touchSurface
      requirements:
        gestures: [ 'swipeUp', 'swipeDown', 'swipeLeft', 'swipeRight' ]
        multitouch: true

    - thing: myLed
      type: ledRGB
      params: ['color', 'intensity']

    - thing: myButton
      type: button

    - thing: myFile
      type: outputStorage
      filepath: 'app:/log.txt'
      requirements:
        events: [ 'isCreated', 'isModified', 'isDeleted' ]
      params: [ 'data' ]

    - thing: myScript
      type: processor
      lang: 'Python3'
      sourcepath: './scripts/start.py'

  workflows:

    - workflow: w1
      when:
        - myButton: starts
      do:
        - myPlayer: start
          mediaurl: 'rtsp://10.10.10.207:5544/'
```

```
46          - myLed: start
47            color: '#ff0000'
48
49      - workflow: w2
50        when:
51          - myButton: starts
52          - w1: starts
53        do:
54          - myPlayer: stop
55          - myLed: stop
56
57      - workflow: w3
58        when:
59          - w1: starts
60          - myTouchSurface: swipeUp
61        do:
62          - myLed: 'start'
63            intensity: '1.0'
64          - myScript: start
65
66      - workflow: w4
67        when:
68          - w1: starts
69          - myTouchSurface: swipeDown
70        do:
71          - myLed: 'start'
72            intensity: '0.2'
73          - myScript: start
74
75      - workflow: w5
76        when:
77          - myPlayer: pauses
78        do:
79          - myLed: 'start'
80            color: '#ffff00'
81          - myPlayer: read
82            reference: currenttime
83          - myFile: 'write'
84            data: 'currenttime'
85
86      - workflow: w6
87        when:
88          - w2: stops
89        do:
90          - myLed: stop
91          - myTouchSurface: stop
92          - myScript: start
```

C.1 Eventos registrados no caso de uso em ambiente não preparado


```
[
    {
        "_id": "5d607228ddc27b079889c8d5",
        "id": "0ee781a4-c5fb-11e9-8b8f-685b357d8a1a_w1_when_0_myButton",
        "condition": "starts",
        "thing": "button"
    },
    {
        "_id": "5d607228ddc27b079889c8d6",
        "id": "0ee7832a-c5fb-11e9-8b8f-685b357d8a1a_w2_when_0_myButton",
        "condition": "starts",
        "thing": "button"
    },
    {
        "_id": "5d607228ddc27b079889c8d9",
        "id": "0ee78456-c5fb-11e9-8b8f-685b357d8a1a_w3_when_1_myTouchSurface",
        "condition": "swipeUp",
        "thing": "touchSurface"
    },
    {
        "_id": "5d607228ddc27b079889c8db",
        "id": "0ee7850a-c5fb-11e9-8b8f-685b357d8a1a_w4_when_1_myTouchSurface",
        "condition": "swipeDown",
        "thing": "touchSurface"
    },
    {
        "_id": "5d607228ddc27b079889c8dc",
        "id": "0ee7856e-c5fb-11e9-8b8f-685b357d8a1a_w5_when_0_myPlayer",
        "condition": "pauses",
        "thing": "mediaPlayer"
    }
]
```