Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação de Arquitetura Baseada em IoT para Leitura de Tags RFID de Ultra Alta Frequência

Yago Luiz dos Santos

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Orientadora
Prof. Dr.a Edna Dias Canedo

Brasília
2019

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

dos Santos, Yago Luiz
di Implementação de Arquitetura Baseada em IoT para Leitura de Tags RFID de Ultra Alta Frequência / Yago Luiz dos Santos; orientador Edna Dias Canedo . -- Brasília, 2019.
77 p.

Dissertação (Mestrado - Mestrado Profissional em Computação Aplicada) -- Universidade de Brasília, 2019.

1. Internet das Coisas (IoT). 2. Identificação por Radiofrequência (RFID). 3. Arquitetura de Software. 4. Computação em Nuvem. 5. Microsserviços. I. , Edna Dias Canedo, orient. II. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação de Arquitetura Baseada em IoT para Leitura de Tags RFID de Ultra Alta Frequência

Yago Luiz dos Santos

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Prof. Dr.a Edna Dias Canedo (Orientadora)
CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
CIC/UnB

Dr. Robson de Oliveira Albuquerque
CEPESC

Prof.a Dr.a Aleteia Patrícia Favacho de Araújo
Coordenadora do Programa de Pós-graduação em Computação Aplicada

Brasília, 06 de Junho de 2019

Dedicatória

Dedico esta dissertação a minha família, que está comigo em todos os momentos.

Agradecimentos

Agradeço primeiramente a Deus, por sempre estar presente em minha vida e por me dar forças nos momentos de dificuldades. Agradeço a minha família, por toda dedicação e apoio em todos os momentos. Agradeço a minha orientadora, Prof. Dr.a Edna Dias Canedo por sua dedicação, ensinamentos e compreensão no decorrer do desenvolvimento desta dissertação. Agradeço aos colegas e professores da Universidade de Brasília pela contribuição acadêmica e profissional. Por fim, agradeço a todos que me apoiaram e estiveram do meu lado nesse período. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Internet das Coisas envolve um número crescente de dispositivos inteligentes interconectados em que sua comunicação acontece a qualquer hora e em qualquer lugar, sendo possível reduzir custos de *hardware* de arquiteturas complexas. A leitura de tags com Identificação por Radiofrequência utilizando Ultra Alta Frequência é uma atividade que pode gerar um grande volume de dados, devido ao leitor de tags dessa frequência. Este trabalho propõe uma arquitetura que implementa a leitura de tags com Identificação por Radiofrequência utilizando Ultra Alta Frequência com um leitor de tags de baixo custo de mercado em relação aos que utilizam essa frequência em uma infraestrutura com computação em nuvem e microsserviços. A utilização da computação em nuvem e microsserviços se fazem necessários devido à escalabilidade e flexibilidade para o grande volume de dados que podem ser gerados na leitura de tags com Identificação por Radiofrequência que utilizam Ultra Alta Frequência. A arquitetura proposta foi aplicada em um estudo de caso real para verificar a sua aderência e conformidade, além de mostrar-se adequada ao estudo de caso realizado. Os resultados obtidos demonstraram que a placa de leitura de tags com Identificação por Radiofrequência escolhida obteve desempenho satisfatório na leitura de tags, assim como as decisões arquiteturais propostas no trabalho. Em cenários onde a distância de leitura é um requisito fundamental, é necessário incluir uma antena externa para obter melhores resultados de leitura de tags. A utilização de computação em nuvem e microsserviços demonstraram ter um custo alto de desenvolvimento na arquitetura proposta, devido às suas complexidades e à quantidade de recursos criados para implementação da arquitetura.

Palavras-chave: Internet das Coisas (IoT), Identificação por Radiofrequência (RFID), Arquitetura de Software, Computação em Nuvem, Microsserviços.

Abstract

Internet of Things comprises an increasing number of interconnected smart devices, where communication happens anytime, anywhere, reducing hardware costs and the complexity of the architectures. Reading Radio Frequency Identification tags using Ultra High Frequency is an activity that can generate a large amount of data due to the tag reader of that frequency. This work proposes an architecture that implements the reading of Radio Frequency Identification tags using Ultra High Frequency with a low cost tag reader in relation to those that use this frequency in an infrastructure with cloud computing and microservices. The use of cloud computing and microservices is necessary because of the scalability and flexibility for the large volume of data that can be generated in the reading of Radio Frequency Identification tags using Ultra High Frequency. The proposed architecture was applied in a real case study to verify their adherence and compliance, in addition to showing up properly performed the case study. The results obtained demonstrate that the tag reading board with Radio Frequency Identification chosen obtained a satisfactory performance in the reading of tags, as well as the architectural decisions proposed in the work. In scenarios where reading distance is a fundamental requirement, it is necessary to include an external antenna for better tag reading results. The use of cloud computing and microservices have been shown to have a high development cost in the proposed architecture due to its complexities and the amount of resources created to implement the architecture.

Keywords: Internet of Things (IoT), Radio-Frequency Identification (RFID), Software Architecture, Cloud Computing, Microservices.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fontes

Lista de Abreviaturas e Siglas

API Interface de Programação de Aplicativos.

BLOB *Binary Large Object*.

DDD *Domain-Driven Design*.

EPC Código Eletrônico de Produto.

GHz Giga-hertz.

HF Alta Frequência.

IaaS Infraestrutura como um Serviço.

IoT Internet das Coisas.

JSON Notação de Objetos JavaScript.

JWT *JSON Web Tokens*.

KHz Quilo-hertz.

LF Baixa Frequência.

MHz Mega-hertz.

NoSQL *Not Only SQL*.

PaaS Plataforma como um Serviço.

REST Representação de Transferência de Estado.

RFID Identificação por Radiofrequência.

SaaS *Software* como um Serviço.

SDK Kit de Desenvolvimento de Software.

SGBD Sistema Gerenciador de Banco de Dados.

SLA Acordo de Nível de Serviço.

SO Sistema Operacional.

SQL Linguagem de Consulta Estruturada.

TCP *Transmission Control Protocol*.

UART Receptor/Transmissor Assíncrono Universal.

UHF Ultra Alta Frequência.

UnB Universidade de Brasília.

USB *Universal Serial Bus*.

XML Linguagem Extensível de Marcação Genérica.

Capítulo 1

Introdução

A Internet das Coisas (IoT) é um paradigma para a construção de sistemas computacionais distribuídos pela Internet, nos quais, em princípio, os mais diversos dispositivos, objetos e coisas estarão conectados e interagindo com aplicativos para estender diversos serviços às pessoas [9] [10]. A IoT envolve um número crescente de dispositivos inteligentes interconectados e sensores que geralmente não são intrusivos, transparentes e invisíveis. A comunicação entre esses dispositivos deverá acontecer a qualquer hora e em qualquer lugar, tornando a comunicação descentralizada e complexa [11].

Ashton [12] propôs a IoT para tratar seu uso na área de logística, utilizando Identificação por Radiofrequência (RFID) para rastreamento de itens. A RFID utiliza ondas de rádio para realizar a identificação de itens. Os itens são identificados em uma etiqueta com RFID conhecida como tag, que pode ser rastreada em tempo real. As ondas de rádio da RFID atuam em três regiões de frequência: Baixa Frequência (LF), Alta Frequência (HF), e Ultra Alta Frequência (UHF) [3], [8].

A computação em nuvem é um sistema distribuído e paralelo que consiste em uma coleção de computadores interconectados e virtualizados que são provisionados dinamicamente. As aplicações presentes na computação em nuvem são expostas como serviços sofisticados que podem ser acessados em uma rede [13].

Microsserviços são serviços pequenos e independentes que se comunicam entre si para formar aplicações que utilizam Interface de Programação de Aplicativos (API) [14], [15]. Os microsserviços foram introduzidos no campo da aplicação de API devido à sua flexibilidade, baixo acoplamento e escalabilidade, diferente das aplicações monolíticas, que se tornam muito maior em escala e com estrutura ainda mais complexa. A utilização de computação em nuvem na arquitetura de microsserviços é muito comum, devido à sua alta disponibilidade e extensibilidade, que permite provisionar microsserviços rapidamente [16].

A leitura de tags RFID utilizando a frequência UHF é uma atividade que pode gerar um grande volume de dados, em razão da utilização da frequência UHF, elevando o custo final de um projeto que envolva esse tipo de arquitetura [17], [18], [19]. A criação de uma arquitetura de IoT para leitura de tags RFID utilizando a frequência UHF com um leitor de tags RFID de baixo custo, reduz o orçamento de um projeto que envolva esse tipo de arquitetura. A computação em nuvem e microsserviços permitem uma arquitetura escalável e flexível para o grande volume de dados que podem ser gerados na leitura de tags RFID.

Diante do exposto, este trabalho propõe uma arquitetura baseada em Internet das Coisas para leitura de tags RFID de Ultra Alta Frequência, utilizando computação em nuvem e microsserviços. Será apresentado um estudo de caso aplicando a arquitetura desenvolvida em um contexto real, para verificar a sua aderência e conformidade ao proposto.

1.1 Problema

A maior parte dos projetos envolvendo leitores de tags RFID utilizando a frequência UHF possuem um preço de mercado alto e podem gerar uma grande quantidade de leituras de tags RFID, em decorrência da utilização da frequência UHF.

1.2 Justificativa

A maior parte dos leitores de tags RFID que utilizam a frequência UHF possuem um preço de mercado alto, inviabilizando o seu uso para empresas de pequeno porte e/ou usuários comuns. Logo, é necessária a criação de uma arquitetura de IoT para leitura de tags RFID utilizando a frequência UHF com um leitor de tags RFID de baixo custo em relação aos que utilizam essa frequência, a fim de reduzir o orçamento de um projeto que envolva esse tipo de arquitetura. A computação em nuvem e microsserviços permitem uma arquitetura escalável e flexível para o grande volume de dados que podem ser gerados na leitura de tags RFID em virtude da utilização da frequência UHF.

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo geral deste trabalho é desenvolver uma arquitetura de IoT que envolva *hardware* e *software* e que seja capaz de coletar, processar e armazenar dados de leitura de

tags RFID, utilizando a frequência UHF em um ambiente de computação em nuvem e microsserviços.

1.3.2 Objetivo Específico

Para atingir o objetivo geral, os seguintes objetivos específicos foram definidos:

- Identificar na literatura os trabalhos acadêmicos abordando o tema;
- Avaliar *hardwares* de leitura de tags RFID que utilizam a frequência UHF disponíveis no mercado para o desenvolvimento da arquitetura proposta;
- Implementar a comunicação do *hardware* de IoT;
- Implementar a arquitetura proposta utilizando um ambiente de computação em nuvem e microsserviços;
- Realizar o monitoramento do *hardware* de IoT;
- Realizar o monitoramento dos microsserviços implementados;
- Validar a arquitetura proposta aplicando a um estudo de caso real.

1.4 Metodologia de Pesquisa

A pesquisa é o processo formal e sistemático de desenvolvimento do método científico. O objetivo fundamental da pesquisa é descobrir respostas para problemas mediante o emprego de procedimentos científicos [20].

Podemos classificar as pesquisas de várias formas, conforme a busca de respostas para o problema encontrado [21]. A Figura 1.1 apresenta o processo de classificação da pesquisa, assim como o processo de implementação definido neste trabalho.

O processo descrito na Figura 1.1 começa com a identificação da pesquisa, sendo que a primeira tarefa consiste na identificação de sua natureza. Este trabalho faz uso da natureza de pesquisa aplicada e objetiva gerar conhecimentos para aplicação prática, que estão dirigidos à solução de problemas específicos [20], [21]. A arquitetura proposta neste trabalho faz uso dessa natureza de pesquisa.

A próxima tarefa foi a definição da abordagem da pesquisa. Este trabalho faz uso da abordagem qualitativa, em que considera-se a existência de uma relação dinâmica entre o mundo real e o sujeito, isto é, um vínculo indissociável entre o mundo objetivo e a subjetividade do sujeito, que não pode ser traduzida em números [20], [21]. A abordagem qualitativa não requer o uso de métodos e técnicas estatísticas, e por isso, que não são utilizadas neste trabalho.

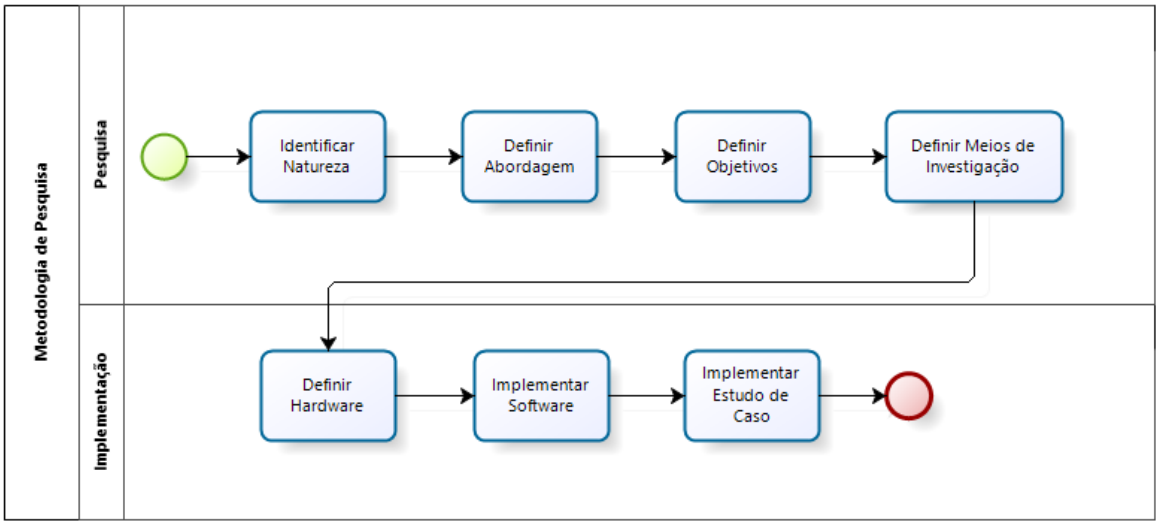

Figura 1.1: Processo de metodologia de pesquisa.

Posteriormente, foi realizada a definição dos objetivos. Este trabalho faz uso do objetivo exploratório, que proporciona maior familiaridade com o problema, com vistas a torná-lo mais explícito ou a constituir hipóteses [20], [21]. Os objetivos deste trabalho são definidos na Seção 1.3.

Por fim, foi realizada a definição dos meios de investigação. Este trabalho faz uso do meio de investigação bibliográfica, estudo que envolve levantamento bibliográfico com base na busca de trabalhos relacionados ao tema deste trabalho em bibliotecas digitais [20], [21]. Os trabalhos relacionados a este trabalho são compostos por 4 grandes áreas de pesquisas: IoT, RFID, computação em nuvem e microsserviços. São áreas de pesquisa que estão em constante transformação, tendo em vista a evolução quase recorrente nos últimos anos.

Após a identificação da pesquisa, é descrito o processo de identificação da implementação da arquitetura proposta. A primeira tarefa consiste na definição do *hardware* de IoT para leitura de tags RFID de UHF. Com base no meio de investigação bibliográfica foi definido o *hardware* adequado para implementação da arquitetura, respeitando os objetivos definidos na Seção 1.3.

A próxima tarefa foi a implementação do *software* responsável pela leitura de tags RFID a partir do *hardware* de IoT, assim como a comunicação das informações geradas pela leitura de tags, utilizando computação em nuvem e microsserviços. A arquitetura proposta na parte de *hardware* e *software* está definida no Capítulo 3.

Por fim, foi realizado um estudo de caso, método de procedimento que constitui em etapas mais concretas da investigação [22]. O estudo de caso se fez necessário para avaliar

e validar a arquitetura proposta neste trabalho.

1.5 Contribuições

As principais contribuições que pretendem-se obter a partir deste trabalho são:

- Utilização de leitor de tags RFID com frequência UHF de baixo custo de mercado em relação aos que utilizam essa frequência;
- Implementação de monitoramento do *hardware* de IoT;
- Utilização de computação em nuvem e microsserviços;
- Implementação da arquitetura proposta aplicada em um estudo de caso real;
- Disponibilização da implementação da arquitetura proposta para que outros pesquisadores possam se beneficiar da arquitetura desenvolvida, efetuando testes e adequando-a em outros contextos.

1.6 Estrutura da Dissertação

Este trabalho está organizado em 4 Capítulos além deste:

- **Capítulo 2 - Fundamentação Teórica:** Neste Capítulo são apresentadas a fundamentações teóricas e os trabalhos relacionados aos assuntos necessários para o entendimento deste trabalho;
- **Capítulo 3 - Implementação:** Neste Capítulo é apresentada a implementação proposta para o desenvolvimento deste trabalho;
- **Capítulo 4 - Estudo de Caso:** Neste Capítulo é apresentado um estudo de caso aplicando a arquitetura desenvolvida em um contexto real;
- **Capítulo 5 - Conclusão:** Neste Capítulo é apresentada a conclusão deste trabalho, bem como os trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta uma revisão dos principais conceitos relacionados ao tema desta dissertação. A Seção 2.1 apresenta os conceitos sobre Internet das Coisas. A Seção 2.2 apresenta os conceitos sobre Identificação por Radiofrequência. A Seção 2.3 apresenta os conceitos sobre computação em nuvem. A Seção 2.4 apresenta os conceitos sobre microsserviços. A Seção 2.5 apresenta os principais trabalhos relacionados ao estudo deste trabalho. E por fim, a Seção 2.6 apresenta uma discussão acerca da implementação proposta em relação à fundamentação teórica e aos trabalhos relacionados.

2.1 Internet das Coisas

A IoT é um paradigma para a construção de sistemas computacionais distribuídos pela Internet, nos quais, em princípio, os mais diversos dispositivos, objetos e coisas estarão conectados e interagindo com aplicativos para estender diversos serviços às pessoas [9] [10]. A IoT permite que todos os tipos de "coisas", que são objetos do mundo real, sejam conectadas à Internet e interajam entre si com o mínimo de intervenção humana [1], [2].

A IoT envolve dispositivos inteligentes conectados e sensores que geralmente não são intrusivos [11]. A diferença entre a IoT e as redes de sensores se dá com a inteligência que a IoT tem em seu domínio, transformando informações e ações em conhecimento. Desta forma, esse conhecimento alimenta uma rede, criando novas ações e informações [9], [23].

A IoT permite que pessoas e objetos sejam conectados a qualquer hora, em qualquer lugar, com qualquer dispositivo, com qualquer serviço e utilizando qualquer caminho de comunicação. A comunicação é frequentemente feita de forma autônoma e realizada em uma rede [1]. A Figura 2.1 apresenta a definição mais ampla relacionada a IoT. O fluxo de trabalho básico da IoT pode ser descrito da seguinte forma [2]:

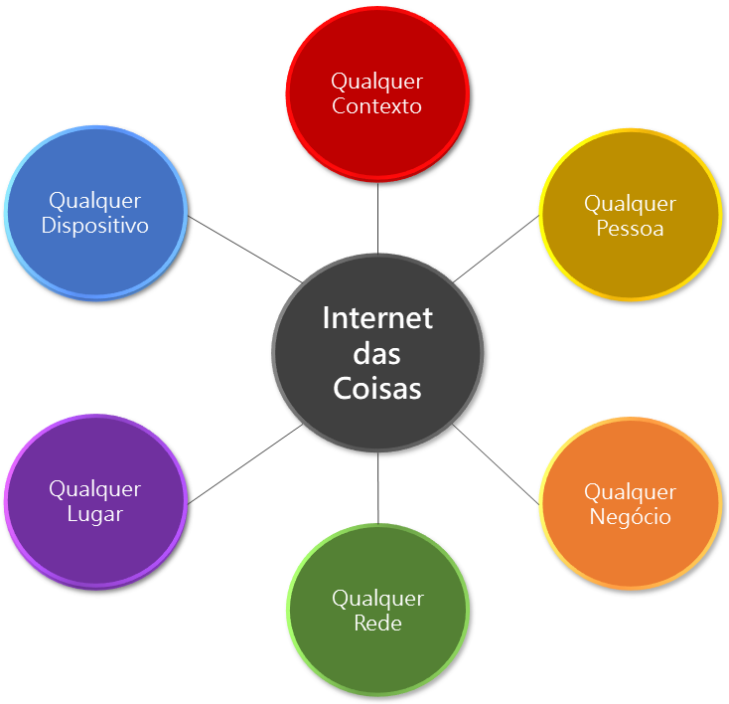

Figura 2.1: Definição mais ampla sobre IoT. Adaptado de [1].

- **Detecção de objetos:** as informações dos objetos são os dados detectados sobre temperatura, orientação, movimento, vibração, aceleração, umidade, mudanças químicas no ar, dentre outras, a depender do tipo de sensor;
- **Ação do objeto:** as informações dos objetos são processadas por um dispositivo inteligente que determina uma ação automatizada a ser executada;
- **Situação do dispositivo:** o dispositivo inteligente fornece serviços e um mecanismo para fornecer o *feedback* sobre a situação atual do dispositivo e os seus resultados.

No contexto da IoT, os dispositivos possuem cinco características fundamentais [24], [25]: Conectado à Internet, Capacidade de Detecção e Atuação, Inteligência Incorporada e Capacidade de Comunicação Interoperável.

1. **Conectado à Internet:** os dispositivos devem estar conectados à Internet usando conexões com ou sem fio;
2. **Único:** os dispositivos são exclusivamente identificáveis por meio de uma rede;
3. **Capacidade de detecção e atuação:** os dispositivos são capazes de realizar tarefas de detecção/atuação de forma autônoma;

4. **Inteligência incorporada:** os dispositivos possuem funções de inteligência e são capazes de autoconfigurar-se;

5. **Capacidade de comunicação interoperável:** o sistema IoT possui uma capacidade de comunicação baseada em tecnologias padrões.

A arquitetura em IoT é definida como uma estrutura para a especificação dos componentes físicos de uma rede, sua organização e configuração funcional, seus princípios e procedimentos operacionais, bem como formatos de dados usados em sua operação [24]. A arquitetura da IoT é definida em cinco camadas [2]: Camada de Percepção, Camada de Rede, Camada de *Middleware*, Camada de Aplicação e Camada de Negócio. A Figura 2.2 apresenta a arquitetura de IoT.

Figura 2.2: Arquitetura em camadas da IoT. Adaptado de [2].

A arquitetura IoT em camadas compreende [2]:

- **Camada de Percepção:** a Camada de Percepção também é conhecida como Camada do Dispositivo. Consiste em objetos físicos e dispositivos. Os dispositivos podem ser RFID, código de barras, sensor infravermelho, dependendo do método de identificação de objetos. Esta camada trata basicamente da identificação e coleta de informações específicas de objetos pelos dispositivos. Conforme o tipo de sensores, as informações podem ser sobre localização, temperatura, orientação, movimento, vibração, aceleração, umidade, dentre outras. As informações coletadas

são enviadas para a Camada de Rede, para uma transmissão segura para ao sistema de processamento de informações (Figura 2.2);

- **Camada de Rede:** a Camada de Rede também é conhecida como Camada de Transmissão. Esta camada transfere com segurança as informações dos dispositivos e sensores para o sistema de processamento de informações. O meio de transmissão pode ser com fio ou sem fio, e a tecnologia pode ser 3G/4G, *Wifi*, *bluetooth*, dentre outras. A Camada de Rede transfere as informações da Camada de Percepção para a Camada de *Middleware* (Figura 2.2);
- **Camada de *Middleware*:** os dispositivos de IoT implementam diferentes tipos de serviços. Cada dispositivo se conecta e se comunica apenas com os outros dispositivos que implementam o mesmo tipo de serviço. Esta camada é responsável pelo gerenciamento de serviços e possui um *link* para o banco de dados. Ele recebe as informações da camada de rede e as armazena no banco de dados. A Camada de *Middleware* realiza processamento de informações e computação onipresente e toma decisões automáticas, com base nos resultados (Figura 2.2);
- **Camada de Aplicação:** esta camada fornece o gerenciamento global do dispositivo, com base nas informações de objetos processadas na Camada de *Middleware*. As aplicações implementadas pela IoT podem ser de saúde inteligente, agricultura inteligente, casa inteligente, cidade inteligente, transporte inteligente, dentre outras (Figura 2.2);
- **Camada de Negócios:** esta camada é responsável pelo gerenciamento e gestão do sistema geral da IoT, incluindo as aplicações e serviços. Ela constrói modelos de negócios e gráficos com base nos dados recebidos da Camada de Aplicação. O verdadeiro sucesso da tecnologia IoT também depende dos bons modelos de negócios. Baseado na análise dos resultados, esta camada ajudará a determinar as ações futuras e estratégias de negócios (Figura 2.2).

A IoT permite que muitas aplicações sejam criadas em domínios de aplicações diferentes. O domínio de aplicação pode ser dividido principalmente em três categorias, baseadas em seu foco: indústria, meio ambiente e sociedade. Aplicações de transporte e logística, aeroespacial, aviação e automotivo, são algumas das aplicações focadas na indústria. Telecomunicação, tecnologia médica, saúde, construção inteligente, casa e escritório, mídia, entretenimento e emissão de bilhetes, são algumas das aplicações focadas na sociedade. Agricultura, reciclagem, alerta para desastres e monitoramento ambiental, são algumas das aplicações focadas no meio ambiente. A IoT apresenta vários benefícios. Os principais benefícios são: melhorar a eficiência de aplicações que necessitam de in-

teração com dispositivos físicos, aumentar a flexibilidade e interações das aplicações e o constante barateamento de aplicações que utilizam IoT [1], [2].

Neste trabalho, é utilizado a arquitetura padrão em IoT definida por Khan [2] no desenvolvimento da arquitetura proposta, além de utilizar a IoT para identificação do dispositivo de leitura de tags RFID com a frequência UHF de baixo custo presente na arquitetura proposta.

2.2 RFID

A RFID é uma tecnologia para identificação automatizada de objetos e pessoas, sendo uma das principais tecnologias de IoT. A tecnologia RFID apresenta grande penetração no mercado e tem sido utilizada em várias áreas de aplicação: gestão de transporte e logística, sistemas de estacionamento inteligentes, gestão de resíduos, pecuária, monitoramento de pacientes, perfuração de petróleo, controle de qualidade, rastreamento de ativos, entre outras aplicações [7], [26].

A RFID já é utilizada amplamente nos dias de hoje. Os exemplos incluem cartões de proximidade, pagamento automático de pedágios, *tokens* de pagamento, as chaves de ignição de muitos automóveis, que além disso, incluem etiquetas RFID como um dispositivo de segurança para impedir roubos, entre outros [7]. A RFID utiliza ondas de rádio e campos eletromagnéticos para ler ou gravar automaticamente informações armazenadas em etiquetas RFID, também conhecidas como tags [3], [7]. As tags RFID possuem um identificador único conhecido como EPC, responsável por identificar de forma exclusiva uma tag rastreada [27], [28]. Por questão de baixo custo, as tags EPC aderem a um design minimalista. Elas carregam poucos dados na memória interna. O código EPC de uma tag RFID inclui as informações necessárias para a identificação de um item rastreado dentro de um banco de dados [7]. A RFID aparece como o sucessor do código de barras, mas com diferenças fundamentais [3], [7]. As principais características de uma tag RFID que os diferenciam do código de barras são:

- **Identificação exclusiva:** um código de barras indica o tipo de objeto no qual é impresso. Uma tag RFID emite um número de série único, EPC, que distingue entre milhões de objetos manufaturados de forma idêntica. O código EPC de uma tag RFID pode atuar como identificador único para um banco de dados;
- **Automação:** os códigos de barras, sendo digitalizados opticamente, exigem contato da linha com os leitores e, portanto, um cuidadoso posicionamento físico dos objetos digitalizados. Exceto nos ambientes mais rigorosamente controlados, a leitura de código de barras requer intervenção humana. Em contraste, as tags RFID são

legíveis sem contato de linha de visão e sem posicionamento preciso. Os leitores de RFID podem digitalizar tags a taxas de centenas por segundo.

Existem três tipos de tags RFID em relação ao seu poder energético: passiva, semi-passiva e ativa [3], [7], [8]:

- **Tags RFID passivas:** não possuem fontes de energia interna. Ao receberem o sinal de rádio frequência do leitor de tag RFID, parte dessa energia é transformada em corrente elétrica dentro da tag RFID, que por fim enviam um sinal de resposta com informações presentes no chip na tag RFID. A tag passiva possui um custo menor do que os outros tipos de tags, mas é necessário que o leitor de tag RFID seja de maior potência, devido à quantidade de energia necessária para emissão do sinal de resposta, presentes na tag RFID [7], [3];
- **Tags RFID semi-passivas:** possuem fonte de energia utilizando uma bateria interna. A bateria interna é ativada ao receber um sinal de rádio frequência do leitor de tag RFID. A bateria interna é utilizada para alimentar o chip presente dentro da tag RFID, enquanto a energia utilizada para comunicação é recebida pelo leitor de tag RFID [3], [7];
- **Tags RFID ativas:** assim como as tags RFID semi-passivas, possuem fonte de energia utilizando uma bateria interna. A bateria interna é ativada ao receber um sinal de rádio frequência do leitor de tag RFID. A tag RFID ativa consegue emitir de forma contínua o seu sinal, independente do recebimento do leitor de tag RFID. A tag ativa possui um custo maior do que os outros tipos de tags, mas podem ser identificadas em distâncias maiores de 100 metros e possuem maiores recursos dentro do chip na tag RFID [3], [7].

A identificação de leitura de uma tag RFID ocorre quando a mesma está dentro de uma proximidade de um leitor RFID e o sinal de rádio é transmitido pelo leitor junto à tag. A tag recebe o sinal e então se identifica com o leitor pelo seu identificador único EPC. O leitor capta o sinal enviado pela tag decodifica-o e transmite a informação para um sistema de *software* responsável por realizar o processamento da informação da tag RFID [3].

Essa transmissão pode ocorrer de várias formas, como a partir da Internet, *Wifi*, comunicação serial, *bluetooth*, dentre outras [8], [29]. A Figura 2.3 apresenta a arquitetura de como ocorre a identificação de uma tag RFID.

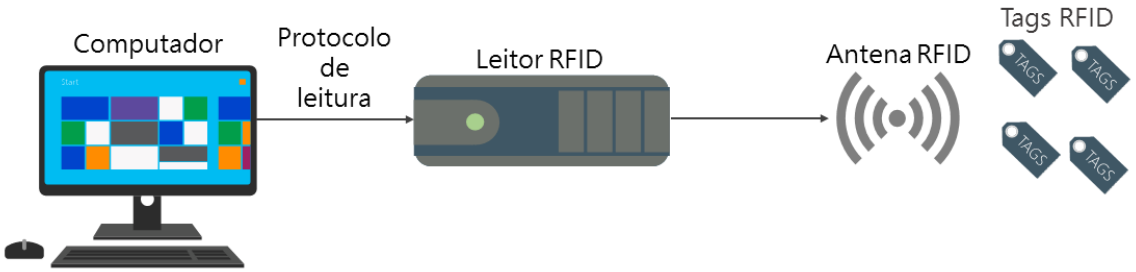

Figura 2.3: Arquitetura de identificação de leitura de tag RFID. Adaptado de [3].

Os sistemas RFID podem ser classificados pelo seu tipo de frequência e região de frequência [19]: Baixa Frequência (LF), Alta Frequência (HF) e Ultra Alta Frequência (UHF). A Tabela 2.1 apresenta a classificação da RFID em relação à sua frequência e características [3], [7], [8].

Tabela 2.1: Classificação da RFID em relação à frequência [3], [7], [8].

Tipo	**Região**	**Alcance**	**Características**
Baixa Frequência (LF)	30-500 Quilo-hertz (KHz)	Até 0.5 metro	Custo baixo e alcance baixo de leitura e gravação.
Alta Frequência (HF)	10-15 Mega-hertz (MHz)	Até 1 metro	Potencialmente de baixo custo e alcance médio de leitura e gravação.
Ultra Alta Frequência (UHF)	850-950 Mega-hertz (MHz), 2.4-3.5 Giga-hertz (GHz) e 5.8 Giga-hertz (GHz)	Dezenas de metros	Alto custo e alcance alto de leitura e gravação.

Há normas e padrões que definem os dispositivos RFID. A *ISO 18000* é uma norma que especifica protocolos para diversas frequências diferentes, incluindo bandas as LF, HF e UHF. Para tags UHF, o padrão predominante é o *EPCglobal Class-1 Gen-2*. Para tags HF, existem dois padrões principais além da *ISO 18000*. A *ISO 14443* é um padrão para dispositivos RFID, com uma faixa nominal de operação de 10 centímetros. O *ISO 15693* é um padrão de HF para dispositivos RFID, com faixas nominais maiores, até 1 metro para configurações de antenas grandes [7].

Diferentemente dos códigos de barras, as tags RFID não precisam estar dentro da linha de visão do leitor para serem rastreadas, ocasionando maior eficiência e velocidade na busca de itens com tags RFID. Em comparação com o tipo de frequência LF e HF, a frequência UHF possui um alcance de leitura e gravação mais amplo. Entretanto,

em comparação às demais frequências, o custo de implementação utilizando esse tipo de frequência é mais alto [3], [7].

Neste trabalho, é utilizada a etiqueta passiva e a frequência UHF, respectivamente, devido ao custo menor em comparação aos outros tipos de tags e ao alcance alto de distância de leitura das tags RFID.

2.3 Computação em Nuvem

A computação em nuvem é um tipo de sistema paralelo e distribuído, que consiste em uma coleção de computadores interconectados e virtualizados, dinamicamente provisionados e disponibilizados como recursos computacionais (por exemplo, redes, servidores, armazenamento, aplicações) [13]. Os recursos computacionais são disponibilizados com base em SLA estabelecido por meio de negociação entre os provedores de serviços de computação em nuvem e seus clientes. Os ambientes em computação em nuvem são rapidamente configurados e liberados com um mínimo esforço de gerenciamento ou de interação com os provedores de serviços [4].

O termo nuvem, presente em computação em nuvem, denota a infraestrutura onde os usuários podem acessar aplicações incluídas no ambiente de qualquer lugar do mundo, a qualquer momento. Assim, o mundo da computação está se transformando rapidamente em desenvolvimento de softwares utilizando serviços em nuvem, ao invés de executar esses softwares em ambientes e infraestruturas locais [4], [30].

As plataformas de computação em nuvem possuem características de computação em grade e clusterização, com atributos e recursos especiais, como suporte para virtualização, serviços dinamicamente compostos com interfaces Web para gerenciamento e facilidade na configuração de ambientes. A computação em nuvem fornece serviços aos usuários sem referência à infraestrutura na qual eles estão hospedados [13].

A arquitetura de computação em nuvem provê modelos de serviços, sendo que cada um trata de uma particularidade na disponibilização de recursos para as aplicações: Infraestrutura como um Serviço (IaaS), *Software* como um Serviço (SaaS) e Plataforma como um Serviço (PaaS) [4], [30], [31]. No modelo IaaS, o fornecedor do serviço disponibiliza servidores como máquinas virtuais que são consumidas como serviços. O cliente desse tipo de serviço é responsável por toda a configuração nesse tipo de modelo. No modelo SaaS, o fornecedor do serviço se responsabiliza por toda a estrutura necessária para disponibilização do sistema, como infraestrutura, segurança e conectividade. O cliente desse tipo de serviço paga um valor pelo serviço que é acessado via Internet. O modelo PaaS é um meio termo entre o IaaS e SaaS, onde o provedor do serviço oferece os dois em um

único modelo [13]. A Figura 2.4 apresenta as camadas da arquitetura da computação em nuvem, as quais são:

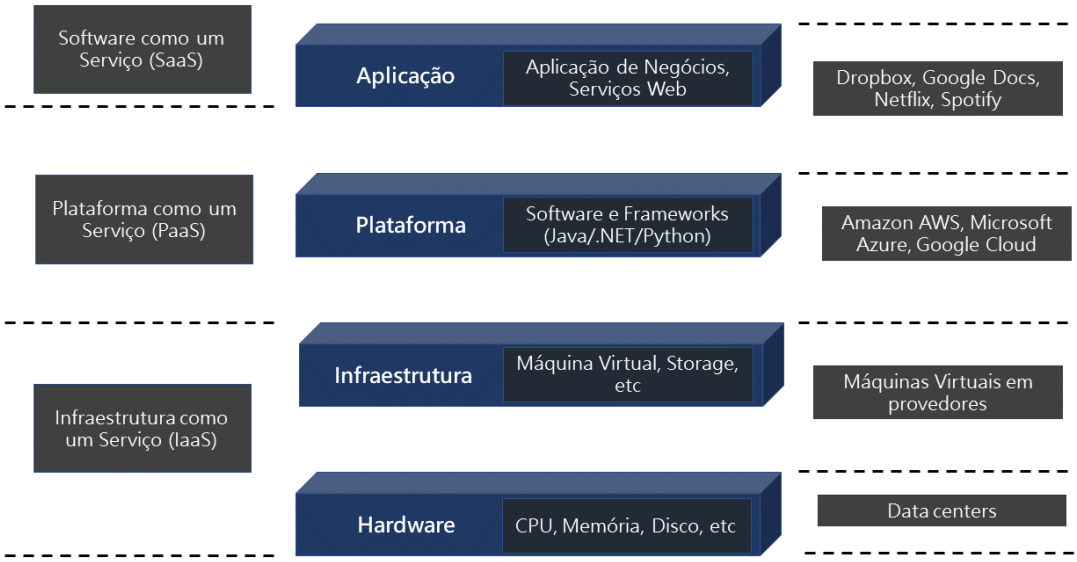

Figura 2.4: Arquitetura da Computação em Nuvem. Adaptado de [4].

- **Infraestrutura como um Serviço (IaaS):** fornece a capacidade de provisionar processamento, armazenamento, redes, incluir sistemas operacionais, serviços e aplicações. O cliente tem controle total da infraestrutura utilizando máquinas virtuais. A IaaS pode escalar dinamicamente, aumentando ou diminuindo os recursos de acordo com as necessidades das aplicações. Os gastos das aplicações IaaS estão diretamente vinculados ao consumo [4], [30], [31]. Exemplos de aplicações como IaaS: máquinas virtuais presentes em provedores de computação em nuvem (*Amazon AWS*, *Microsoft Azure*, *Google Cloud*, dentre outros);
- ***Software* como um Serviço (SaaS):** fornece todas as funções de uma aplicação tradicional, mas por acesso via Internet. O cliente não tem controle na infraestrutura de servidores, rede, sistemas operacionais e armazenamento. O SaaS elimina preocupações com servidores de aplicações, sistemas operacionais, armazenamento e desenvolvimento de aplicações. Os desenvolvedores concentram-se no desenvolvimento de aplicações e não na infraestrutura, permitindo o desenvolvimento rápido de sistemas de software. O SaaS reduz os custos, pois é dispensada a aquisição de licenças de sistemas de software e os usuários usam o serviço sob demanda [4], [30], [31]. Exemplos de aplicações como SaaS: *Dropbox*, *Google Docs*, *Netflix*, *Spotify*, dentre outras;

- **Plataforma como um Serviço (PaaS):** fornece um sistema operacional, linguagens de programação e ambientes de desenvolvimento para as aplicações, auxiliando o desenvolvimento de sistemas de software, que são suportadas pelo provedor de serviço de computação em nuvem. O cliente não tem controle na infraestrutura de servidores, rede, sistemas operacionais e armazenamento, mas tem controle sobre as aplicações enviadas e algumas configurações de ambiente que são disponibilizados pelos provedores de computação em nuvem. O modelo PaaS possui as mesmas características do modelo SaaS no desenvolvimento de aplicações [4], [30], [31]. Exemplos de aplicações como PaaS: provedores de computação em nuvem (*Amazon AWS*, *Microsoft Azure*, *Google Cloud*, dentre outros).

A empresa de consultoria *Gartner* apresenta anualmente um relatório sobre os principais provedores de computação em nuvem. O quadrante mágico, representação gráfica desenvolvida pela *Gartner* é dividido em 4 tópicos que definem: líderes, desafiadores, visionários e concorrentes de nicho de mercado. Em seu último relatório realizado no ano de 2018, a avaliação é orientada a IaaS com alguns recursos que podem ser classificados como PaaS [5]. Não há relatórios específicos para os modelos de serviços SaaS e PaaS. A Figura 2.5 apresenta os líderes mundiais de mercado em computação em nuvem.

Figura 2.5: Quadrante mágico para IaaS em nuvem. Adaptado de [5].

Além de prover modelos de serviços, a arquitetura da computação em nuvem provê modelos de implantação: público, privado, híbrido, comunidade e federado [13], [32]. O modelo público é o modelo de implementação mais comum, e possui a estrutura de rede aberta para uso público. O cliente utiliza o serviço mediante pagamento, a ser definido de acordo com os serviços utilizados. O modelo privado possui estrutura de rede fechada.

O cliente é responsável pelo controle interno de todos os serviços. O modelo híbrido é um meio termo entre o modelo privado e público. Os serviços utilizados pelo cliente podem estar tanto em uma nuvem privada, quanto em uma nuvem pública. O modelo em comunidade ocorre quando organizações constroem e compartilham, em conjunto, uma infraestrutura de nuvem. O modelo federado é um conjunto de provedores de nuvens públicos e privados [4], [30], [31], [32]. Os modelos de implantação da computação em nuvem são:

- **Público:** o modelo público permite o acesso dos usuários por meio de interfaces usando navegadores da Web. Os usuários precisam pagar apenas pelo tempo que utilizam o serviço, reduzindo os custos operacionais. No entanto, são menos seguras em comparação com outros modelos, pois todos os aplicativos e dados são mais propensos a ataques mal-intencionados. A solução para isso pode se dar nas verificações de segurança, a serem implementadas por meio de validação em ambos os lados, tanto pelo provedor de computação nuvem quanto pelo cliente;
- **Privado:** o modelo privado permite que as operações em nuvem ocorram dentro do *data center* de uma organização. Nesse modelo, é fácil gerenciar a segurança, a manutenção e as atualizações, além de fornecer mais controle sobre a implantação e o uso. A nuvem privada pode ser comparada à Intranet. Na nuvem privada, os serviços são reunidos e disponibilizados para os usuários no nível organizacional. Os recursos e aplicativos são gerenciados pela própria organização. A segurança é aprimorada, pois somente os usuários das organizações têm acesso à nuvem privada;
- **Híbrido:** o modelo híbrido é uma combinação dos modelos público e privado. Neste modelo, uma nuvem privada está vinculada a um ou mais serviços de nuvem externos. Permite que uma organização atenda às suas necessidades na nuvem privada e, se alguma necessidade ocasional ocorrer, ele solicita à nuvem pública recursos computacionais;
- **Comunidade:** o modelo em comunidade permite que organizações construam e compartilhem, em conjunto, uma infraestrutura de nuvem, seus requisitos e políticas. A infraestrutura da nuvem pode ser hospedada por um provedor terceirizado ou em uma das organizações da comunidade;
- **Federado:** o modelo federado permite que múltiplos provedores de nuvem distintos tem seus serviços e recursos integrados para o usuário final de maneira transparente, oferecendo um maior poder de processamento e de armazenamento.

Os provedores de computação em nuvem possuem *data centers* espalhados em vários locais do mundo para fornecer redundância e confiabilidade em relação aos serviços esta-

belecidos. Independente da arquitetura de serviço utilizada ou modelo de implementação, os provedores de computação em nuvem devem garantir que sua infraestrutura seja segura e que os dados e aplicações de seus clientes sejam protegidos, enquanto o cliente deve tomar medidas para fortalecer suas aplicações. A redução de custos da computação em nuvem em relação à infraestrutura local é evidente, além de ser uma tecnologia importante para interoperabilidade, flexibilidade, escalabilidade e provisionamento rápido de aplicações [4], [30], [31].

Neste trabalho, é utilizado o provedor de computação em nuvem *Microsoft Azure* devido à sua representatividade no mercado de computação em nuvem, conforme apresentado na Figura 2.5, além de possuir serviços e ferramentas que facilitam a implementação da arquitetura proposta e por ser o provedor de computação em nuvem utilizado no estudo de caso aplicado neste trabalho. Na arquitetura proposta é utilizado o modelo de implementação público e o modelo de serviço PaaS, em atenção ao menor custo de implementação e maior facilidade no desenvolvimento desse tipo de serviço.

2.4 Microsserviços

Microsserviços são estilos arquiteturais, inspirados nas práticas ágeis que surgiram na indústria de software, com o objetivo de aumentar a capacidade das equipes de desenvolvimento, de construir e manter grandes aplicações em ambientes corporativos. Neste estilo, uma aplicação é construída pela composição de vários microsserviços [14], [33]. Um microsserviço é um serviço pequeno e autônomo, que se comunica através de uma infraestrutura de rede e protocolos. Para um microsserviço ser considerado pequeno e autônomo, deve ter uma única responsabilidade e possuir sua própria infraestrutura, que é independente dos outros microsserviços no qual está relacionado [15], [33].

Microsserviços gerenciam a complexidade crescente decompondo funcionalmente dos grandes sistemas em um conjunto de serviços independentes. Ao tornar os serviços completamente independentes em desenvolvimento e implantação, os microsserviços enfatizam o baixo acoplamento e a alta coesão, levando a modularidade dos microsserviços implementados. Essa abordagem oferece todos os tipos de benefícios em termos de capacidade de manutenção, escalabilidade, flexibilidade, modularidade, entre outros [14]. As principais características de um microsserviço são [14]:

- **Tamanho:** o tamanho é comparativamente pequeno. Um serviço típico, apoiando a crença de que o projeto arquitetônico de um sistema é altamente dependente do projeto estrutural da organização que o produz. O uso idiomático da arquitetura de microsserviços sugere que, se um serviço for muito grande, ele deve ser dividido

em dois ou mais serviços, preservando assim a granularidade e mantendo o foco em fornecer apenas uma única capacidade de negócio;

- **Contexto delimitado:** as funcionalidades relacionadas são combinadas em um único recurso de negócio, que é então implementado como um serviço;
- **Independência:** cada serviço na arquitetura de microsserviço é operacionalmente independente de outros serviços e a única forma de comunicação entre os serviços é por meio de suas interfaces publicadas.

Para um microsserviço ser pequeno, autônomo e com responsabilidade única, o conceito de contextos delimitados foi importado do padrão arquitetural *Domain-Driven Design* (DDD) onde as funções de negócio de um serviço devem ser construídas e executadas independente de outros serviços [34]. As principais características de sistema arquitetado com microsserviços são [14]:

- **Disponibilidade:** a disponibilidade é uma grande preocupação em microsserviços, pois afeta diretamente o sucesso de um sistema. Dada independência dos serviços, toda a disponibilidade do sistema pode ser estimada em termos da disponibilidade dos serviços individuais que compõem o sistema. Mesmo que um único serviço não esteja disponível para atender a uma solicitação, todo o sistema pode ficar comprometido e sofrer consequências diretas. Se levarmos a implementação do serviço, quanto mais um componente propenso a falhas for, mais frequentemente o sistema sofrerá falhas. Os microsserviços devem ser impedidos de se tornar excessivamente complexos, refinando-os em dois ou mais serviços diferentes. Gerar um número crescente de serviços tornará o sistema propenso a falhas no nível de integração, o que resultará em menor disponibilidade devido à grande complexidade associada à disponibilização instantânea de dezenas de serviços;
- **Confiabilidade:** construir o sistema a partir de componentes pequenos e simples também é uma das regras, que afirma que, para alcançar maior confiabilidade, é preciso encontrar uma maneira de gerenciar as complexidades de um sistema grande. A maior ameaça à confiabilidade de microsserviços está nos mecanismos de integração. A confiabilidade de microsserviços é inferior aos sistemas que usam chamadas na memória ao invés de chamadas através da rede. Essa desvantagem não é exclusiva apenas dos microsserviços e pode ser encontrada em qualquer sistema distribuído;
- **Manutenção:** por definição, a arquitetura de microsserviços é fracamente acoplada, o que significa que existe um pequeno número de integrações entre serviços.

Isso contribui muito para a manutenção de um sistema, minimizando os custos de modificações de serviços, corrigindo erros ou adicionando novas funcionalidades.

- **Desempenho:** o fator proeminente que afeta negativamente o desempenho na arquitetura de microsserviços é a comunicação em uma rede. A latência da rede é muito maior que a da memória. As chamadas na memória são muito mais rápidas para serem concluídas do que o envio de mensagens pela rede. Em termos de comunicação, o desempenho será degradado em comparação com sistemas que usam mecanismos de chamada de memória. As restrições que os microsserviços dão ao tamanho dos serviços também contribuem indiretamente para esse fator. Em arquiteturas mais gerais sem restrições relacionadas ao tamanho, a proporção de chamadas na memória para o número total de chamadas é maior do que na arquitetura de microsserviços, o que resulta em menos comunicação na rede. Assim, a quantidade exata de degradação do desempenho também dependerá da interconectividade do sistema. Dessa forma, sistemas com contextos bem delimitados sofrerão menos degradação, devido ao acoplamento mais flexível e à menor quantidade de mensagens enviadas;

- **Segurança:** os microsserviços sofrem vulnerabilidades de segurança. Como os microsserviços usam o mecanismo REST e o XML com JSON como principais formatos de troca de dados, uma atenção especial deve ser dada ao fornecimento de segurança dos dados que estão sendo transferidos. Isso significa adicionar mais sobrecarga ao sistema em termos de funcionalidade adicional de criptografia. Os microsserviços promovem a reutilização de serviços e, como tal, é natural supor que alguns sistemas incluirão serviços de terceiros. Portanto, um desafio adicional é fornecer mecanismos de autenticação com serviços de terceiros e garantir que os dados enviados sejam armazenados de forma segura. Em resumo, a segurança de microsserviços é afetada de maneira bastante negativa porque é necessário considerar e implementar mecanismos de segurança adicionais para fornecer a funcionalidade de segurança adicional mencionada acima;

- **Testabilidade:** como todos os componentes em uma arquitetura de microsserviços são independentes, cada componente pode ser testado isoladamente, o que melhora significativamente a testabilidade do componente. Com microsserviços, é possível isolar partes do sistema que sofreram alterações e partições que foram afetadas pela mudança e que as consideraram independentemente do resto do sistema. O teste de integração, por outro lado, pode se tornar muito complicado, especialmente quando o sistema que está sendo testado é muito grande e há muitas conexões entre os

componentes. É possível testar cada serviço individualmente, mas as anomalias podem emergir da colaboração de vários serviços.

A integração de um sistema arquitetado com microsserviços acontece por meio de padrões de projeto como: *API Gateway* e *Backend por front-end* para comunicação externa entre serviços, mensageria para o uso mensagens assíncronas para comunicação interna entre serviços, entre outros padrões de projeto [34].

Como cada microsserviço pode representar um único recurso de negócio que é entregue e atualizado de forma independente, descobrir um erro e/ou alguma melhoria não causará impacto em outros serviços e em seu desenvolvimento (desde que a compatibilidade com versões anteriores seja preservada e a interface de serviço permaneça inalterada) [14]. No entanto, para aproveitar verdadeiramente o poder da implantação independente, é preciso utilizar mecanismos de integração e entrega contínua. Os microsserviços são a primeira arquitetura desenvolvida na era de *DevOps*, essencialmente, os microsserviços devem ser usados com entrega contínua e integração contínua, tornando cada estágio do *pipeline* de entrega automático. Ao usar *pipelines* automatizados de entrega contínua e modernas ferramentas de *container*, por exemplo, é possível implantar uma versão atualizada de um serviço para produção em questão de segundos, o que prova ser muito benéfico em ambientes de negócios em constante mudança [14].

Os microsserviços estão em contraste com os sistemas monolíticos, que tendem a colocar toda a funcionalidade de um serviço em um aplicativo único e bem coordenado [6]. A abordagem monolítica e de microsserviços são apresenta na Figura 2.6 e Figura 2.7.

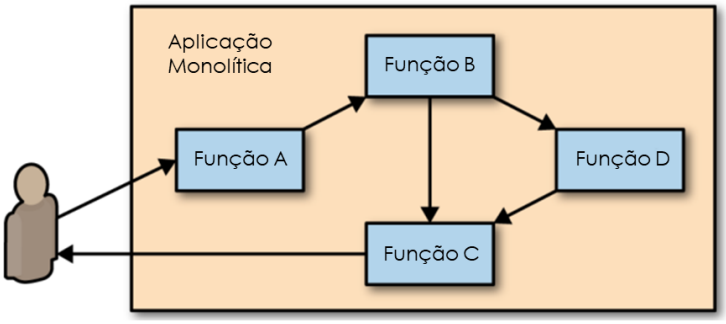

Figura 2.6: Arquitetura monolítica com todas as funções em um único sistema. Adaptado de [6].

Em uma aplicação monolítica, uma aplicação de *software* não pode ser executada de forma independente. O *software* é construído com o mesmo conjunto de tecnologia e infraestrutura. Um grande problema nesse estilo arquitetural é a adição ou atualização

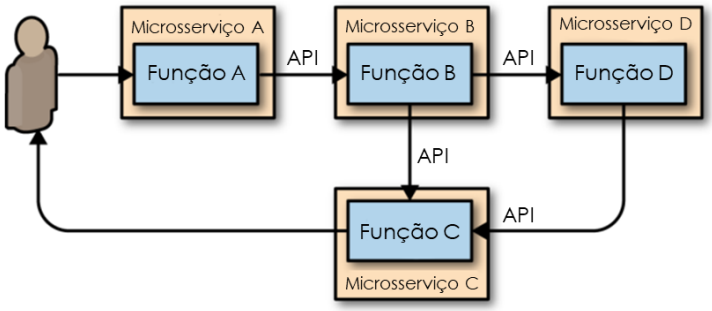

Figura 2.7: Arquitetura de microsserviços com cada função dividida em um microsserviço separado. Adaptado de [6].

de linguagens de programação, *framework*, banco de dados ou infraestrutura que afeta toda a construção da aplicação [6], [14], [33].

O desacoplamento de microsserviços permite um melhor dimensionamento. Como cada componente foi dividido em seu próprio serviço, ele pode ser dimensionado de forma independente. É raro que cada serviço dentro de um aplicativo maior cresça na mesma taxa ou tenha a mesma forma de dimensionamento. Alguns sistemas são sem estado e podem simplesmente escalar horizontalmente, enquanto outros sistemas mantêm o estado e exigem *sharding* ou outras abordagens para escalar. Na separação, cada serviço pode usar a abordagem de dimensionamento mais adequada. Mas isso não é possível quando todos os serviços fazem parte de um único monolito [6].

Os Microsserviços possuem uma separação clara de todos os componentes de uma aplicação de *software*, diferente do monolítico, que centraliza todos os componentes. Cada microsserviço tem sua própria infraestrutura e definição de negócio bem definida, podendo ser compartilhada entre outros serviços. Uma aplicação de *software* é construída pela decomposição de vários microsserviços. O desenvolvimento de microsserviços além de ser importante para o conhecimento dos contextos delimitados de uma aplicação, facilita a manutenção e o provisionamento rápido de aplicações [6].

A mudança para microsserviços é uma questão sensível para várias empresas envolvidas em uma grande refatoração de seus sistemas *back-end* [14]. Também há desvantagens na abordagem de microsserviços para o design do sistema. As duas principais desvantagens são que, como o sistema se tornou mais fracamente acoplado, depurar o sistema quando ocorrem falhas é significativamente mais difícil. Não se pode mais simplesmente carregar um único aplicativo em um depurador e determinar o que deu errado. Quaisquer erros são os subprodutos de um grande número de sistemas, muitas vezes em execução em máquinas diferentes. Esse ambiente é bastante desafiador para reproduzir em um depurador [6].

Os sistemas baseados em microsserviços também são difíceis de projetar e arquitetar. Um sistema baseado em microsserviços usa vários métodos de comunicação entre serviços, padrões diferentes (por exemplo, síncrono, assíncrono, mensageria, etc.) e múltiplos padrões diferentes de coordenação e controle entre os serviços. Esses desafios são a motivação para padrões distribuídos. Se uma arquitetura de microsserviços é composta de padrões conhecidos, é mais fácil projetar, porque muitas das práticas de projeto são especificadas pelos padrões. Além disso, os padrões facilitam a depuração dos sistemas, pois permitem que os desenvolvedores apliquem as lições aprendidas em vários sistemas diferentes que usam os mesmos padrões [6].

Neste trabalho, o estilo arquitetural de microsserviços é utilizado para substituir a arquitetura monolítica presente no estudo de caso aplicado neste trabalho, separando os contextos delimitados na arquitetura proposta com o uso do padrão arquitetural *Domain-Driven Design* (DDD) [34], além de facilitar a integração com a computação nuvem devido aos serviços descentralizados presentes no estilo arquitetural de microsserviços. *Pipelines* de integração e entrega contínua [14] é utilizado para automatizar publicações de serviços. Na arquitetura proposta são utilizados os principais padrões de projeto nesse tipo de arquitetura: *API Gateway*, para comunicação externa entre serviços e mensageria, para o uso mensagens de assíncronas para comunicação interna entre serviços.

2.5 Trabalhos Relacionados

No trabalho de Rayes [11] é afirmada a importância da arquitetura em IoT para gerenciar a complexidade de dispositivos inteligentes interconectados com sensores. A computação em nuvem permite provisionar serviços rapidamente, assim como garante alta disponibilidade e extensibilidade, importantes para gerenciar a complexidade de uma arquitetura de IoT [16], [23].

Martins et al. [33] apresenta a arquitetura em IoT utilizando computação em nuvem e microsserviços para disponibilização dos dados processados de um dispositivo IoT por intermédio do estilo arquitetural REST e arquitetura de microsserviços. Os trabalhos de Sun [16], Vresk [35], Celesti et al. [36] e Vandikas [37] apresentam a arquitetura em IoT utilizando computação em nuvem e microsserviços. Os trabalhos demonstram como as características da computação em nuvem e de microsserviços se completam na criação de arquiteturas que utilizam IoT.

As características de microsserviços facilitam a comunicação e integração com dispositivos de IoT [14], [15], [34]. Os trabalhos de Ferreira apresentam arquiteturas em IoT utilizando serviços no estilo arquitetural REST por sua facilidade de comunicação com dispositivos IoT [38], [39].

Tsiropoulou et al. [26] apresenta a comunicação com tag RFID passiva, dentro do contexto de um sistema de estacionamento inteligente e aplicando maior eficiência energética e operacional. Para aplicar a minimização de potência de transmissão do roteamento do leitor RFID com junto com a tag RFID passiva, é utilizado o paradigma de comunicação *tag-to-tag*. A comunicação *tag-to-tag* em redes RFID passivas difere fundamentalmente das redes *multi-hop* tradicionais devido à característica de reflexão de energia pelas tags RFID passivas e não à retransmissão da potência original recebida.

Os trabalhos de Prudanov et al. [17], Farris et al. [18] e Chiochan [19] apresentam arquiteturas de leitura de tags RFID de UHF utilizando IoT. Prudanov et al. [17] apresenta uma arquitetura de leitura de tags RFID utilizando a frequência UHF e IoT. A arquitetura proposta utiliza um leitor IoT para leitura de tags RFID de baixo custo *Cottonwood*. Um microcomputador *Raspberry Pi 3* é responsável pelo processamento das tags lidas pelo dispositivo IoT. A arquitetura proposta implementa autenticações de segurança na aplicação, a fim de aplicar confidencialidade nos dados.

Farris et al. [18] apresenta uma arquitetura de leitura e escrita de tags RFID utilizando a frequência UHF e IoT em um ambiente com comunicação via protocolo de comunicação IPV6. A arquitetura proposta utiliza um leitor IoT para leitura e gravação de tags RFID comercial de baixo custo *ThingMagic Micro Embedded*. As informações de leitura ou gravação de tags são enviadas para um servidor local utilizando comunicação serial.

Chieochan [19] demonstra uma arquitetura de leitura de tags RFID utilizando a frequência UHF, IoT e computação em nuvem. O modelo de implantação e o modelo de serviço de computação em nuvem não é informado no trabalho. A arquitetura proposta utiliza um leitor de tags RFID comercial de alto custo e um microcontrolador para realizar o processamento das tags lidas que são enviadas para um servidor local. A conexão entre o microcontrolador e o leitor de tags RFID é realizada a partir da comunicação serial utilizando um *hardware* UART. O SGBD MariaDB é responsável pelo armazenamento das informações no formato JSON. É utilizada computação em nuvem através do modelo de nuvem privada, que é responsável por armazenar as informações presentes no banco de dados e hospedar a aplicação Web responsável por apresentar os resultados. A utilização de *hardware* de baixo custo em IoT é fundamental para diminuição do custo de implementação de arquiteturas em IoT complexas. Há diversos dispositivos de IoT que possuem um preço de mercado acessível para empresas de pequeno porte e/ou usuários comuns.

As arquiteturas em IoT de leitura de tags RFID utilizando a frequência UHF apresentadas possuem características em comum, principalmente em termos de comunicação e integração entre os dispositivos IoT e as aplicações. Em sua maioria, as arquiteturas em IoT fazem uso da comunicação entre serviços utilizando o estilo arquitetural REST com

retorno da resposta dos serviços no formato JSON, assim como um SGBD com suporte a essa tecnologia.

2.6 Discussão de Implementação Proposta

Diferentemente das arquiteturas em IoT de leitura de tags RFID de UHF apresentadas, a arquitetura proposta neste trabalho faz uso de microsserviços com o uso de computação em nuvem, em virtude das características já apresentadas, como vantagens na implementação em relação a arquitetura monolítica, bem como serviços e ferramentas da computação em nuvem que facilitam a implementação e o provisionamento da arquitetura proposta.

Neste trabalho, é utilizado o provedor de computação em nuvem *Microsoft Azure*, em razação da sua representatividade no mercado de computação em nuvem além de possuir serviços e ferramentas que facilitam a implementação da arquitetura proposta e ser o provedor de computação em nuvem utilizado no estudo de caso aplicado neste trabalho.

A utilização do modelo de implementação público e o modelo de serviço PaaS diminuem o custo de implementação e facilitam o desenvolvimento da arquitetura proposta. O modelo de implementação público facilita a implementação por ser um modelo que não necessita de nenhum tipo de configuração interna ou externa, apenas o uso do navegador Web para acessar os serviços e ferramentas disponíveis na nuvem. O custo de implementação é menor, pois a utilização de serviços e ferramentas só ocorre quando há o consumo, tornando desnecessários os gastos com a não utilização de serviços e ferramentas.

Essas características são essenciais para o gerenciamento de grandes arquiteturas em Internet das Coisas, devido à escalabilidade e provisionamento rápido da arquitetura proposta no ambiente de computação em nuvem [40], assim como o gerenciamento do grande volume de dados que podem ser gerados na leitura de tags RFID, com a utilização da frequência UHF, aproximadamente 100 tags lidas por segundo. A utilização de um banco de dados NoSQL é importante para a quantidade de informações que podem ser geradas na leitura de tags RFID, pela quantidade de registros que podem ser inseridos e pela baixa latência presente nesse tipo de banco de dados [41].

A utilização da tecnologia de *containers Docker* [37], [40] facilita a integração entre microsserviços e a computação em nuvem, além de ser possível utilizar outros tipos de provedores de computação em nuvem que aceitam esse tipo de tecnologia [37], [40].

Por fim, as arquiteturas apresentadas não apresentam implementações para o monitoramento do *hardware* IoT. A arquitetura proposta neste trabalho apresenta o monitoramento do dispositivo IoT assim como o monitoramento dos microsserviços implementados.

Capítulo 3

Implementação Proposta

Este Capítulo apresenta a arquitetura proposta neste trabalho. A Seção 3.1 apresenta a arquitetura proposta de maneira abstrata, definindo as camadas da arquitetura, suas utilidades e relações. A Seção 3.2 apresenta com detalhes a arquitetura de Internet das Coisas, definindo o dispositivo IoT utilizado para leitura de tags RFID e o funcionamento da arquitetura de Internet das Coisas. A Seção 3.3 apresenta com detalhes a arquitetura de serviços, definindo como os microsserviços e a computação em nuvem são providos na arquitetura.

3.1 Arquitetura Proposta

A arquitetura abstrata proposta neste trabalho tem como objetivo o gerenciamento do dispositivo de IoT de leitura de tags RFID, assim como a disponibilização dos dados utilizando serviços providos de microsserviços e computação em nuvem. A arquitetura proposta é implementada de acordo com a arquitetura padrão em Internet das Coisas, conforme descrito na Seção 2.1. A Figura 3.1 é composta por cinco camadas: Camada de Percepção, Camada de Rede, Camada de *Middleware*, Camada de Aplicação e Camada de Negócio.

1. **Camada de Percepção:** responsável pela identificação e coleta de informações do dispositivo de IoT de leitura de tags RFID;
2. **Camada de Rede:** responsável pela transmissão dos dados gerados pelo dispositivo de IoT e sua comunicação com um servidor local;
3. **Camada de *Middleware*:** responsável pelo processamento, gerenciamento e armazenamento dos dados gerados pelo dispositivo de IoT;

4. **Camada de Aplicação:** responsável pela disponibilização dos dados gerados pelo dispositivo de IoT;
5. **Camada de Negócio:** responsável pela apresentação dos dados gerados pelo dispositivo de IoT para interação com o usuário.

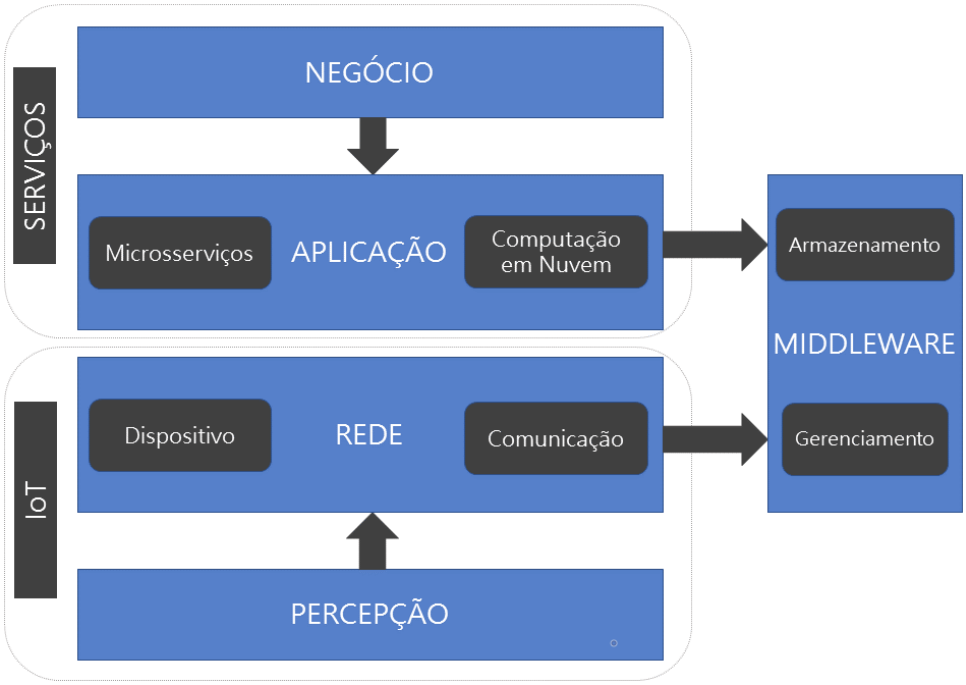

Figura 3.1: Arquitetura abstrata da solução proposta.

A Camada de Percepção consiste em identificar e coletar as informações do dispositivo de IoT de leitura de tags RFID. O dispositivo IoT possui dois tipos de eventos em sua comunicação: evento de leitura e evento de log. O evento de leitura ocorre quando a leitura de uma tag é realizada, já o evento de log, quando ocorrer algum erro na leitura de uma tag. Um terceiro evento é responsável pelo monitoramento da telemetria do dispositivo IoT. O evento de monitoramento é responsável por enviar informações sobre o estado do dispositivo IoT. A cada evento realizado pelo dispositivo IoT, os dados gerados são processados na Camada de Rede.

A Camada de Rede consiste em transmitir os dados gerados pelo dispositivo de IoT, além de realizar a comunicação com um servidor local. Os dados gerados nos eventos de leitura e de log realizados no dispositivo IoT, além do evento de monitoramento da telemetria, são enviados para a Camada de *Middleware*.

A Camada de *Middleware* consiste no processamento, gerenciamento e armazenamento dos dados gerados pelo dispositivo de IoT. Os dados gerados pelo dispositivo de IoT no evento de leitura e de log, além do evento de monitoramento da telemetria, são processados

e armazenados em um banco de dados *Not Only SQL* (NoSQL) pelo grande volume de informações que podem ser geradas, assim como pela necessidade de alta disponibilidade e escalabilidade da arquitetura.

A Camada de Aplicação consiste em disponibilizar serviços utilizando o padrão de microsserviços com a utilização de computação em nuvem. O uso de microsserviços e computação em nuvem possuem características que são essenciais para o gerenciamento de grandes arquiteturas em Internet das Coisas. Esta camada expõe as informações armazenadas na Camada de *Middleware*, para que possam ser utilizadas na Camada de Negócio.

A Camada de Negócio consiste na visualização das informações providas pelos serviços da Camada de Aplicação que expõem as informações do dispositivo de IoT.

A arquitetura proposta, além de ser representada por cinco camadas, é dividida em dois grandes grupos: Grupo de Internet das Coisas e Grupo de Serviços.

1. **Grupo de Internet das Coisas:** composto pela Camada de Percepção e Camada de Rede;
2. **Grupo de Serviços:** composto pela Camada de Aplicação e Camada de Negócio.

A Camada de *Middleware* é compartilhada entre os dois grupos da arquitetura. A separação arquitetural em camadas e em grupos se faz necessária para melhor utilização do hardware, descentralização do gerenciamento e armazenamento dos dados, e principalmente pelo processamento dos dados gerados pelo dispositivo de IoT.

As Seções seguintes trazem uma visão detalhada do funcionamento da arquitetura. Os grupos presentes na arquitetura são detalhados dentro das camadas da arquitetura.

3.2 Grupo de Internet das Coisas

O Grupo de Internet das Coisas é composto pela Camada de Percepção e Camada de Rede, além de ser responsável por enviar as informações geradas pelo dispositivo de IoT para a Camada de *Middleware*. Para realizar a leitura de tags RFID, a arquitetura proposta utiliza a placa *SparkFun Simultaneous RFID Reader - M6E Nano*[1], apresentada na Figura 3.2, como dispositivo de IoT de leitura de tags RFID.

A escolha desse dispositivo deve-se pelos seguintes fatores: baixo custo de mercado em relação aos leitores de tags RFID convencionais que utilizam a frequência UHF, possibilidade de utilizar antena interna presente na placa ou antena externa a ser acoplada e integração com SDK presente na placa, sendo possível utilizar as linguagens de programação C, C# ou Java.

[1] *SparkFun Simultaneous RFID Reader - M6E*

A placa *SparkFun Simultaneous RFID Reader - M6E Nano* realiza a função da Camada de Percepção, com a leitura de tags RFID a partir de sua antena interna localizada na parte superior da placa. Neste trabalho é utilizada a etiqueta passiva e a frequência UHF, respectivamente, dado que o custo é menor do que em outros tipos de tags, e ao alcance de distância de leitura das tags RFID é mais alto.

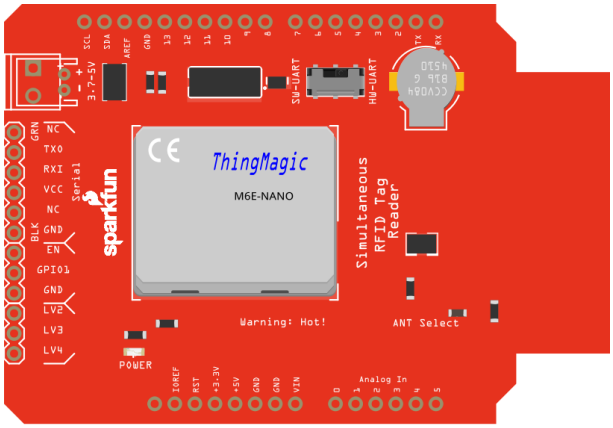

Figura 3.2: *SparkFun Simultaneous RFID Reader - M6E Nano.*

A Figura 3.3 apresenta a arquitetura do Grupo de Internet das Coisas responsável pela leitura e comunicação do dispositivo de IoT, seu processamento em relação aos eventos de leitura e log, além do evento de monitoramento da telemetria e do armazenamento dos dados em um banco de dados NoSQL utilizando um ambiente de computação em nuvem.

A comunicação da placa é realizada com um adaptador para a comunicação serial utilizando um *hardware* Receptor/Transmissor Assíncrono Universal (UART). A comunicação serial é uma das comunicações disponíveis na placa. Esta comunicação da placa realiza a função da Camada de Rede. Esse tipo de comunicação é necessária para não envolver outro tipo de *hardware* como um microcontrolador ou microcomputador para comunicação com a placa. Com a comunicação serial é possível se conectar com qualquer um desses *hardware* de maneira transparente. A comunicação é realizada em um servidor hospedado localmente, assim, é possível se conectar utilizando uma porta *Universal Serial Bus* (USB) do servidor local. Esta conexão poderia ser realizada em um microcontrolador ou microcomputador da mesma forma, ou, ainda, utilizando outro tipo de comunicação presente na Camada de Rede.

O servidor local é responsável pelo processamento dos dados gerados pela placa. O servidor local utilizado possui as seguintes configurações:

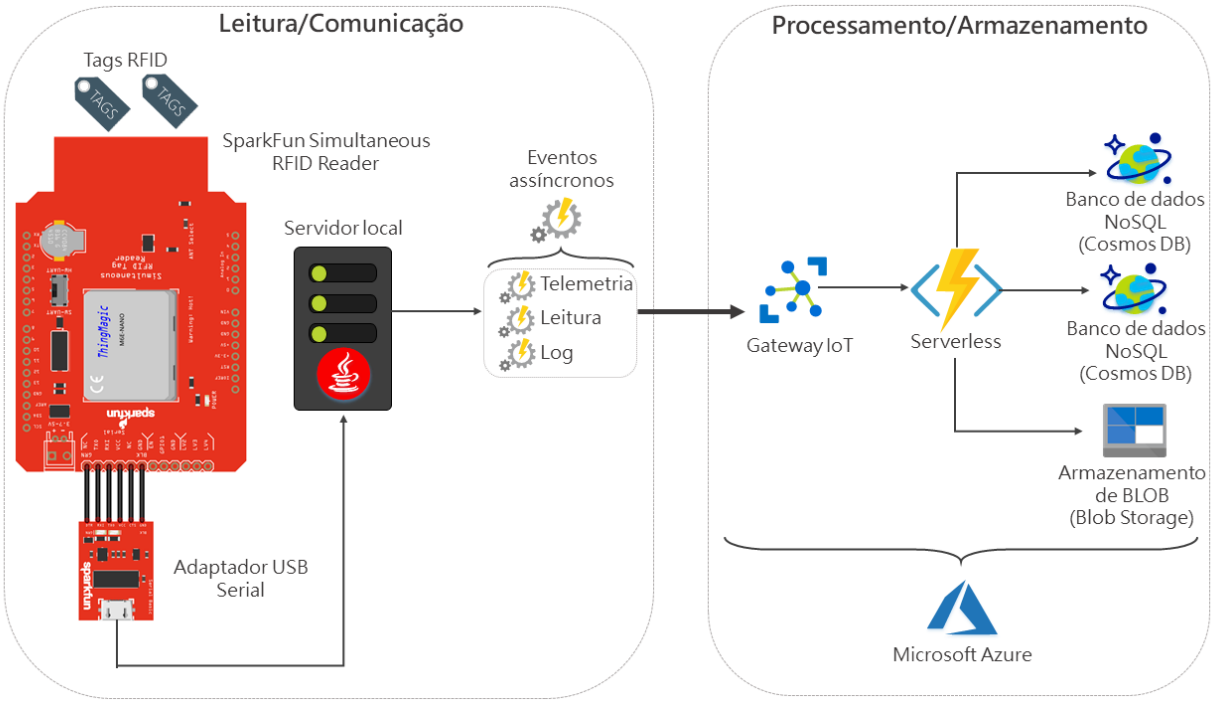

Figura 3.3: Arquitetura do Grupo de Internet das Coisas.

- **Sistema Operacional:** *Windows* 10;
- **Memória RAM**: 16 *Gigabytes*;
- **Processador:** Intel Core(TM) i7 6500U/2.50Giga-hertz (GHz);
- **Placa de Vídeo:** 4 *Gigabytes*;
- **Disco Rígido:** SSD Samsung 500 *Gigabytes*/520 *Megabytes* (escrita e leitura).

O servidor local possui um software desenvolvido na linguagem de programação Java. A utilização da linguagem de programação Java deve-se à interoperabilidade da linguagem em qualquer Sistema Operacional (SO). Nesse contexto, o servidor local utilizado poderia ter qualquer SO. Esse software é responsável por processar as informações dos eventos de leitura e log que ocorrem na placa, além de realizar o monitoramento da telemetria.

Os eventos de leitura e log realizados pela placa, além do monitoramento da telemetria, são processados de maneira assíncrona, dessa forma, nenhum evento necessita esperar algum evento finalizar para continuar. Os eventos podem ocorrer de forma simultânea, sem que ocorra algum gargalo no processamento das requisições dos eventos.

O processamento dos eventos de leitura, log e monitoramento da telemetria da placa são enviados no formato de arquivo Notação de Objetos JavaScript (JSON) para o *Gateway* de IoT *IoT Hub*, serviço oferecido pelo *Microsoft Azure*. O *IoT Hub* é um serviço

que permite o consumo de grandes volumes de dados de dispositivos IoT, além de todo o gerenciamento de dispositivos pela nuvem. O *IoT Hub* é um serviço compatível com as principais linguagens de programação e protocolos de comunicação. Uma das linguagens de programação oferecidas no serviço é a linguagem Java, implementada no servidor local para processamento dos eventos da arquitetura. O *IoT Hub* realiza a função da Camada de *Middleware*, sendo responsável pelo processamento e armazenamento dos dados gerados pela placa.

O Apêndice A.1 apresenta a implementação do evento de monitoramento da telemetria realizado pela placa RFID e o Apêndice A.2 apresenta a implementação dos eventos de leitura e log realizados pela placa RFID.

As seguintes informações de telemetria são enviadas para o *IoT Hub* nesta arquitetura: temperatura, localização e conectividade intermitente. O monitoramento da temperatura é importante para acompanhar o aquecimento da placa em possíveis picos de leitura de tags RFID ou para verificar algum aquecimento excessivo na placa a qualquer momento. A localização é importante para que placa possa ser monitorada e gerenciada de forma remota. A verificação de conectividade intermitente da placa é a função mais importante nesse tipo de serviço, sendo possível saber se a placa possui algum comportamento não esperado e se a placa encontra-se em funcionamento constante. Além do monitoramento da telemetria geradas pela placa, a utilização desse tipo de serviço é importante para a alta disponibilidade e escalabilidade da arquitetura, onde outros dispositivos podem ser conectados à arquitetura e serem monitorados e gerenciados.

O armazenamento dos dados gerados dos eventos de leitura, log e monitoramento da telemetria da placa ocorrem quando alguma dessas informações são enviadas para o *IoT Hub*. Um serviço da *Microsoft Azure* que utiliza a arquitetura de *Serverless*, *Azure Functions* é responsável por "escutar"os dados enviados para o *IoT Hub*. O *Azure Functions* está implementado na linguagem de programação C#. O *Azure Functions* fica hospedada no *Microsoft Azure* em *background*, sendo acionada apenas quando há alguma informação enviada para o *IoT Hub*.

O Apêndice A.3 apresenta a implementação do *Azure Functions*. Quando o *Azure Functions* é acionado, o mesmo verifica qual tipo de evento foi invocado: leitura ou telemetria, que são realizados pela placa, ou log. Cada um desses eventos são armazenados em seus respectivos repositórios definidos na arquitetura.

A utilização desse tipo de arquitetura *Serverless* se faz necessária pelo contexto de funcionamento em *background* desse tipo de serviço na arquitetura, além de ser importante para resiliência no envio dos dados, tornando a arquitetura escalonável e menos sensível a falhas individuais de outros componentes da arquitetura.

O banco de dados NoSQL utilizado para armazenamento dos eventos de leitura e

monitoramento da telemetria da placa é um serviço do *Microsoft Azure*, *Cosmos DB*. O *Cosmos DB* é um banco de dados multimodelo globalmente distribuído que dá suporte a bancos de dados de documentos, valor-chave, coluna e grafo. Neste trabalho é utilizado o banco de dados orientado a documentos. Esse tipo de banco de dados foi escolhido por ser ideal para obter baixa latência, alta disponibilidade e por possuir um esquema de dados flexível.

As informações de log geradas são armazenadas em um *Blob Storage*, serviço oferecido pelo *Microsoft Azure* e específico para armazenamento do tipo de arquivo *Binary Large Object* (BLOB). O tipo de arquivo BLOB é uma coleção de dados binários que são armazenados como uma única entidade. É uma coleção de dados ideal para armazenar qualquer tipo de dado não estruturado. Caso aconteça alguma mudança no formato do arquivo JSON do log, não é necessário nenhuma adequação no armazenamento desse tipo de arquivo.

Os dados armazenados dos eventos de leitura e monitoramento da telemetria da placa no *Cosmos DB* e os dados de log armazenadas no *Blob Storage* são acessados pelo Grupo de Serviços, responsáveis pela disponibilização dessas informações utilizando microsserviços e computação em nuvem.

3.3 Grupo de Serviços

O Grupo de Serviços é composto pela Camada de Aplicação e Camada de Negócio, além de ser responsável por se comunicar com as informações armazenadas na Camada de *Middleware*. A Camada de Aplicação consiste em disponibilizar serviços providos de microsserviços e computação em nuvem. Esses serviços são consumidos na Camada de Negócio para exposição das informações. A Figura 3.4 apresenta a arquitetura do Grupo de Serviços utilizando microsserviços em um ambiente de computação em nuvem.

O Grupo de Serviços é composto por cinco microsserviços: microsserviço de Ativo, microsserviço de Leitura, microsserviço de Telemetria, microsserviço de Log e microsserviço de Identidade. Os microsserviços foram implementados utilizando o *framework* Web *ASP.NET Core*, na linguagem de programação C#. Cada Interface de Programação de Aplicativos (API) do microsserviço é implementado utilizando o estilo arquitetural Representação de Transferência de Estado (REST) com o retorno da resposta dos *endpoints* no formato JSON.

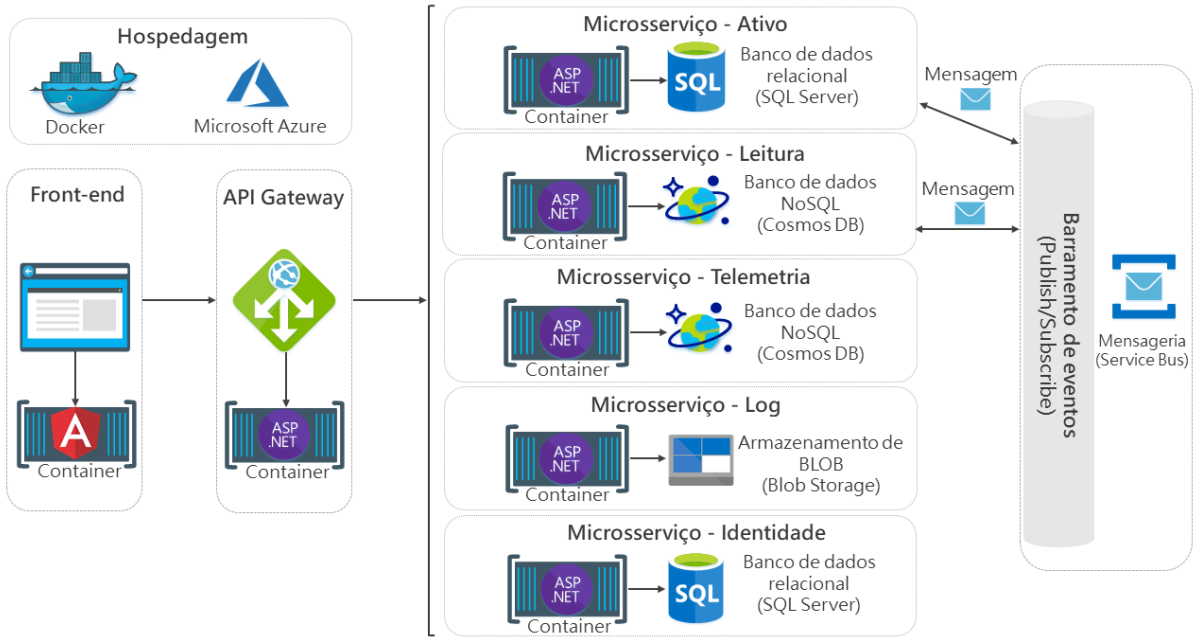

Figura 3.4: Arquitetura do Grupo de Serviços.

3.3.1 Microsserviço de Ativo

O microsserviço de Ativo é responsável por disponibilizar as informações de objetos físicos que são identificados com uma tag RFID. Os dados do microsserviço de Ativo são armazenados no banco de dados *SQL Server*, hospedado no *Microsoft Azure*. Neste trabalho é utilizado um banco de dados relacional, por ser um tipo de banco de dados comum para armazenamento de informações. O armazenamento dos dados poderia utilizar qualquer outro tipo de banco de dados. A Figura 3.5 apresenta o modelo conceitual do banco de dados do microsserviço de Ativos que foi criado a partir do mapeamento das principais informações necessárias para identificação de um objeto físico.

O modelo conceitual do banco de dados é composto por três entidades físicas:

- **Ativo:** Representa o objeto físico identificado com uma tag RFID;
- **Local:** Representa a localidade do objeto físico. Podendo ser uma garagem, portaria, escritório, entre outras localidades;
- **Tipo:** Representa o tipo do objeto físico. Podendo ser um equipamento eletrônico, um móvel físico, entre outros tipos.

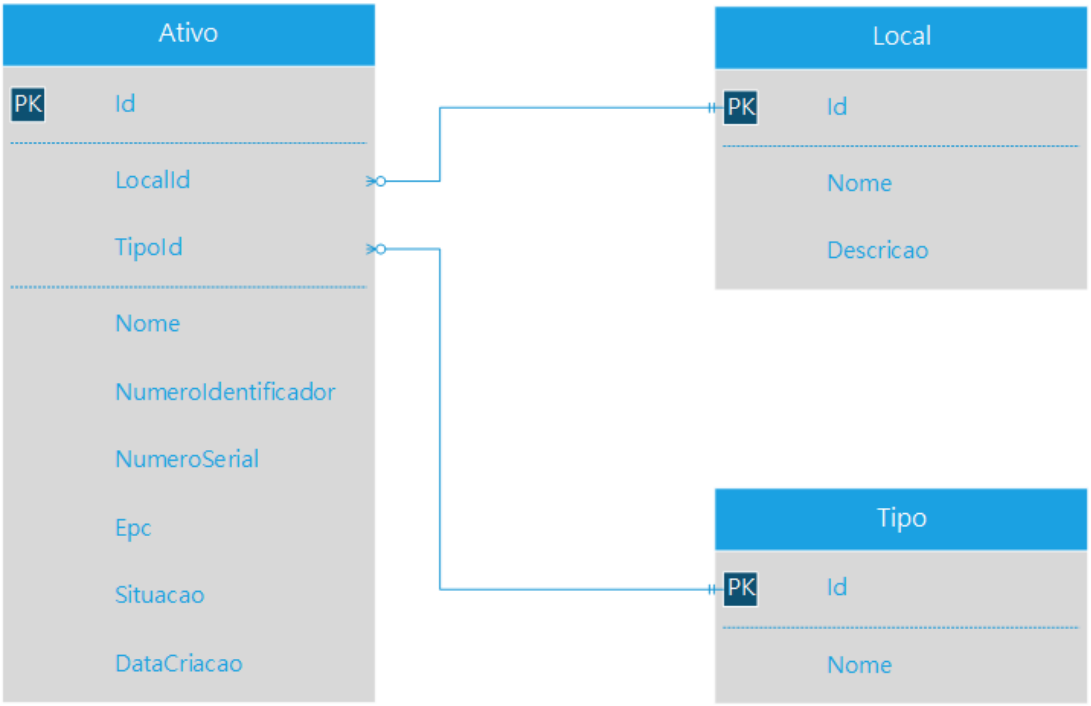

Figura 3.5: Modelo conceitual do microsserviço de Ativo.

As Tabelas 3.1, 3.2 e 3.3 apresentam as descrição dos atributos e dos tipos de dados presentes no modelo conceitual do banco de dados microsserviço de Ativo.

Tabela 3.1: Descrição dos atributos e tipo de dados da entidade Ativo.

Atributo	**Tipo**	**Descrição**
Id	Numérico	Identificador da entidade.
LocalId	Numérico	Identificador de relacionamento com a entidade Local.
TipoId	Numérico	Identificador de relacionamento com a entidade Tipo.
Nome	Texto	Nome do objeto físico vinculado com a tag RFID.
NumeroIdentificador	Texto	Número de identificação do objeto físico.
NumeroSerial	Texto	Número de série de acordo com o fabricante do objeto físico.
Epc	Texto	Código único da tag RFID vinculada ao objeto físico.
Situacao	Booleano	Apresenta se o objeto físico ainda está sendo monitorado.
DataCriacao	Data	Data de criação do vínculo do objeto físico com a tag RFID.

Tabela 3.2: Descrição dos atributos e tipo de dados da entidade Local.

Atributo	Tipo	Descrição
Id	Numérico	Identificador da entidade.
Nome	Texto	Nome do local onde o objeto físico foi vinculado.
Descricao	Texto	Descrição do local onde o objeto físico foi vinculado.

Tabela 3.3: Descrição dos atributos e tipo de dados da entidade Tipo.

Atributo	Tipo	Descrição
Id	Numérico	Identificador da entidade.
Nome	Texto	Nome do local onde o objeto físico foi vinculado.

O código fonte 3.1 apresenta o retorno em JSON do objeto físico identificado pela placa RFID presente no microsserviço de Ativo. O atributo *"id"* representa o identificador único de leitura de um objeto físico, o atributo *"ativo"* representa o nome do objeto físico identificado, o atributo *"tipo"* representa o tipo do objeto físico identificado, o atributo *"local"* representa o local onde o objeto físico foi identificado, e por fim, o atributo *"epc"* representa o código único da tag RFID:

```
{
    "id": "72f9fcdb-1fe7-490b-93ed-c0ec61868167",
    "ativo": "Computador",
    "local": "Garagem",
    "tipo": "Eletronico",
    "epc": "494E44553030303030613133"
}
```

Código Fonte 3.1: JSON de resposta do microsserviço de Ativo.

3.3.2 Microsserviço de Leitura

O microsserviço de Leitura é responsável por disponibilizar as informações de leituras de tags RFID. O código fonte 3.2 apresenta o retorno em JSON de um objeto físico com tag RFID presente no microsserviço de Leitura. O atributo *"id"* representa o identificador único de leitura de um objeto físico, o atributo *"ip"* representa o número de protocolo de rede do servidor local, o atributo *"epc"* representa o código único da tag RFID, o atributo *"antena"* representa o número da antena da placa RFID que realizou a leitura, e por fim, o atributo *"dataLeitura"* representa a data de leitura que a tag RFID foi identificada:

```
{
    "id": "7cca1259-ed23-43a9-b5d2-11c475871df1",
    "ip": "10.0.75.1",
    "epc": "E2002083980201001070A87F",
    "dataLeitura": "2019-03-06T14:13:55.037",
    "antena": 1
}
```

Código Fonte 3.2: JSON de resposta do microsserviço de Leitura.

3.3.3 Microsserviço de Telemetria

O microsserviço de Telemetria é responsável por disponibilizar as informações de telemetria da placa *SparkFun Simultaneous RFID Reader - M6E Nano*. Conforme apresentado na arquitetura do Grupo de Internet das Coisas, os dados desses microsserviços são armazenados no banco de dados NoSQL *Cosmos DB* e hospedado no *Microsoft Azure*.

O código fonte 3.3 apresenta o retorno em JSON do monitoramento da telemetria da placa RFID presente no microsserviço de Telemetria. O atributo *"id"* representa o identificador único de leitura de um objeto físico, o atributo *"ip"* representa o número de protocolo de rede do servidor local, o atributo *"temperatura"* representa a temperatura da placa RFID, e por fim, o atributo *"conexao"* representa o estado de conexão da placa RFID:

```
{
    "id": "4a31fc92-9780-4e9d-9c0f-aac2faa3d085",
    "ip": "10.0.75.1",
    "temperatura": 45,
    "conexao": true
}
```

Código Fonte 3.3: JSON de resposta do microsserviço de Telemetria.

3.3.4 Microsserviço de Log

O microsserviço de Log é responsável por disponibilizar as informações de erros que podem ocorrer na leitura de tags RFID. Conforme apresentado na arquitetura do Grupo de Internet das Coisas, os dados desse microsserviço são armazenados em um *Blob Storage*, hospedado no *Microsoft Azure* no formato de arquivo BLOB.

O código fonte 3.4 apresenta o retorno em JSON do log da placa RFID presente no microsserviço de Log. O atributo *"id"* representa o identificador único de leitura de um objeto físico, o atributo *"ip"* representa o número de protocolo de rede do servidor local, o atributo *"excecao"* representa o nome da exceção ocorrida na placa RFID, o atributo

"dataExcecao" representa a data que ocorreu o erro na placa RFID, e por fim, o atributo *"stackTrace"* representa a pilha de erros ocorrida exceção:

```
{
    "id": "7cca1259-ed23-43a9-b5d2-11c475871df",
    "ip": "10.0.75.1",
    "Excecao": "Timeout",
    "dataExcecao": "2019-03-06T14:13:55.037",
    "stackTrace": "[com.thingmagic.SerialTransportNative.nativeSendBytes(
    Native Method), com.thingmagic.SerialTransportNative.sendBytes(
    SerialTransportNative.java:210), com.thingmagic.SerialReader.sendMessage
    (SerialReader.java:2159), com.thingmagic.SerialReader.sendTimeout(
    SerialReader.java:2326), com.thingmagic.SerialReader.sendOpcode(
    SerialReader.java:2384), com.thingmagic.SerialReader.cmdClearTagBuffer(
    SerialReader.java:4420), com.thingmagic.SerialReader.read(SerialReader.
    java:10890), com.rfid.reader.services.ReaderService.readDataTag(
    ReaderService.java:20), com.rfid.reader.events.ReaderMessageSender.run(
    ReaderMessageSender.java:26), java.util.concurrent.ThreadPoolExecutor.
    runWorker(ThreadPoolExecutor.java:1142), java.util.concurrent.
    ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617), java.lang.
    Thread.run(Thread.java:745)]"
}
```

Código Fonte 3.4: JSON de resposta do microsserviço de Log.

3.3.5 Microsserviço de Identidade

O microsserviço de Identidade é responsável por realizar a autenticação e a autorização de todos os microsserviços presentes na arquitetura: microsserviço de Ativo, microsserviço de Leitura, microsserviço de Telemetria e microsserviço de Log. A autenticação e a autorização é realizada utilizando o protocolo *OAuth 2.0*, onde é gerado um *token* no padrão JWT para aplicação. A implementação da autenticação e autorização dentro do microsserviço é realizada pela biblioteca de código aberto *IdentityServer4*.

A *API Gateway* é responsável por gerenciar a autorização dos *endpoints* dos microsserviços que são disponibilizados. O Apêndice A.4 apresenta a implementação realizada na classe inicial de configuração da *API Gateway*. Quando um *endpoint* é solicitado, é verificado utilizando a biblioteca do *IdentityServer4* se o cliente que realizou a solicitação está autenticado e autorizado para acessar o *endpoints* solicitado.

3.3.6 Barramento de Eventos

A comunicação entre os microsserviços é realizada com o barramento de eventos através de mensagens utilizando o padrão de projeto *Publish-subscribe* [14] [34]. O gerenciamento do barramento de eventos e o envio desses eventos de mensagens é controlada pelo serviço do *Microsoft Azure*, *Service Bus*, implementada no microsserviço de Ativo e no microsserviço de Leitura. A Figura 3.6 apresenta com detalhes como o barramento de eventos é utilizado. O microsserviço de Ativo comunica-se via evento de mensagem com o microsserviço de Leitura, que notifica via evento de mensagem caso há alguma leitura de tag RFID associada a algum objeto físico que esteja vinculado ao banco de dados do microsserviço de Ativo.

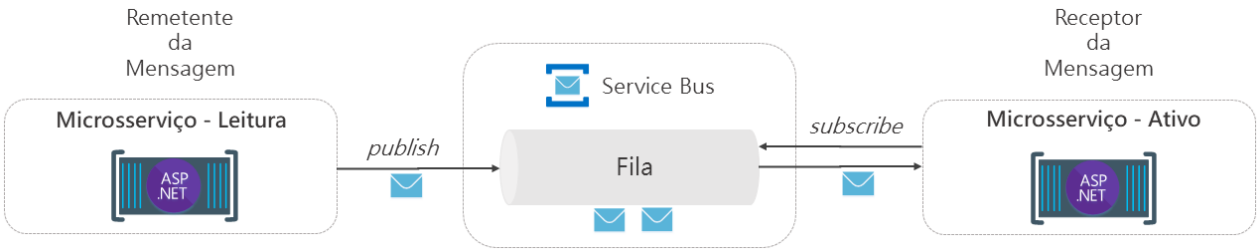

Figura 3.6: Comunicação entre os microsserviços de Ativo e Leitura.

A identificação ocorre quando o código EPC da tag RFID comparado com os dados presentes no banco de dados do microsserviço de Ativo resulta em algum objeto físico identificado. Os outros microsserviços não possuem a necessidade de se comunicar via barramento de eventos por não possuirem dependência de informações. Por esse motivo, não utilizam do barramento de eventos.

O código fonte 3.5 apresenta a consulta SQL realizada no banco de dados do microsserviço de Ativo para verificar se o objeto físico identificado no microsserviço de Leitura via barramento de eventos está vinculado a algum objeto físico. O campo *@Epc* identifica onde é realizado o filtro do código EPC da tag RFID identificado no barramento de eventos:

```
1 SELECT a.Nome, l.Nome AS Local, t.Nome AS Tipo, a.Epc
2 FROM dbo.Ativo a
3 INNER JOIN dbo.Local l
4 ON a.LocalId = l.Id
5 INNER JOIN dbo.Tipo t
6 ON a.TipoId = t.Id
7 WHERE a.Epc = @Epc
```

Código Fonte 3.5: Consulta SQL no microsserviço de Ativo.

3.3.7 *API Gateway*

A comunicação dos microsserviços com a aplicação *front-end* é realizada utilizando uma *API Gateway*. A *API Gateway* é responsável por gerenciar e monitorar os microsserviços a serem expostos para serem consumidos na aplicação *front-end*. Na arquitetura proposta, todos os microsserviços são expostos para serem consumidos na aplicação *front-end*. A *API Gateway* foi implementada utilizando o *framework* Web *ASP.NET Core*, na linguagem de programação C#. A utilização de uma *API Gateway* na arquitetura em microsserviços é importante para disponibilidade e escalabilidade da arquitetura, onde outros microsserviços podem ser conectados à arquitetura e serem monitorados e gerenciados.

O Apêndice A.5 apresenta a configuração dos *endpoints* dos microsserviços implementados na arquitetura. A implementação da *API Gateway* é realizada pela biblioteca de código aberto *Ocelot*. Todos os *endpoints* disponibilizados possuem a autorização realizada no microsserviço de Identidade, pelo único *endpoint* sem autorização, que é responsável por disponibilizar o *token* de autenticação.

3.3.8 Aplicação *Front-end*

A aplicação *front-end* realiza o papel da Camada de Negócio, expondo os microsserviços implementados na Camada de Aplicação. A aplicação *front-end* utiliza a *API Gateway* para consumir os microsserviços implementados. A aplicação *front-end* foi implementada utilizando o *framework* Web *Angular*, na linguagem de programação TypeScript.

A Figura 3.7 apresenta a tela com o resultado da identificação de ativos disponibilizado pelo microsserviço de Ativo e a comunicação via barramento de eventos com o microsserviço de Leitura. A tela apresenta o total de ativos identificados e a listagem dos ativos de acordo com o arquivo JSON apresentado anteriormente.

A Figura 3.8 apresenta a tela com os resultados de leituras de tags RFID disponibilizado pelo microsserviço de Leitura. A tela apresenta o total de leituras realizadas e a listagem das leituras de acordo com o arquivo JSON apresentado anteriormente.

A Figura 3.9 apresenta a tela com os resultados de telemetrias da placa disponibilizado pelo microsserviço de Telemetria. A tela apresenta o total de telemetrias realizadas e a listagem das telemetrias de acordo com o arquivo JSON apresentado anteriormente.

A Figura 3.10 apresenta a tela com os logs gerados pela placa disponibilizado pelo microsserviço de Log. A tela apresenta o total de logs realizados e a listagem dos logs de acordo com o arquivo JSON apresentado anteriormente.

Ativos: 10

Ativo	Local	Tipo	EPC
Mesa	Garagem	Mobiliário	414953443030303030303032
Mesa	Garagem	Mobiliário	414953443030303030303032
Cadeira	Recepção	Mobiliário	E2002083980201001070A87F
Computador	Garagem	Eletrônico	494E44553030303030613133
Cadeira	Recepção	Mobiliário	E2002083980201001070A87F
Mesa	Garagem	Mobiliário	414953443030303030303032
Computador	Garagem	Eletrônico	494E44553030303030613133
Cadeira	Recepção	Mobiliário	E2002083980201001070A87F
Computador	Garagem	Eletrônico	494E44553030303030613133
Cadeira	Recepção	Mobiliário	E2002083980201001070A87F

Figura 3.7: Resultado de identificação de ativos.

Leituras: 100

IP	EPC	Data	Antena
10.0.75.1	42414E303030303336343631	20/5/19 - 7:20:47	1
10.0.75.1	42414E303030303336343631	20/5/19 - 7:20:44	1
10.0.75.1	494E44303030303030613161	20/5/19 - 7:14:24	1
10.0.75.1	494E44553030303030613133	20/5/19 - 6:40:13	1
10.0.75.1	494E44553030303030613133	20/5/19 - 6:40:08	1
10.0.75.1	E2002083980201001070A87F	20/5/19 - 6:40:02	1
10.0.75.1	E2002083980201001070A87F	20/5/19 - 6:39:44	1
10.0.75.1	E2002083980201001070A87F	20/5/19 - 6:39:42	1
10.0.75.1	494E44303030303030613161	20/5/19 - 6:39:35	1
10.0.75.1	494E44303030303030613161	20/5/19 - 6:39:31	1

Figura 3.8: Resultado de leituras de tags RFID.

Telemetrias: 100

IP	Temperatura	Conexão
10.0.75.1	37°C	Sim
10.0.75.1	37°C	Sim
10.0.75.1	37°C	Sim
10.0.75.1	37°C	Sim
10.0.75.1	37°C	Sim
10.0.75.1	36°C	Sim
10.0.75.1	38°C	Sim
10.0.75.1	36°C	Sim
10.0.75.1	36°C	Sim
10.0.75.1	36°C	Sim

Figura 3.9: Resultado de telemetrias na placa RFID.

Logs: 1

IP	Exceção	Data	StackTrace
10.0.75.1	Timeout	20/5/19 - 7:20:52 PM	[com.thingmagic.SerialTransportNative.nativeSendBytes(Native Method), com.thingmagic.SerialTransportNative.sendBytes(SerialTransportNative.java:210), com.thingmagic.SerialReader.sendMessage(SerialReader.java:2159), com.thingmagic.SerialReader.sendTimeout(SerialReader.java:2326), com.thingmagic.SerialReader.sendOpcode(SerialReader.java:2384), com.thingmagic.SerialReader.cmdClearTagBuffer(SerialReader.java:4420), com.thingmagic.SerialReader.read(SerialReader.java:10890), com.rfid.reader.services.ReaderService.readDataTag(ReaderService.java:20), com.rfid.reader.events.ReaderMessageSender.run(ReaderMessageSender.java:26), java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142), java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617), java.lang.Thread.run(Thread.java:745)]

Figura 3.10: Resultado de logs de leitura de tags RFID.

Os microsserviços, a *API Gateway* e a aplicação *front-end* são provisionados em *containers* utilizando a tecnologia *Docker*, hospedadas no *Microsoft Azure*. As aplicações são provisionadas em *containers Docker* utilizando o SO Linux. A utilização da tecnologia de *containers Docker* facilita a integração entre microsserviços e a computação em nuvem [37], [40].

A utilização da tecnologia de *containers* facilita uma possível mudança da aplicação para outro provedor de nuvem que possua suporte para *containers Docker*, além de ser

possível implementar os microsserviços, a *API Gateway* e a aplicação *front-end* em qualquer outra linguagem de programação que ofereça o suporte pra construção de API.

O Apêndice A.6 apresenta a configuração da imagem *Docker* do microsserviço de Ativo. A mesma configuração da imagem contempla os outros microsserviços e a *API Gateway* presente do Grupo de Serviços. O Apêndice A.7 apresenta a configuração da imagem *Docker* do aplicação *front-end*.

3.4 Arquitetura de Integração e Entrega Contínua

Dragoni et al. [14] afirma que para aproveitar verdadeiramente o poder da implantação independente dos microsserviços é preciso utilizar mecanismos de integração e entrega contínua. A integração e integra contínua estão associadas com a cultura de *DevOps*.

A arquitetura de serviços e de IoT com a utilização de *Serverless* na arquitetura proposta deste trabalho são fundamentais para esse tipo de cenário. Os *pipelines* automatizados de integração e entrega contínua são provisionados em *containers* utilizando a tecnologia *Docker*, é possível implantar uma versão atualizada de um serviço para produção em questão de segundos, além de ser possível realizar testes em todos os microsserviços e principalmente os que se comunicam, garantindo assim, uma maior confiabilidade quando uma nova versão dos serviços é atualizada.

A Figura 3.11 apresenta a arquitetura de integração e entrega contínua aplicada na arquitetura proposta. O *Azure DevOps* é a ferramenta responsável por realizar os *pipelines* de integração contínua e entrega contínua da arquitetura de serviços (microsserviços, *API Gateway* e aplicação *front-end*) e arquitetura de Internet das Coisas (*Serverless*). A arquitetura proposta deste trabalho possui o *pipeline* dos microsserviços, serviço *Serverless* e *API Gateway* e o *pipeline* da aplicação *front-end*.

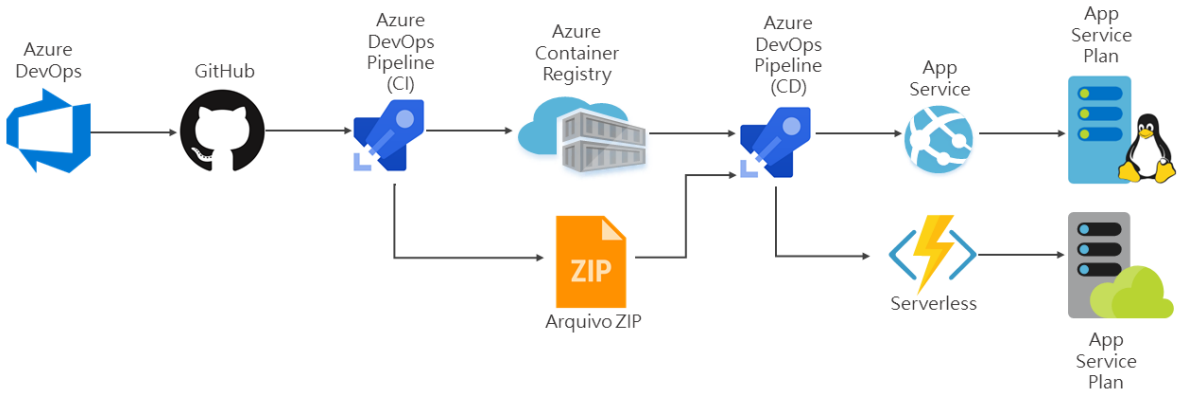

Figura 3.11: Arquitetura de integração contínua e entrega contínua da arquitetura proposta.

Os códigos fontes dos projetos estão armazenados no GitHub nos repositórios informados no Apêndice A. O GitHub é integrado com o *Azure DevOps* que o informa quando há algum novo *commit* na *branch master* dos projetos. Quando esse novo *commit* acontece, o *Azure DevOps* é acionado para realizar a integração contínua.

O Apêndice A.8 apresenta a configuração das imagens *Docker* dos microsserviços e da *API Gateway* implementados no arquivo *docker-compose.yml*. Esse arquivo é responsável por ser o orquestrador das imagens dos microsserviços implementados para realização dos *pipelines* de integração contínua e entrega contínua dos microsserviços.

O Apêndice A.9 apresenta a configuração do *pipeline* de integração contínua dos microsserviços, *API Gateway* e serviço *Serverless* presente no arquivo *azure-pipelines.yml*. Esse arquivo é executado no momento que o *pipeline* de integração contínua se inicia para o *pipeline* dos microsserviços, serviço *Serverless* e *API Gateway*. O *pipeline* de integração contínua realiza a integração de acordo com os serviços definidos no arquivo *docker-compose.yml* e com os arquivos *Docker* dos microsserviços e *API Gateway* apresentados no Capítulo 3.

O Apêndice A.10 apresenta a configuração do pipeline de integração contínua da aplicação *front-end*. Esse arquivo é executado no momento que o pipeline de integração contínua se inicia para o pipeline da aplicação *front-end*. O pipeline de integração contínua realiza a integração de acordo o arquivo *Docker* da aplicação *front-end*.

Após a realização da integração contínua dos *pipelines* de acordo com a implementação independente de cada *pipeline*, as imagens *Docker* são armazenadas no *Azure Container Registry*, serviço do *Microsoft Azure* responsável por armazenar e gerenciar imagens de contêiner *Docker*. O código do serviço *Serverless* é armazenado em um arquivo com o formato *zip*.

A entrega contínua possui a finalidade de publicar os microsserviços, *API Gateway*, serviço *Serverless* e a aplicação *back-end* no *Microsoft Azure*. Os microsserviços, *API Gateway* e a aplicação *back-end* são publicados em um *App Service*, serviço do *Microsoft Azure* utilizado para publicar aplicações Web onde cada aplicação se faz uso de *App Service Plan*, serviço do *Microsoft Azure* que publica as aplicações em um servidor. Neste trabalho, as aplicações são armazenadas em um servidor com SO Linux.

O serviço *Serverless* é publicado a partir do arquivo *zip* gerado no *Azure Functions* e armazenado em um servidor com SO Windows. Esse servidor é acionado quando o serviço *Serverless* é processado.

Capítulo 4

Estudo de Caso

De acordo com Runeson et al. [22] o estudo de caso é um estudo empírico para investigar o acontecimento de um fenômeno de engenharia de software contemporâneo dentro de seu contexto de vida real, principalmente quando a fronteira entre o fenômeno e o contexto não pode ser claramente especificada. A técnica de estudo de caso foi utilizada neste trabalho por ser uma técnica bastante utilizada em pesquisas qualitativas e capaz de reunir informações detalhadas a respeito de determinado problema.

O estudo de caso foi aplicado em uma empresa de pequeno à médio porte, denominada pelo nome fictício de Empresa XYZ. A Empresa XYZ atua na área de gestão patrimonial de ativos. A gestão patrimonial de ativos é realizada utilizando leitores de tags RFID móveis e antenas RFID expostas em alguns pontos estratégicos das empresas clientes. Esses pontos estratégicos podem ser garagens, portarias, salas confidenciais, etc. A Empresa XYZ possui clientes com atuação na área bancária, engenharia e hospitais, possuindo aproximadamente 100 mil ativos cadastrados em sua base de dados. A arquitetura proposta neste trabalho visa contemplar a gestão patrimonial utilizando antenas RFID.

A Figura 4.1 apresenta a arquitetura de IoT da Empresa XYZ. A arquitetura é compostas por dois grupos: leitura e processamento/armazenamento. Para leitura de tags RFID é utilizado o leitor de alto custo *Edge-50*[1]. O leitor *Edge-50* utiliza a frequência UHF, sendo necessário acoplar uma antena externa ao leitor.

A comunicação do leitor é realizada através do protocolo TCP, uma das comunicações presentes no leitor. A integração do leitor com o servidor local acontece por meio do SDK presente no leitor, sendo possível utilizar as linguagens de programação C, C# ou Java. O servidor local possui um software desenvolvido na linguagem de programação C#.

Esse software é responsável por processar as informações dos eventos de leitura e log que ocorrem no leitor. Os eventos de leitura e log realizados pelo leitor, são processados de

[1] *Edge-50*

maneira assíncrona, dessa forma, nenhum evento necessita esperar algum evento finalizar para continuar. Os eventos podem ocorrer de forma simultânea, sem que ocorra algum gargalo no processamento das requisições dos eventos. O processamento dos eventos de leitura e log do leitor são enviados no formato de arquivo JSON para o banco de dados NoSQL *Table Storage*, serviço oferecido pelo *Microsoft Azure*.

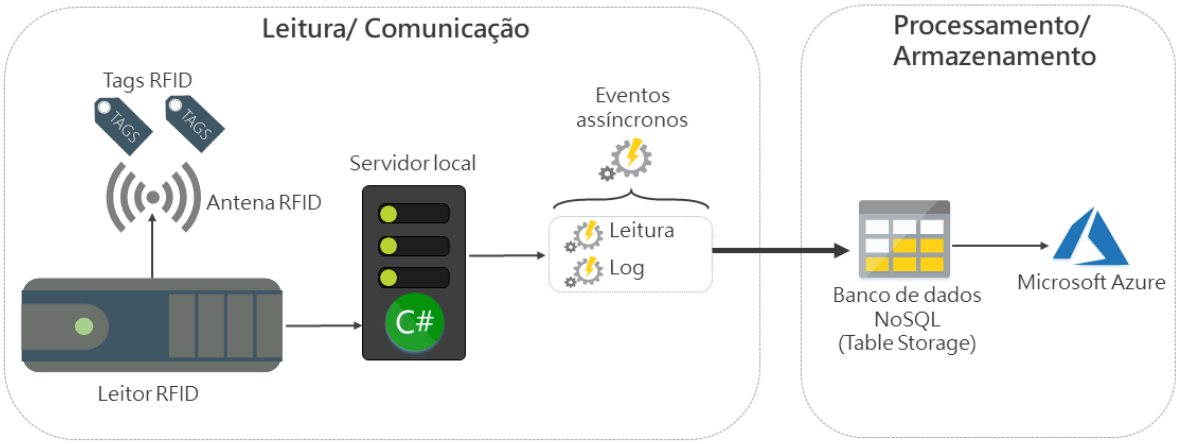

Figura 4.1: Arquitetura IoT da Empresa XYZ.

A Figura 4.2 apresenta a arquitetura de serviços da Empresa XYZ. A arquitetura foi desenvolvida no estilo arquitetural *multi tenant*, onde todos clientes da Empresa XYZ utilizam do mesmo *back-end* e *front-end*. Esse mesmo estilo arquitetural não é aplicado na arquitetura de IoT, onde todos os clientes possuem sua arquitetura conforme apresentado na Figura 4.1. A arquitetura de serviços é composta por dois grupos: *back-end* e *front-end*, ambos hospedados no *Microsoft Azure*.

O *back-end* da aplicação é uma API implementada utilizando o *framework* Web *ASP.NET MVC*, na linguagem de programação C#. A API foi implementada utilizando o estilo arquitetural REST com o retorno da resposta dos *endpoints* no formato JSON. O banco de dados relacional *SQL Server* e o banco de dados NoSQL *Table Storage* são utilizados na arquitetura *back-end*.

A arquitetura *back-end* apresenta uma aplicação monolítica, onde todos os *endpoints* disponibilizados estão no mesmo projeto, com nenhuma separação de domínios ou responsabilidades. A aplicação é responsável por autenticar, autorizar usuários e gerenciar a gestão patrimonial de ativos. Essas informações estão armazenadas no banco de dados relacional *SQL Server*. As informações dos eventos de leitura e log que ocorrem no leitor RFID são armazenadas no banco de dados NoSQL *Table Storage*.

O *front-end* da aplicação, assim como a API implementada no *back-end*, utiliza o *framework* Web *ASP.NET MVC*, na linguagem de programação C#. A aplicação *front-end* é responsável por expor os serviços implementados na aplicação *back-end*.

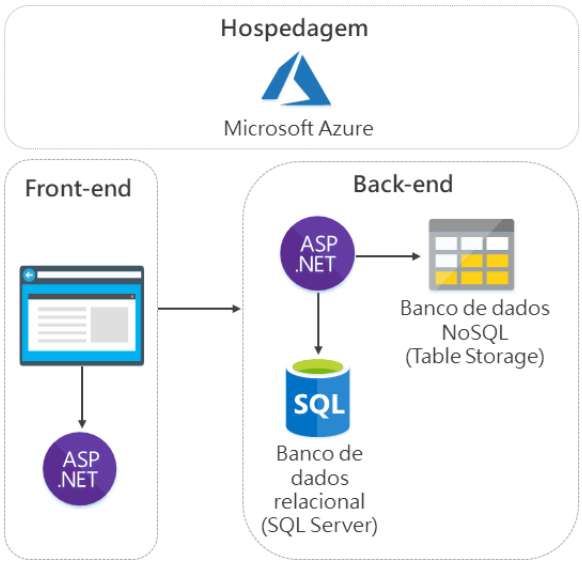

Figura 4.2: Arquitetura de serviços da Empresa XYZ.

O estudo de caso prático desenvolvido neste trabalho tem como objetivo responder às seguintes questões de pesquisa:

1. **RQ.1.** Na arquitetura proposta, quais são as vantagens da arquitetura de serviços e arquitetura de IoT em relação a apresentada na Empresa XYZ?
2. **RQ.2.** Na arquitetura proposta, quais são as principais características apresentadas na placa *SparkFun Simultaneous RFID Reader - M6E Nano* em relação ao leitor *Edge-50*?
3. **RQ.3.** Na arquitetura proposta, é possível realizar o monitoramento dos microsserviços implementados?

A **RQ.1.** está relacionada com a implementação da arquitetura de serviços e da arquitetura de IoT proposta no cenário real apresentado da Empresa XYZ. Pretende-se demonstrar como os microsserviços e o provedor de computação em nuvem *Microsoft Azure* facilitam a interoperabilidade, flexibilidade, escalabilidade e provisionamento rápido dos microsserviços, além de demonstrar com os serviços oferecidos pelo *Microsoft Azure* facilitam o desenvolvimento da arquitetura de IoT.

A **RQ.2.** está relacionada com a implementação da arquitetura de Internet das Coisas proposta no cenário real apresentado da Empresa XYZ. Pretende-se demonstrar as principais características da placa RFID *SparkFun Simultaneous RFID Reader - M6E Nano* em relação ao leitor utilizado na arquitetura da Empresa XYZ, *Edge-50*.

A **RQ.3.** está relacionada com a visualização do monitoramento dos microsserviços implementados na arquitetura proposta. Pretende-se demonstrar como a *API Gateway* e o provedor de computação em nuvem *Microsoft Azure* fornecem informações sobre o monitoramento dos microsserviços implementados.

4.1 Coleta de Dados e Procedimentos de Análise

Para realizar a coleta dos dados e realizar a análise dos resultados, toda a arquitetura proposta foi implementada para responder as questões de pesquisa destinadas ao estudo de caso deste trabalho. Um dos grandes problemas presentes na arquitetura de serviços da Empresa XYZ é a impossibilidade de escalar os serviços de forma independente, além de não possuir uma cultura de *DevOps* para automatizar a integração e entrega contínua dos serviços. Esses problemas apresentados são características comuns em arquiteturas monolíticas, conforme a arquitetura de serviços da Empresa XYZ apresentada na Figura 4.2.

A arquitetura de serviços e de IoT com a utilização de *Serverless* na arquitetura proposta neste trabalho em conjunto com a cultura de *DevOps*, são fundamentais para esse tipo de cenário. Os *pipelines* automatizados de integração e entrega contínua, quando provisionados em *containers* e utilizando a tecnologia *Docker*, torna possível a implementação de uma versão atualizada de um serviço para produção em questão de segundos, além de ser possível realizar testes em todos os microsserviços, principalmente nos que se comunicam, garantindo assim, uma maior confiabilidade quando uma nova versão dos serviços é atualizada. Para responder a **RQ.1.**, foi implementado a arquitetura de integração e entrega contínua conforme apresentada no Capítulo 3.

Para responder a **RQ.2.** do estudo de caso deste trabalho, foram analisadas as principais características da placa *SparkFun Simultaneous RFID Reader - M6E Nano* apresentada na arquitetura proposta e o leitor *Edge-50* presente na arquitetura da Empresa XYZ. Ambos utilizam a mesma frequência UHF para leitura de tags RFID e possuem o mesmo SDK de desenvolvimento, sendo possível utilizar as linguagens de programação C, C# ou Java.

Os principais diferenciais dos dois leitores são a quantidade de antenas externas suportadas e o custo de mercado. A placa *SparkFun Simultaneous RFID Reader - M6E Nano* apresenta uma antena interna acoplada na placa, sendo possível adicionar apenas uma antena externa à placa. No leitor *Edge-50* é necessário incluir alguma antena externa ao leitor, suportando até 4 antenas de tags.

O custo de mercado do leitor *Edge-50* é aproximadamente 10 vezes mais alto do que a placa *SparkFun Simultaneous RFID Reader - M6E Nano*. Mesmo com a diferença de suporte das antenas externas, o custo da placa *SparkFun Simultaneous RFID Reader - M6E Nano* ainda é ainda mais baixo em relação ao leitor *Edge-50*.

Para responder a **RQ.3** do estudo de caso deste trabalho, é utilizado um serviço do *Microsoft Azure* capaz de realizar o monitoramento de aplicações hospedadas no provedor. O serviço oferecido pelo *Microsoft Azure*, *Application Insights*, realiza o monitoramento de todas as aplicações da arquitetura, sendo capaz de diagnosticar exceções, problemas de desempenho e de apresentar métricas de uso.

Na arquitetura proposta, a *API Gateway* agrega todos os microsserviços monitorados, sendo responsável por gerenciar e monitorar os microsserviços a serem expostos na aplicação *front-end*.

A Figura 4.3 e Figura 4.4 apresentam o monitoramento realizado nos microsserviços utilizados pela *API Gateway*. É possível observar que qualquer inconsistência nos serviços é apresentada no monitoramento.

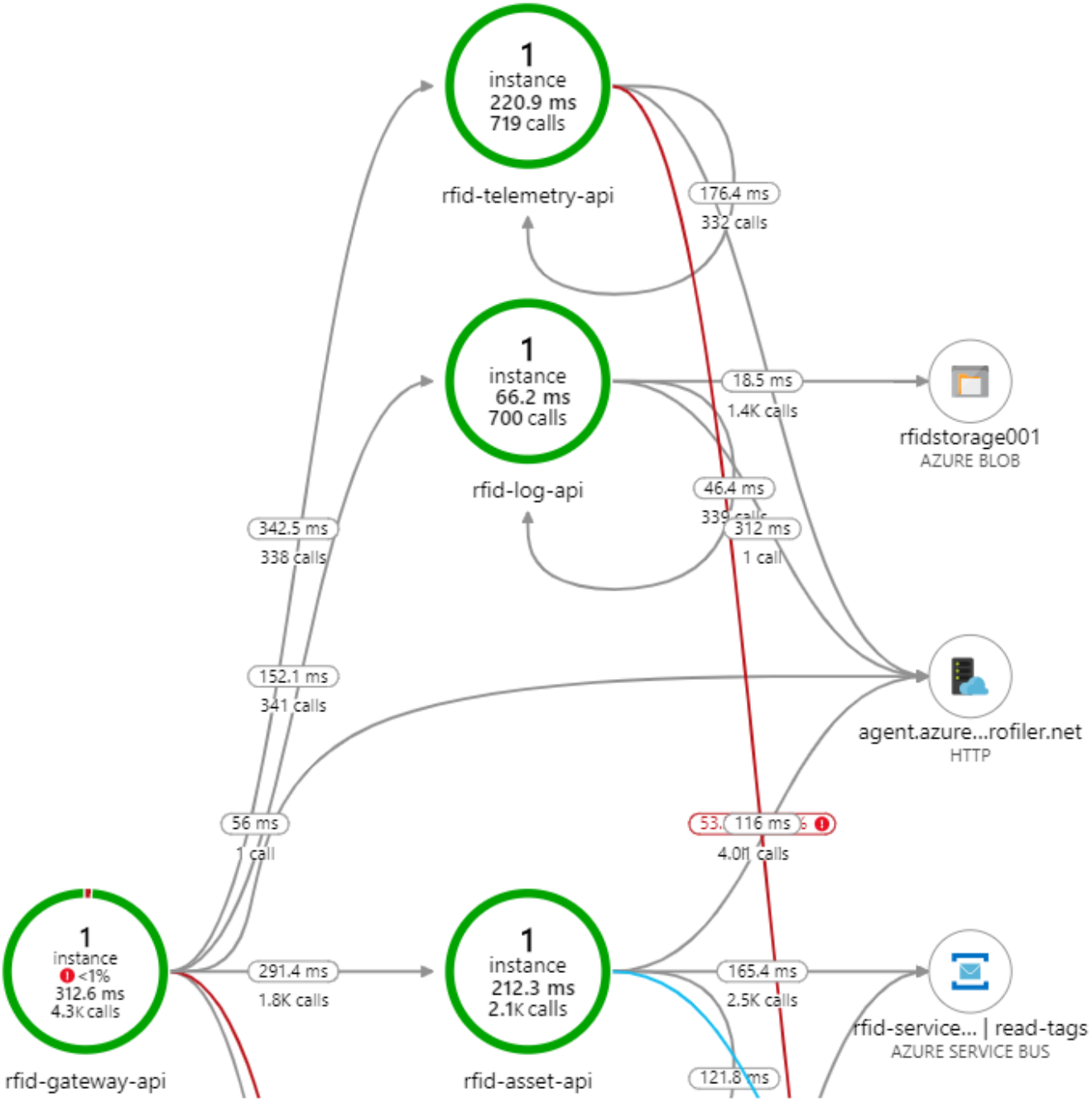

Figura 4.3: Monitoramento dos microsserviços utilizados pela *API Gateway* - Parte 01.

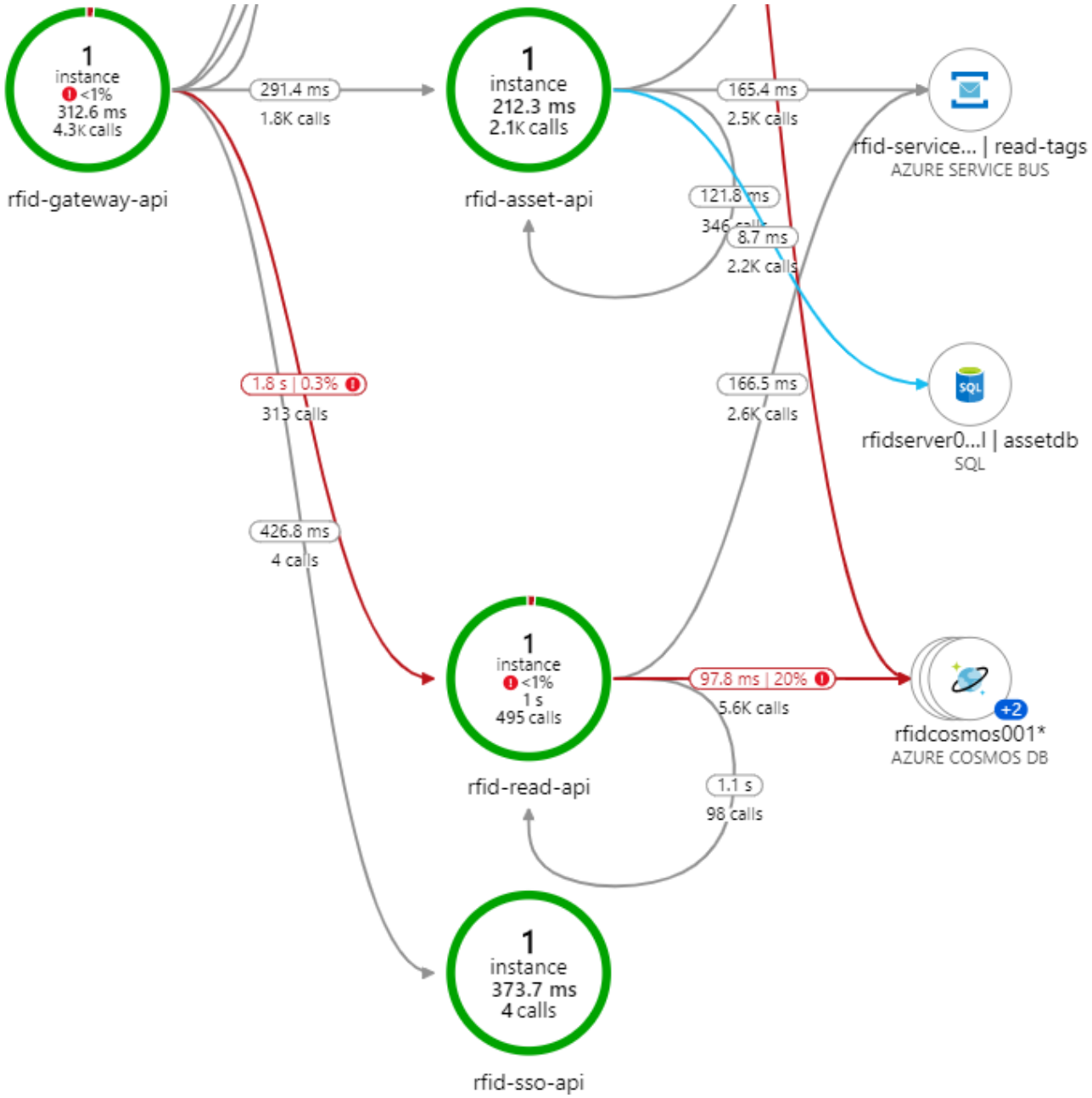

Figura 4.4: Monitoramento dos microsserviços utilizados pela *API Gateway* - Parte 02.

Para validar a performance da arquitetura proposta, a arquitetura de serviços e IoT foram monitorados utilizando o serviço do *Microsoft Azure*, *Azure Monitor*. Através do monitoramento da arquitetura, foram extraídas métricas de performance. Apenas a aplicação *back-end* não foi monitorada, por não mostrar resultados importantes como os extraídos pelo serviço *Serverless* e pela *API Gateway*.

A arquitetura de serviços e IoT foram monitorados no período de 1 hora, com o servidor local processando a leitura de tags RFID e a aplicação *back-end* solicitando 1

requisição para cada *endpoint* da *API Gateway*, a cada 3 segundos. O servidor onde estão hospedadas as aplicações da arquitetura de serviços possui as seguintes configurações:

- **Sistema Operacional:** Unix 4.4.0.128;
- **Memória RAM:** 1.75 *Gigabytes*;
- **Processador:** 1 Core.

Essas são as únicas informações disponibilizadas pelo *Microsoft Azure* sobre a configuração do servidor. O servidor do serviço *Serverless*, *Azure Functions* não possui configuração fixa. O servidor utiliza a mémoria RAM e o processamento de CPU necessário para executar cada solicitação.

A Figura 4.5 apresenta a quantidade de requisições realizadas pela placa *SparkFun Simultaneous RFID Reader - M6E Nano*. As requisições realizadas contemplam os eventos descritos na arquitetura proposta: telemetria, leitura e log. Foram geradas mais de 4000 requisições no período de monitoramento com 3085 inserções de leituras de tags RFID e 5950 inserções de telemetrias na placa, que foram posteriormente inseridas no banco de dados NoSQL *Cosmos DB*.

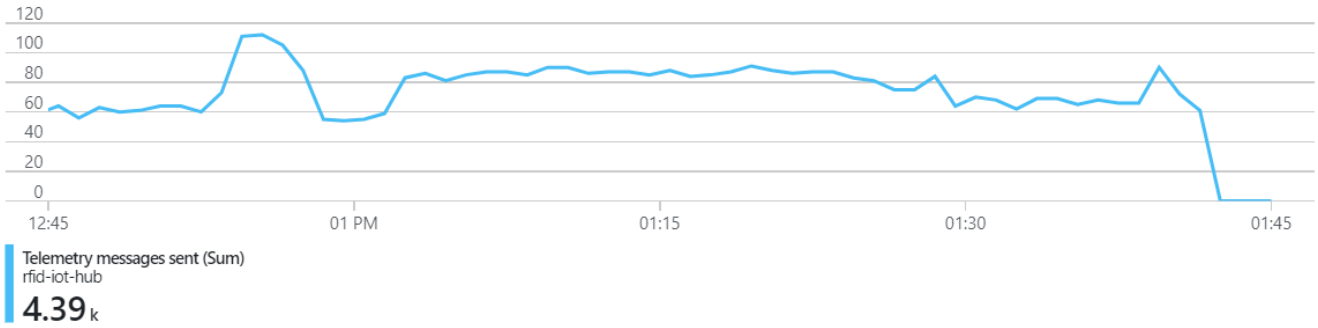

Figura 4.5: Quantidade de requisições na placa *SparkFun Simultaneous RFID Reader - M6E Nano*.

A Figura 4.6 apresenta a utilização média de processamento de CPU e memória RAM no serviço *Serverless*. É comum a grande utilização de memória RAM em relação ao processamento de CPU nesse tipo de serviço, em decorrência da arquitetura de consumo de serviços *Serverless*. No período foi utilizados, em média, 2GB de mémoria RAM e 1% de consumo do processamento da CPU.

Figura 4.6: Processamento de CPU e memória RAM no serviço *Serverless*.

A Figura 4.7 apresenta o tempo médio de resposta do serviço *Serverless* quando solicitado. Em média, o serviço respondia as requisições em um tempo médio de 13.88 milissegundos. Essa resposta rápida deve-se pela arquitetura de consumo de serviços *Serverless*.

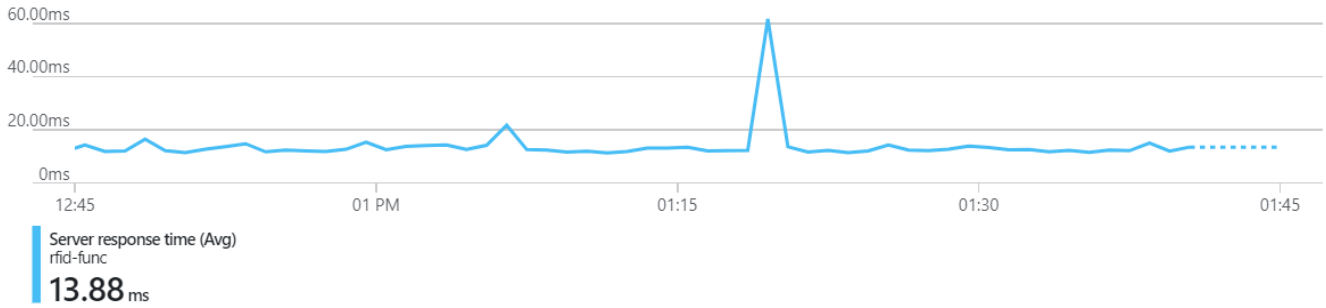

Figura 4.7: Tempo de resposta do serviço *Serverless*.

A Figura 4.8 apresenta a utilização média de processamento de CPU e memória RAM da *API Gateway*. Diferentemente do serviço *Serverless* a um consumo maior de processamento de CPU, mas com o consumo de memória RAM ainda maior. No período, foram utilizados, em média 81% de mémoria RAM e 19% de consumo do processamento da CPU. Esse consumo apresentado também reflete nos servidores dos microsserviços.

A Figura 4.9 apresenta o tempo médio de resposta da *API Gateway* quando solicitada. Em geral, o serviço respondia as requisições em um tempo de aproximadamente 275.54 milissegundos. Essa resposta é maior em comparação com o serviço *Serverless*. Esse consumo apresentado também reflete nos servidores dos microsserviços.

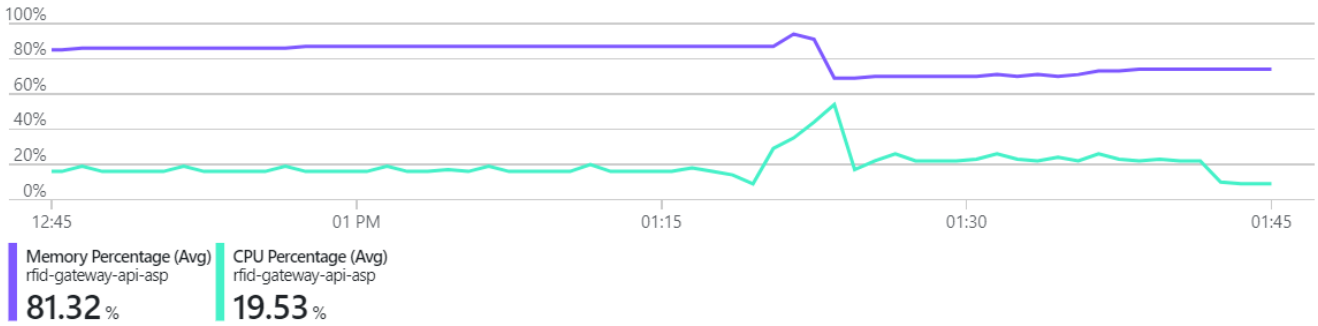

Figura 4.8: Processamento de CPU e memória RAM na *API Gateway*.

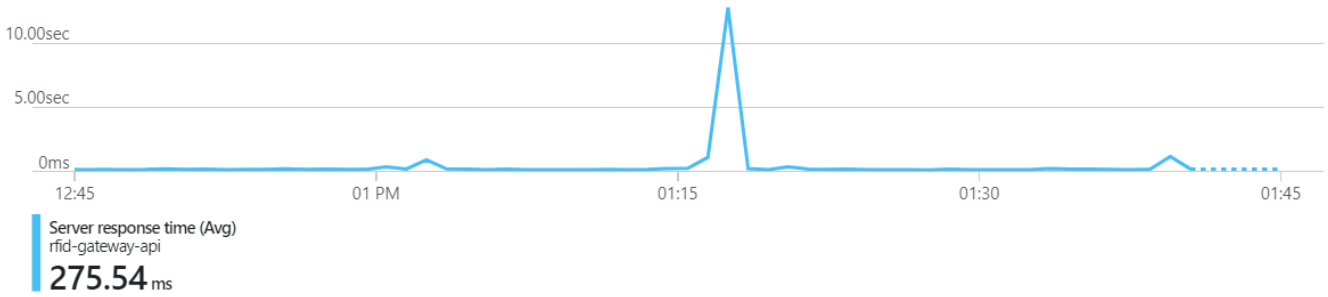

Figura 4.9: Tempo de resposta da *API Gateway*.

A Figura 4.10 apresenta a quantidade de requisições realizadas no barramento de eventos *Service Bus*. Foram realizados um total de 45044 requisições e a inserção de 9076 mensagens. A quantidade alta de requisições deve-se à comunicação dos microsserviços de Ativo e Leitura.

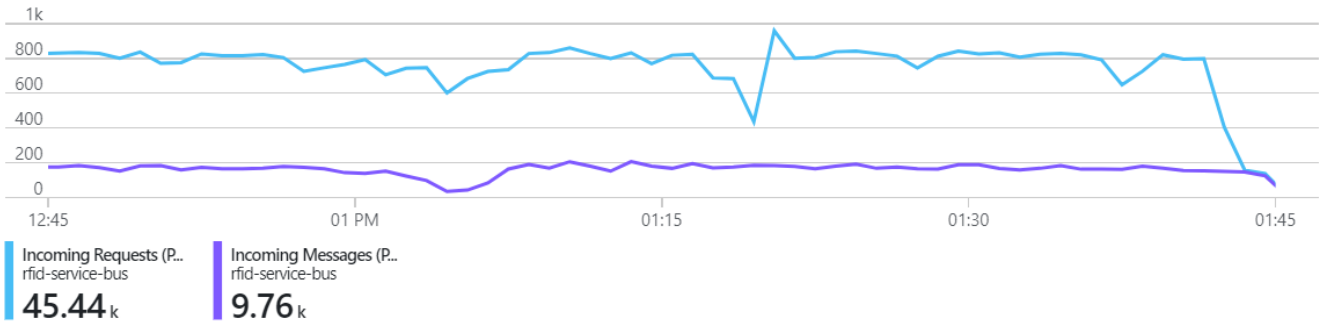

Figura 4.10: Quantidade de requisições no barramento de eventos *Service Bus*.

Por fim, a Figura 4.11 apresenta a quantidade média de requisições realizadas no banco de dados NoSQL *Cosmos DB*. Foram realizadas, em média, 3.29 requisições por segundo.

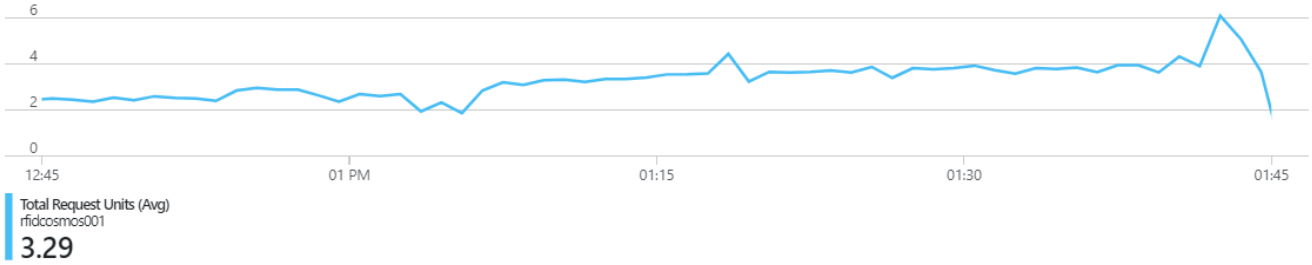

Figura 4.11: Quantidade de requisições no banco de dados *Cosmos DB*.

4.2 Discussão, Lições Aprendidas e Ameaças à Validade

Os resultados obtidos na **RQ.1.** demonstraram a complexidade e alto custo de desenvolvimento da arquitetura proposta utilizando microsserviços e computação em nuvem, em relação à arquitetura monolítica de serviços e a arquitetura de IoT da Empresa XYZ. A arquitetura proposta mostrou-se extremamente necessária para o contexto proposto no estudo de caso deste trabalho. A arquitetura de serviços e de IoT com a utilização de *Serverless*, em conjunto com a cultura de *DevOps* e dos conceitos de integração e entrega contínua, foram fundamentais para o contexto proposto no estudo de caso deste trabalho.

Ademais, a utilização de *containers* em *Docker* e os serviços oferecidos pelo *Microsoft Azure* foram facilitadores para o desenvolvimento da arquitetura proposta, por possuírem serviços específicos para aplicações em IoT e para o gerenciamento, processamento e armazenamento das informações de leitura que a placa *SparkFun Simultaneous RFID Reader - M6E Nano* oferece.

Os resultados obtidos na **RQ.2.** do estudo de caso deste trabalho demonstraram que a placa *SparkFun Simultaneous RFID Reader - M6E Nano* utilizada apresentou características satisfatórias para leitura de tags RFID utilizando a frequência UHF. A placa conseguiu realizar leitura de tags RFID com a distância de aproximadamente 1 metro (apenas com sua antena interna), com a capacidade de leitura de aproximadamente 100 tags RFID por segundo. A temperatura média da placa ficou em torno de 48C^{o}. A utilização de uma antena UHF externa acoplada à placa aumentaria a eficiência de sua leitura. A utilização de uma antena UHF externa é necessária onde a distância de leitura é requisito mínimo para execução desse tipo de projeto.

Outra característica importante apresentada neste trabalho foi a interoperabilidade, flexibilidade e escalabilidade da arquitetura proposta, além da possibilidade de a adequação da arquitetura proposta para outro provedor de nuvem que contenha os mesmos serviços utilizados e o suporte ao uso de *containers* em *Docker*.

Os resultados obtidos na **RQ.3.**, assim como na **RQ.1.**, apresentam a importância do uso dos serviços oferecidos pelo *Microsoft Azure*. Com a utilização dos serviços oferecidos foi realizada a implementação e os testes de performance para responder as questões de pesquisa definidas neste trabalho.

Capítulo 5

Conclusão

Neste trabalho, foi apresentada uma arquitetura que implementa a leitura de tags RFID, utilizando a frequência UHF com um leitor de tags de baixo custo de mercado em relação aos que utilizam essa frequência, em uma infraestrutura composta por computação em nuvem e microsserviços. A utilização da placa *SparkFun Simultaneous RFID Reader - M6E Nano* foi utilizada para redução do custo do equipamento de leitura de tags RFID com frequência UHF.

A metodologia de pesquisa utilizada neste trabalho contribuiu para a construção da arquitetura proposta. A pesquisa bibliográfica e os trabalhos relacionados foram importantes para o desenvolvimento da arquitetura proposta neste trabalho. A implementação da arquitetura proposta seguindo o padrão de Internet das Coisas apresentado pela pesquisa bibliográfica e os trabalhos relacionados, se mostraram um facilitador para o desenvolvimento da arquitetura. A divisão arquitetural em dois grandes grupos (Grupo de Internet das Coisas e Grupo de Serviços) foi um facilitador para implementação da arquitetura proposta seguindo padrão de Internet das Coisas.

A arquitetura proposta foi aplicada em um estudo de caso real para verificar a sua aderência e conformidade neste trabalho. Os resultados apresentados demonstraram a importância da utilização da computação em nuvem e seus serviços oferecidos para a arquitetura proposta. A utilização de microsserviços foi outro facilitador dentro da arquitetura proposta, sendo importante nos resultados obtidos. Os testes de performance realizados validaram as decisões arquiteturais propostas no trabalho.

A utilização de computação em nuvem e microsserviços demonstraram ter um custo alto de desenvolvimento na arquitetura proposta, em decorrência de suas complexidades e da quantidade de recursos criados para implementação da arquitetura.

Para a leitura de tags RFID foi identificado que, em cenários onde a distância de leitura é um requisito fundamental, é necessário incluir uma antena externa para obter melhores resultados nesse quesito.

5.1 Trabalhos Futuros

Como trabalhos futuros, são propostos alguns pontos que podem ser evoluídos, testados e implementados na arquitetura proposta.

Na Camada de *Middleware*, avaliar outros dispositivos de *hardwares* como microcontroladores ou microcomputadores para realizar a função do servidor local. Na Camada de Rede, avaliar outros tipos de comunicação diferentes da implementada na arquitetura proposta. Na arquitetura de serviços, avaliar outros tipos de orquestradores de *containers* como o *Kubernetes*.

Existe a necessidade de um estudo mais profundo de segurança, na leitura de leitor de tags RFID, sobretudo na placa *SparkFun Simultaneous RFID Reader - M6E Nano*. As tags RFID possuem alguns mecanismos de segurança como inserção de senha para leitura e/ou gravação e até possibilidade de bloquear a tag RFID, não sendo mais possível fazer nenhum tipo de ação sobre a tag RFID.

Por fim, realizar outros experimentos práticos para avaliar o uso da arquitetura proposta em outros contextos relacionados à utilização de IoT e leitura de tags RFID.

5.2 Publicações Relacionadas

Santos, Yago Luiz dos e Edna Dias Canedo: On the Design and Implementation of an IoT Based Architecture for Reading Ultra High Frequency Tags. Information, 10(2), 2019, ISSN 2078-2489. *http://www.mdpi.com/2078-2489/10/2/41*.

Referências

[1] Perera, Charith, Arkady Zaslavsky, Peter Christen e Dimitrios Georgakopoulos: *Context Aware Computing for The Internet of Things: A Survey*. IEEE Communications Surveys & Tutorials, 16(1):414–454, 2014, ISSN 1553-877X. `http://ieeexplore.ieee.org/document/6512846/`. x, 6, 7, 10

[2] Khan, Rafiullah, Sarmad Ullah Khan, Rifaqat Zaheer e Shahid Khan: *Future internet: The internet of things architecture, possible applications and key challenges*. Proceedings - 10th International Conference on Frontiers of Information Technology, FIT 2012, páginas 257–260, 2012, ISSN 1556-3669. x, 6, 8, 10

[3] Bolic, Miodrag, David Simplot-ryl e Ivan Stojmenovi: *RFID Systems: Research Trends and Challanges*. John Wiley & Sons, 2010, ISBN 9780470746028. x, xii, 1, 10, 11, 12, 13

[4] Zhang, Qi, Lu Cheng e Raouf Boutaba: *Cloud computing: State-of-the-art and research challenges*. Journal of Internet Services and Applications, 1(1):7–18, 2010, ISSN 18674828. x, 13, 14, 15, 16, 17

[5] Gartner: *Magic quadrant for cloud infrastructure as a service, worldwide*. https://www.gartner.com/doc/reprints?id=1-50WJ5CK&ct=180525&st=sb. Acessado: 07-04-2019. x, 15

[6] Burns, Brendan: *Designing Distributed Systems - Patterns and Paradigms for Scalable, Reliable Services*, volume 80. O'Reilly Media, Inc., 2018, ISBN 9789004310087. x, 20, 21, 22

[7] Juels, A.: *RFID security and privacy: a research survey*. IEEE Journal on Selected Areas in Communications, 24(2):381–394, fevereiro 2006, ISSN 0733-8716. `http://airccj.org/CSCP/vol3/csit3526.pdfhttp://ieeexplore.ieee.org/document/1589116/`. xii, 10, 11, 12, 13

[8] Li, Dong ying, Shun dao Xie e Rong jun Chen: *Design of Internet of Things System for Library Materials Management Using UHF RFID*. IEEE International Conference on RFID Technology and Applications (RFID-TA) Design, páginas 44–48, 2016. xii, 1, 11, 12

[9] Gubbi, Jayavardhana, Rajkumar Buyya, Slaven Marusic e Marimuthu Palaniswami: *Internet of Things (IoT): A vision, architectural elements, and future directions*. Future Generation Computer Systems, 29(7):1645–1660, 2013, ISSN 0167739X. `http://dx.doi.org/10.1016/j.future.2013.01.010`. 1, 6

[10] Filho, Francisco L. de Caldas, Lucas M. C. e Martins, Ingrid Palma Araújo, Fábio L. L. de Mendonça, João Paulo C. L. da Costa e Rafael T. de Sousa Júnior: *Design and Evaluation of a Semantic Gateway Prototype for IoT Networks*. Em *Companion Proceedings of the10th International Conference on Utility and Cloud Computing - UCC '17 Companion*, páginas 195–201, New York, New York, USA, 2017. ACM Press, ISBN 9781450351959. `http://dl.acm.org/citation.cfm?doid=3147234.3148091`. 1, 6

[11] Rayes, Ammar e Samer Salam: *Internet of Things (IoT) Overview*. Internet of Things From Hype to Reality, páginas 1–34, 2017, ISSN 0167-739X. 1, 6, 22

[12] Kevin, Ashton: *That 'internet of things' thing*. RFID journal, 22(7):97–114, 2009. 1

[13] Buyya, Rajkumar, Rajkumar Buyya, Chee Shin Yeo, Chee Shin Yeo, Srikumar Venugopal, Srikumar Venugopal, James Broberg, James Broberg, Ivona Brandic e Ivona Brandic: *Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility*. Future Generation Computer Systems, 25(June 2009):17, 2009, ISSN 0167-739. `http://portal.acm.org/citation.cfm?id=1528937.1529211`. 1, 13, 14, 15

[14] Dragoni, Nicola, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin e Larisa Safina: *Microservices: Yesterday, Today, and Tomorrow*. Em *Present and Ulterior Software Engineering*, páginas 195–216. Springer International Publishing, Cham, junho 2017, ISBN 9783319674254. `http://arxiv.org/abs/1606.04036http://link.springer.com/10.1007/978-3-319-67425-4_12`. 1, 17, 18, 20, 21, 22, 37, 41

[15] Lewis, James; Fowler, Martin: *Microservices - a definition of this new architectural term*. https://martinfowler.com/articles/microservices.html. Acessado: 07-04-2019. 1, 17, 22

[16] Sun, Long, Yan Li e Raheel Ahmed Memon: *An open IoT framework based on microservices architecture*. China Communications, 14(2):154–162, 2017, ISSN 16735447. 1, 22

[17] Prudanov, Anton, Sergey Tkachev, Nikolay Golos, Pavel Masek, Jiri Hosek, Radek Fujdiak, Krystof Zeman, Aleksandr Ometov, Sergey Bezzateev, Natalia Voloshina, Sergey Andreev e Jiri Misurec: *A trial of yoking-proof protocol in RFID-based smart-home environment*. Communications in Computer and Information Science, 678(November):25–34, 2016, ISSN 18650929. 2, 23

[18] Farris, Ivan, Sara Pizzi, Massimo Merenda, Antonella Molinaro, Riccaro Carotenuto e Antonio Iera: *6lo-RFID: A framework for full integration of smart UHF RFID tags into the internet of things*. IEEE Network, 31(5):66–73, 2017, ISSN 08908044. 2, 23

[19] Chieochan, Oran, Aukit Saokaew e Ekkarat Boonchieng: *An integrated system of applying the use of Internet of Things, RFID and cloud computing: A case study of logistic management of Electricity Generation Authority of Thailand (EGAT) Mae Mao Lignite Coal Mining, Lampang, Thailand*. Em *2017 9th International Conference*

on Knowledge and Smart Technology: Crunching Information of Everything, KST 2017, páginas 156–161, 2017, ISBN 9781467390774. 2, 12, 23

[20] Gil, Antonio Carlos: *Métodos e técnicas de pesquisa social*. 6. ed. - São Paulo: Atlas, 2008, ISBN 978-85-224-5142-5. 3, 4

[21] Marconi, Marina de Andrade e Eva Maria Lakatos: *Fundamentos de metodologia científica*. 5. ed. - São Paulo: Atlas, 2003, ISBN 85-224-3397-6. 3, 4

[22] Runeson, Per, Martin Höst, Austen Rainer e Björn Regnell: *Case Study Research in Software Engineering*. John Wiley & Sons, Inc., Hoboken, NJ, USA, março 2012, ISBN 9781118181034. `http://www.wiley.com/WileyCDA/WileyTitle/productCd-1118104358.html%5Cnhttp://doi.wiley.com/10.1002/9781118181034http://doi.wiley.com/10.1002/9781118181034`. 4, 43

[23] Botta, Alessio, Walter De Donato, Valerio Persico e Antonio Pescape: *On the integration of cloud computing and internet of things*. Proceedings - 2014 International Conference on Future Internet of Things and Cloud, FiCloud 2014, páginas 23–30, 2014, ISSN 0167739X. 6, 22

[24] Vashi, Shivangi, Jyotsnamayee Ram, Janit Modi, Saurav Verma e Chetana Prakash: *Internet of Things (IoT): A vision, architectural elements, and security issues*. Em *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, volume 16 de *1*, páginas 492–496. IEEE, fevereiro 2017, ISBN 978-1-5090-3242-6. `http://ieeexplore.ieee.org/document/6512846/http://ieeexplore.ieee.org/document/8058399/`. 7, 8

[25] Yousaf, Hasnain: *Internet of Things: 'A panoramic observation'*. Em *International Conference on Communication Technologies, ComTech 2017*, páginas 27–33, 2017, ISBN 9781509059843. 7

[26] Tsiropoulou, Eirini Eleni, John S. Baras, Symeon Papavassiliou e Surbhit Sinha: *RFID-based smart parking management system*. Cyber-Physical Systems, 3(1-4):22–41, 2017, ISSN 2333-5777. `https://www.tandfonline.com/doi/full/10.1080/23335777.2017.1358765`. 10, 23

[27] Huiting, Jordy, Andre B.J. Kokkeler e Gerard J.M. Smit: *The effects of single bit quantization on direction of arrival estimation of UHF RFID tags*. 2016 IEEE International Conference on RFID Technology and Applications (RFID-TA), páginas 55–60, 2016. `http://ieeexplore.ieee.org/document/7750731/`. 10

[28] Casula, G. A., G. Montisci, A. Michel e P. Nepa: *Analysis of wearable ungrounded antennas for UHF RFIDs with respect to the coupling with human-body*. 2016 IEEE International Conference on RFID Technology and Applications, RFID-TA 2016, páginas 6–9, 2016. 10

[29] Chen, Honglong, Guoliang Xue e Zhibo Wang: *Efficient and Reliable Missing Tag Identification for Large-Scale RFID Systems with Unknown Tags*. IEEE Internet of Things Journal, 4(3):736–748, 2017, ISSN 23274662. 11

[30] Jadeja, Yashpalsinh e Kirit Modi: *Cloud computing - Concepts, architecture and challenges*. 2012 International Conference on Computing, Electronics and Electrical Technologies, ICCEET 2012, 1(November):877–880, 2012. 13, 14, 15, 16, 17

[31] Puthal, Deepak, B.P.S. Sahoo, Sambit Mishra e Satyabrata Swain: *Cloud Computing Features, Issues, and Challenges: A Big Picture*. Em *2015 International Conference on Computational Intelligence and Networks*, número Cine em *1*, páginas 116–123. IEEE, janeiro 2015, ISBN 978-1-4799-7548-8. `http://ieeexplore.ieee.org/document/7053814/`. 13, 14, 15, 16, 17

[32] Buyya, Rajkumar, Rajiv Ranjan e Rodrigo N Calheiros: *InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services*. International Conference on Algorithms and Architectures for Parallel Processing, páginas 13–31, 2010. `http://link.springer.com/10.1007/978-3-642-13119-6_2`. 15, 16

[33] Martins, Lucas M. C. e, Francisco L De Caldas Filho, Rafael T De Sousa Júnior, William F Giozza e João Paulo C.L. da Costa: *Increasing the Dependability of IoT Middleware with Cloud Computing and Microservices*. Em *Companion Proceedings of the10th International Conference on Utility and Cloud Computing - UCC '17 Companion*, páginas 203–208, New York, New York, USA, 2017. ACM Press, ISBN 9781450351959. `http://dl.acm.org/citation.cfm?doid=3147234.3148092`. 17, 21, 22

[34] Richardson, Chris: *Microservices patterns*. http://microservices.io/patterns/index.html. Acessado: 07-04-2019. 18, 20, 22, 37

[35] Vresk, Tomislav e Igor Cavrak: *Architecture of an interoperable IoT platform based on microservices*. 2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2016 - Proceedings, páginas 1196–1201, 2016, ISSN 2327-4662. 22

[36] Celesti, Antonio, Lorenzo Carnevale, Antonino Galletta, Maria Fazio e Massimo Villari: *A Watchdog Service Making Container-Based Micro-services Reliable in IoT Clouds*. Em *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, número Lcv em *1*, páginas 372–378. IEEE, agosto 2017, ISBN 978-1-5386-2074-8. `http://ieeexplore.ieee.org/document/8114506/`. 22

[37] Vandikas, Konstantinos e Vlasios Tsiatsis: *Microservices in IoT clouds*. 2016 Cloudification of the Internet of Things, CIoT 2016, páginas 1–6, 2017. 22, 24, 40

[38] Ferreira, Hiro Gabriel Cerqueira, Edna Dias Canedo e Rafael Timoteo De Sousa: *IoT architecture to enable intercommunication through REST API and UPnP using IP, ZigBee and arduino*. International Conference on Wireless and Mobile Computing, Networking and Communications, páginas 53–60, 2013, ISSN 21619646. 22

[39] Ferreira, Hiro Gabriel Cerqueira, Edna Dias Canedo e Rafael Timóteo De Sousa Junior: *A ubiquitous communication architecture integrating transparent UPnP and REST APIs*. International Journal of Embedded Systems, 6(2/3):188, 2014, ISSN 1741-1068. `http://www.inderscience.com/link.php?id=63816`. 22

[40] Khazaei, Hamzeh, Hadi Bannazadeh e Alberto Leon-Garcia: *End-to-end management of IoT applications*. 2017 IEEE Conference on Network Softwarization: Softwarization Sustaining a Hyper-Connected World: en Route to 5G, NetSoft 2017, 2017. 24, 40

[41] Dias, Lucas B, Maristela Holanda, Ruben C Huacarpuma e Rafael T. de Sousa Jr: *NoSQL Database Performance Tuning for IoT Data - Cassandra Case Study*. Em *Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security*, páginas 277–284. SCITEPRESS - Science and Technology Publications, 2018, ISBN 978-989-758-296-7. `http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006782702770284`. 24

Apêndice A

Códigos Fontes

Abaixo é apresentado os repositórios onde estão localizados todos os códigos fontes implementados neste trabalho:

- ***Middleware* IoT:** `https://github.com/yagoluiz/rfid-reader-middleware`
- **Microsserviços e serviço *Serverless*:** `https://github.com/yagoluiz/rfid-reader-api`
- **Aplicação *front-end*:** `https://github.com/yagoluiz/rfid-reader-web`

A.1 Monitoramento da telemetria realizado pela placa RFID

```
1  public class DeviceMessageSender implements Runnable {
2
3      @Override
4      public void run() {
5          try {
6              // Telemetria realizada de modo continuo
7              while (true) {
8                  // Informacoes de telemetria:
9                  // (IP, temperatura e status da conexao)
10                 TelemetryModel telemetryModel = new TelemetryModel(
11                     InetAddress.getLocalHost().getHostAddress(),
12                     ReaderService.getTemperature(),
13                     TelemetryTypeEnum.TELEMETRY.getEnumValue(),
14                     ReaderService.isConnection());
15
16                 // Informacoes de telemetria no formato JSON
17                 String telemetryJson = new JsonUtil()
18                     .serialize(telemetryModel);
```

```
                Message msg = new Message(telemetryJson);

                // Informacoes de callback para biblioteca Iot Hub
                Object lock = new Object();
                DeviceEventCallback callback = new DeviceEventCallback();

                // Envio das informacoes de telemetria:
                // (JSON com informacoes e callback)
                IoTHubService.sendEventDeviceAsync(msg, callback, lock);

                // Sincronizacao das informacoes
                synchronized (lock) {
                    lock.wait();
                }

                System.out.println("Sending message Telemetry: "
                    + telemetryJson);

                // Telemetria realizada a cada 1 segundo
                Thread.sleep(1000);
            }
        } catch (InterruptedException | UnknownHostException |
            URISyntaxException | ReaderException e) {
            System.out.println("Interrupted telemetry");
            e.printStackTrace();
        }
    }

    // Implementacao da biblioteca IoT Hub para envio de informacoes
    public static class DeviceEventCallback implements IotHubEventCallback
    {
        public void execute(IotHubStatusCode status, Object context) {
            System.out.println("IoT Hub responded to message with status: "
                + status.name());

            if (context != null) {
                synchronized (context) {
                    context.notify(); // Sincronizacao de informacoes
                }
            }
        }
    }
}
```

A.2 Leitura e log realizado pela placa RFID

```
public class ReaderMessageSender implements Runnable {

    @Override
    public void run() {
        try {
            TagReadData[] tagReads;
            // Leitura realizada de modo continuo
            while (true) {
                // Leitura realizada a cada 1 segundo
                Thread.sleep(1000);
                // Limite de 10 tags lidas a cada 1 milissegundo
                tagReads = ReaderService.readDataTag(10);
                for (TagReadData tr : tagReads) {
                    if (!tr.epcString().isEmpty()) {

                        // Informacoes de leitura:
                        // (IP, EPC, data e antena)
                        ReadModel readModel = new ReadModel(
                            InetAddress.getLocalHost().getHostAddress(),
                            tr.epcString(),
                            LocalDateTime.now().toString(),
                            tr.getAntenna(),
                            TelemetryTypeEnum.READ.getEnumValue());

                        // Informacoes de leitura no formato JSON
                        String readJson = new JsonUtil()
                            .serialize(readModel);
                        Message msg = new Message(readJson);

                        // Informacoes de callback para biblioteca Iot Hub
                        Object lock = new Object();
                        DeviceMessageSender.DeviceEventCallback callback =
                            new DeviceMessageSender.DeviceEventCallback();

                        // Envio das informacoes de leitura:
                        //(JSON com informacoes e callback)
                        IoTHubService
                            .sendEventDeviceAsync(msg, callback, lock);

                        // Sincronizacao das informacoes
                        synchronized (lock) {
                            lock.wait();
                        }
```

```
44
45                          System.out.println("Sending message Read: "
46                              + readJson);
47                      }
48                  }
49              }
50          } catch (Exception e) {
51              try {
52                  // Informacoes de erro na leitura da placa RFID
53                  synchronizeException(e);
54              } catch (URISyntaxException | UnknownHostException se) {
55                  se.printStackTrace();
56              }
57          }
58      }
59
60
61      private static void synchronizeException(Exception exception)
62          throws URISyntaxException, UnknownHostException {
63          // Informacoes de erro:
64          // (IP, nome do erro, pilha de erros e data)
65          LogModel logModel = new LogModel(
66                  InetAddress.getLocalHost().getHostAddress(),
67                  exception.getMessage(),
68                  exception.getStackTrace().toString(),
69                  LocalDateTime.now().toString(),
70                  TelemetryTypeEnum.LOG.getEnumValue());
71
72          // Informacoes de erro no formato JSON
73          String logJson = new JsonUtil().serialize(logModel);
74          Message msg = new Message(logJson);
75
76          // Informacoes de callback para biblioteca Iot Hub
77          Object lock = new Object();
78          DeviceMessageSender.DeviceEventCallback callback = new
    DeviceMessageSender.DeviceEventCallback();
79
80          // Envio das informacoes de erro (JSON com informacoes e callback)
81          IoTHubService.sendEventDeviceAsync(msg, callback, lock);
82
83          System.out.println("Sending message Log: " + logJson);
84      }
85  }
```

A.3 *Azure Functions* com informações providas do *IoT Hub*

```
public static class IoTTriggerFunction
{
        [FunctionName("IoTTriggerFunction")]
        public static async Task Run(
            // Configuracao de escuta de mensagem via IoT Hub
            [IoTHubTrigger("messages/events",
            Connection = "IoTHubConnectionString",
            ConsumerGroup = "$Default")]EventData[] events,
            // Configuracao de banco de dados Cosmos DB para evento de telemetria
            [CosmosDB(databaseName: "Rfid", collectionName: "Telemetry",
            ConnectionStringSetting = "CosmosDBConnection")]
            DocumentClient clientTelemetry,
            // Configuracao de banco de dados Cosmos DB para evento de leitura
            [CosmosDB(databaseName: "Rfid", collectionName: "Read",
            ConnectionStringSetting = "CosmosDBConnection")]
            DocumentClient clientRead,
            // Configuracao de Blob Storage para evento de log
            [Blob("logs", Connection = "BlobConnectionString")]
            CloudBlobContainer blobContainer,
            ILogger log)
        {
            log.LogInformation
                ($"EventHubTriggerFunction executed: {DateTime.Now}");

            // Listagem de mensagens do IoT Hub
            foreach (EventData eventData in events)
            {
                // Informacoes da mensagem
                var messageBody = Encoding.UTF8
                    .GetString(eventData.Body.Array,
                    eventData.Body.Offset,
                    eventData.Body.Count);

                // Informacao do tipo de mensagem:
                // (Telemetria, leitura ou log)
                var telemetryTypeModel = JsonConvert
                    .DeserializeObject<TelemetryTypeModel>(messageBody);

                log.LogInformation
```

```
40                      ($"EventHubTriggerFunction message: {messageBody}");
41                  log.LogInformation
42                      ($"EventHubTriggerFunction telemetry type:
43                          {telemetryTypeModel.TelemetryType}");
44
45                  // Envio de mensagem de acordo com o tipo
46                  switch (telemetryTypeModel.TelemetryType)
47                  {
48                      // Informacoes de telemetria (Cosmos BD)
49                      case (int)TelemetryTypeEnum.TELEMETRY:
50                          // Informacoes de telemetria
51                          var telemetryModel = JsonConvert
52                              .DeserializeObject<TelemetryModel>
53                              (messageBody);
54                          // Envio das informacoes de telemetria
55                          await clientTelemetry.CreateDocumentAsync
56                              (UriFactory.CreateDocumentCollectionUri
57                              ("Rfid", "Telemetry"), telemetryModel);
58                          break;
59                      // Informacoes de leitura (Cosmos BD)
60                      case (int)TelemetryTypeEnum.READ:
61                          // Informacoes de leitura
62                          var readModel = JsonConvert
63                              .DeserializeObject<ReadModel>(messageBody);
64                          // Envio das informacoes de leitura
65                          await clientRead.CreateDocumentAsync
66                              (UriFactory.CreateDocumentCollectionUri
67                              ("Rfid", "Read"), readModel);
68                          break;
69                      // Informacoes de log (Blob Storage)
70                      case (int)TelemetryTypeEnum.LOG:
71                          // Informacoes de log
72                          var logName = Guid.NewGuid().ToString();
73                          var cloudBlockBlob = blobContainer
74                              .GetBlockBlobReference($"{logName}.json");
75                          // Envio das informacoes de log
76                          await cloudBlockBlob.UploadTextAsync(messageBody);
77                          break;
78
79                      default:
80                          log.LogInformation
81                          ($"EventHubTriggerFunction telemetry type invalid:
    {telemetryTypeModel.TelemetryType}");
82                          break;
83                  }
```

```
84
85                  // Lista proxima mensagem apos o processamento da anterior
86                  await Task.Yield();
87              }
88
89              log.LogInformation
90                  ($"EventHubTriggerFunction finished: {DateTime.Now}");
91          }
92  }
```

A.4 Configuração de autenticação e autorização do cliente na *API Gateway*

```
1   public class Startup
2   {
3       // Configuracao de variaveis de ambiente
4       public Startup(IConfiguration configuration)
5       {
6           Configuration = configuration;
7       }
8
9       // Variaveis de ambiente
10      public IConfiguration Configuration { get; }
11
12      public void ConfigureServices(IServiceCollection services)
13      {
14          // Configuracao de autenticacao de cliente do API Gateway
15          void options(IdentityServerAuthenticationOptions identityServer)
16          {
17              identityServer.Authority = Configuration
18                  .GetSection("IdentityServer:Authority").Value;
19              identityServer.ApiName = Configuration
20                  .GetSection("IdentityServer:ApiName").Value;
21              identityServer.ApiSecret = Configuration
22                  .GetSection("IdentityServer:ApiSecret").Value;
23              identityServer.SupportedTokens = SupportedTokens.Both;
24          }
25
26          // Configuracao de versao do framework ASP.NET Core
27          services.AddMvc()
28              .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
29          // Configuracao de CORS
30          services.AddCors();
31          // Configuracao de biblioteca Ocelot
```

```
32          services.AddOcelot(Configuration);
33          // Configuracao de autenticacao de cliente da API Gateway
34          services.AddAuthentication()
35          .AddIdentityServerAuthentication
36              (Configuration.GetSection("IdentityServer:ProviderKey")
37                  .Value, options);
38      }
39
40      public void Configure(IApplicationBuilder app, IHostingEnvironment env)
41      {
42          // Configuracao de ambiente de desenvolvimento e producao
43          if (env.IsDevelopment())
44          {
45              app.UseDeveloperExceptionPage();
46          }
47          else
48          {
49              app.UseHsts();
50          }
51
52          // Configuracao utilizacao de HTTPS
53          app.UseHttpsRedirection();
54          // Configuracao de CORS para acesso aos endpoints
55          app.UseCors(x =>
56          {
57              x.AllowAnyOrigin();
58              x.AllowAnyHeader();
59              x.AllowAnyMethod();
60              x.AllowCredentials();
61          });
62          // Configuracao de biblioteca Ocelot
63          app.UseOcelot().Wait();
64      }
65  }
```

A.5 *Endpoints* disponibilizados pela *API Gateway*

```
1  {
2    // Configuracao de endpoints
3    "ReRoutes": [
4      {
5        "DownstreamPathTemplate": "/connect/token", // Endpoint microsservico
       de Identidade
6        "DownstreamScheme": "https", // Configuracao de HTTPS
7        "DownstreamHostAndPorts": [
```

```
8         {
9           "Host": "sso.api", // Local de hospedagem
10          "Port": 443 // Porta TCP
11        }
12      ],
13      "UpstreamPathTemplate": "/connect/token", // Endpoint API Gateway
14      "UpstreamHttpMethod": [ "Post" ] // Verbo HTTP
15    },
16    {
17      "DownstreamPathTemplate": "/api/{version}/assets/read", // Endpoint
    microsservico de Ativo
18      "DownstreamScheme": "https", // Configuracao de HTTPS
19      "DownstreamHostAndPorts": [
20        {
21          "Host": "asset.api", // Local de hospedagem
22          "Port": 443 // Porta TCP
23        }
24      ],
25      "UpstreamPathTemplate": "/api/{version}/assets/read", // Endpoint API
     Gateway
26      "UpstreamHttpMethod": [ "Get" ], // Verbo HTTP
27      "AuthenticationOptions": {
28        "AuthenticationProviderKey": "api_gateway", // Permissao de
    autenticacao do cliente
29        "AllowedScopes": [ "gateway" ] // Permissao de scopo
30      }
31    },
32    {
33      "DownstreamPathTemplate": "/api/{version}/logs/limit/{limit}", //
    Endpoint microsservico de Log
34      "DownstreamScheme": "https", // Configuracao de HTTPS
35      "DownstreamHostAndPorts": [
36        {
37          "Host": "log.api", // Local de hospedagem
38          "Port": 443 // Porta TCP
39        }
40      ],
41      "UpstreamPathTemplate": "/api/{version}/logs/limit/{limit}", //
    Endpoint API Gateway
42      "UpstreamHttpMethod": [ "Get" ], // Verbo HTTP
43      "AuthenticationOptions": {
44        "AuthenticationProviderKey": "api_gateway", // Permissao de
    autenticacao do cliente
45        "AllowedScopes": [ "gateway" ] // Permissao de scopo
46      }
```

```
    },
    {
      "DownstreamPathTemplate": "/api/{version}/reads/limit/{limit}", // Endpoint microsservico de Leitura
      "DownstreamScheme": "https", // Configuracao de HTTPS
      "DownstreamHostAndPorts": [
        {
          "Host": "read.api", // Local de hospedagem
          "Port": 443 // Porta TCP
        }
      ],
      "UpstreamPathTemplate": "/api/{version}/reads/limit/{limit}", // Endpoint API Gateway
      "UpstreamHttpMethod": [ "Get" ], // Verbo HTTP
      "AuthenticationOptions": {
        "AuthenticationProviderKey": "api_gateway", // Permissao de autenticacao do cliente
        "AllowedScopes": [ "gateway" ] // Permissao de scopo
      }
    },
    {
      "DownstreamPathTemplate": "/api/{version}/telemetries/limit/{limit}", // Endpoint microsservico de Telemetria
      "DownstreamScheme": "https", // Configuracao de HTTPS
      "DownstreamHostAndPorts": [
        {
          "Host": "telemetry.api", // Local de hospedagem
          "Port": 443 // Porta TCP
        }
      ],
      "UpstreamPathTemplate": "/api/{version}/telemetries/limit/{limit}", // Endpoint API Gateway
      "UpstreamHttpMethod": [ "Get" ], // Verbo HTTP
      "AuthenticationOptions": {
        "AuthenticationProviderKey": "api_gateway", // Permissao de autenticacao do cliente
        "AllowedScopes": [ "gateway" ] // Permissao de scopo
      }
    }
  ],
  "GlobalConfiguration": {}
}
```

A.6 Configuração da imagem *Docker* do microsserviço de Ativo

```
# Imagem do framework ASP.NET Core
FROM mcr.microsoft.com/dotnet/core/aspnet:2.2-stretch-slim AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

# Restauracao e compilacao do projeto API
FROM mcr.microsoft.com/dotnet/core/sdk:2.2-stretch AS build
WORKDIR /src
COPY . .
WORKDIR /src/services/asset/Asset.API
RUN dotnet restore
RUN dotnet build --no-restore -c Release -o /app

# Publicacao do projeto compilado
FROM build AS publish
RUN dotnet publish --no-restore -c Release -o /app

# Publicacao da imagem
FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "Asset.API.dll"]
```

A.7 Configuração da imagem *Docker* da aplicação *back-end*

```
# Imagem do framework Node.js
FROM node:latest as node

# Execucao de comandos para instalacao
WORKDIR /app
COPY . .
RUN npm install
RUN npm install node-sass
RUN npm run build --prod

# Publicacao da imagem
FROM nginx:alpine
COPY --from=node /app/dist/rfid-reader-web /usr/share/nginx/html
```

A.8 Configuração das imagens *Docker* dos microsserviços e da *API Gateway* no arquivo *Docker Compose*

```
version: '3.4'

# Configuracao das imagens dos microsservicos e da API Gateway
services:
# Configuracao da imagem da API Gateway
  ocelot.api:
    image: ${DOCKER_REGISTRY-}ocelotapi
    build:
      context: .
      dockerfile: gateway/ocelot/Ocelot.API/Dockerfile
# Configuracao da imagem do microsservico de Identidade
  sso.api:
    image: ${DOCKER_REGISTRY-}ssoapi
    build:
      context: .
      dockerfile: security/sso/SSO.API/Dockerfile
# Configuracao da imagem do microsservico de Ativo
  asset.api:
    image: ${DOCKER_REGISTRY-}assetapi
    build:
      context: .
      dockerfile: services/asset/Asset.API/Dockerfile
# Configuracao da imagem do microsservico de Log
  log.api:
    image: ${DOCKER_REGISTRY-}logapi
    build:
      context: .
      dockerfile: services/log/Log.API/Dockerfile
# Configuracao da imagem do microsservico de Leitura
  read.api:
    image: ${DOCKER_REGISTRY-}readapi
    build:
      context: .
      dockerfile: services/read/Read.API/Dockerfile
# Configuracao da imagem do microsservico de Telemetria
  telemetry.api:
    image: ${DOCKER_REGISTRY-}telemetryapi
    build:
      context: .
```

```
      dockerfile: services/telemetry/Telemetry.API/Dockerfile
```

A.9 Configuração de integração contínua dos microsserviços e serviços *Serverless*

```
# Branch na qual a integracao continua sera inicializada
trigger:
  branches:
    include:
    - master

# Maquina virtual onde sera realizada a integracao continua
pool:
  vmImage: 'ubuntu-latest'

# Repositorio atual do arquivo yaml
resources:
- repo: self

# Variaveis de ambiente para a etapa de steps
variables:
  buildConfiguration: 'Release'
  restoreBuildProjectFunction: '**/IoT.Function.Trigger.csproj'
  restoreBuildProjectFunctionTest: '**/IoT.Function.Trigger.*[Tt]ests/*.
    csproj'
  azureSubscriptionEndpoint: azure-resource
  azureContainerRegistry: rfidregistry.azurecr.io

# Etapa de complicacao e restauracao dos projetos (Serverless e containers)
steps:
# Restauracao do projeto Azure Function
- task: DotNetCoreCLI@2
  displayName: Restore
  inputs:
    command: restore
    projects: '$(restoreBuildProjectFunction)'

# Compilacao do projeto Azure Function
- task: DotNetCoreCLI@2
  displayName: Build
  inputs:
    projects: '$(restoreBuildProjectFunction)'
    arguments: '--configuration $(BuildConfiguration)'

```

```
39 # Execucao de testes do projeto Azure Function
40 - task: DotNetCoreCLI@2
41   displayName: Test
42   inputs:
43     command: test
44     projects: '$(restoreBuildProjectFunctionTest)'
45     arguments: '--configuration $(buildConfiguration) --collect "Code
     coverage"'
46
47 # Compilacao para arquivo ZIP do projeto Azure Function
48 - task: DotNetCoreCLI@2
49   displayName: Publish
50   inputs:
51     command: publish
52     arguments: '--configuration $(BuildConfiguration) --output $(Build.
     ArtifactStagingDirectory)'
53     projects: '$(restoreBuildProjectFunction)'
54     publishWebProjects: false
55     modifyOutputPath: true
56     zipAfterPublish: true
57
58 # Publicacao no arquivo ZIP do projeto Azure Function
59 - task: PublishBuildArtifacts@1
60   displayName: 'Publish Artifact'
61   inputs:
62     pathtoPublish: '$(Build.ArtifactStagingDirectory)'
63
64 # Compilacao dos projetos em containers
65 - task: DockerCompose@0
66   displayName: Build services
67   inputs:
68     action: Build services
69     azureSubscriptionEndpoint: $(azureSubscriptionEndpoint)
70     azureContainerRegistry: $(azureContainerRegistry)
71     dockerComposeFile: '**/docker-compose.yml'
72     projectName: $(Build.Repository.Name)
73     qualifyImageNames: true
74     includeLatestTag: true
75     additionalImageTags: $(Build.BuildId)
76
77 # Publicacao dos projetos em containers
78 - task: DockerCompose@0
79   displayName: Push services
80   inputs:
81     action: Push services
```

```
    azureSubscriptionEndpoint: $(azureSubscriptionEndpoint)
    azureContainerRegistry: $(azureContainerRegistry)
    dockerComposeFile: '**/docker-compose.yml'
    projectName: $(Build.Repository.Name)
    qualifyImageNames: true
    includeLatestTag: true
    additionalImageTags: $(Build.BuildId)
```

A.10 Configuração de integração contínua da aplicação *front-end*

```
# Branch na qual a integracao continua sera inicializada
trigger:
  branches:
    include:
    - master

# Maquina virtual onde sera realizada a integracao continua
pool:
  vmImage: 'ubuntu-latest'

# Repositorio atual do arquivo yaml
resources:
- repo: self

# Variaveis de ambiente para a etapa de steps
variables:
  azureSubscriptionEndpoint: azure-resource
  azureContainerRegistry: rfidregistry.azurecr.io
  imageName: rfidreaderweb

# Etapa de complicacao e restauracao do projeto em container
steps:
# Compilacao do projeto em container
- task: Docker@1
  displayName: 'Build an image'
  inputs:
    command: build
    azureSubscriptionEndpoint: $(azureSubscriptionEndpoint)
    azureContainerRegistry: $(azureContainerRegistry)
    imageName: $(imageName)
    includeLatestTag: true

# Publicacao do projeto em container
```

```
- task: Docker@1
  displayName: 'Push an image'
  inputs:
    command: push
    azureSubscriptionEndpoint: $(azureSubscriptionEndpoint)
    azureContainerRegistry: $(azureContainerRegistry)
    imageName: $(imageName)
    includeLatestTag: true
```