Capítulo 3

Teoria e Prática de Microserviços Reativos: Um Estudo de Caso na Internet das Coisas

Cleber Santana[2,1], Leandro Andrade[1], Brenno Mello[1], José Sampaio[1], Ernando Batista[2], Cássio Prazeres[1]

[1]Departamento de Ciência da Computação (DCC), UFBA

[2]Instituto Federal de Educação, Ciência e Tecnologia da Bahia (IFBA)

(leandrojsa, brenno.mello, jose.sampaio prazeres)@ufba.br,

dsandrade@dcc.ufba.br, (cleberlira, enrnando.passos)@ifba.edu.br

Abstract

Microservices has recently been employed at Cloud Computing to support the construction of large-scale systems that are resilient, resilient, and better tailored to meet today's demands. In the Internet of Things (IoT) a set of intelligent applications can be built in many different scenarios and that can impact the daily routine of people's lives. The development of applications and services at IoT brings challenges such as deployment, elasticity and resilience. Microservices implemented with resilience, elasticity, and message-driven features are considered reactive microservices. Therefore, in this chapter we introduce the concept of Reactive Microservices and illustrate a case study for building an IoT application.

Resumo

Microservices recentemente tem sido empregado na Cloud Computing para suportar a construção de sistemas de larga escala que sejam resilientes, elásticas e melhor adaptados para atender as demandas atuais. Na Internet das Coisas (IoT) um conjunto de aplicações inteligentes podem ser construídas nos mais diferentes cenários e que podem impactar na rotina diária da vida das pessoas. O desenvolvimento de aplicações e serviços na IoT traz desafios como implantação, elasticidade e resiliência. Microserviços implementados com as caraterísticas de resiliência, elasticidade e dirigido a mensagens são considerados Microserviços reativos. Portanto, neste capítulo introduzimos o conceito de Microserviços reativos e ilustramos um estudo de caso para construção de uma aplicação IoT.

3.1. Introdução

Atualmente, algumas aplicações estão utilizando Microserviços em domínios como o da IoT e Mobile [Francesco et al. 2017] a fim de apoiar a construção de sistemas de grande porte que sejam mais robustos, resilientes, elásticos e melhor adaptados para atender as demandas atuais. Os Microserviços visam construir, gerenciar e projetar arquiteturas de pequenas unidades autônomas [Fowler and Lewis 2014]. As aplicações baseadas na arquitetura de Microserviços são poliglotas, ou seja, são construídas a partir da linguagem de programação da escolha do desenvolvedor e as implementações são realizadas continuamente [Humble and Farley 2011], algumas dezenas de vezes por dia [Trojak 2018] ou em alguns casos centenas de vezes [BLOOM 2013], tornando o uso dessas aplicações mais dinâmica pelos inúmeros usuários.

Os Microserviços são arquiteturas nativas da nuvem, e as aplicações baseadas nesse estilo arquitetural são projetadas para serem resilientes a incidentes e interrupções na infraestrutura, no entanto, em alguns cenários esses aplicativos podem falhar e ficar indisponíveis por horas e, em alguns casos, dias CircleCI [CI 2015]), que tem um impacto negativo na cadeia produtiva dessas organizações.

Na IoT, o número de dispositivos atualmente é de 8,3 bilhões e estima-se que até 2025 esse número chegue a 21,5 bilhões [Lasse Lueth 2018]. Esses novos dispositivos poderão interagir fornecendo novos serviços e aplicações em diferentes domínios e ambientes, gerando valor agregado para o usuário. Os dispositivos da IoT, juntamente com suas tarefas, constituem aplicações específicas de domínio (por exemplo, cidades inteligentes, casas inteligentes, transportes inteligentes, agricultura de precisão) e mercados horizontais (por exemplo, computação ubíqua e serviços analíticos), que constituem serviços independentes de domínio [Shahid and Aneja 2017]. Por exemplo, no cenário da piscicultura, o monitoramento de variáveis (por exemplo, hidrológicas, hidráulicas e de qualidade) pode melhorar o cultivo das espécies. Nesse cenário, os Microserviços podem ser distribuídos na borda da rede, permitindo elasticidade e resiliência ao sistema [de Santana et al. 2019]. Resiliência é a capacidade de resistir a falhas externas e internas; e elasticidade refere-se à capacidade de resposta do sistema de acordo com a variação da demanda [da Rosa Righi et al. 2018]. O sistema deve usar mensagens assíncronas para garantir padrões de comunicação de baixo acoplamento, isolamento e transparência de localização para [Bonér 2017]. A conformidade com essas propriedades transforma os Microserviços em Microserviços Reativos [de Santana et al. 2019].

Os Microserviços reativos podem aumentar a confiabilidade das aplicações. A confiabilidade é um requisito prioritário para aplicações IoT que estão preocupadas com qualidade de serviço (QoS) [White et al. 2017]. Confiabilidade refere-se à capacidade de um sistema executar funções sob condições especificadas por um período de tempo especificado [for Standardization/International Electrotechnical Commission et al. 2011]. Além disso, a confiabilidade leva em consideração quatro sub-características, tais como: **Maturidade**, que é a capacidade de uma aplicação evitar falhas decorrentes de defeitos na aplicação; **Disponibilidade**, a capacidade de uma aplicação estar operacional e acessível quando necessário para uso; **Tolerância a falhas**, a capacidade de uma aplicação de operar como pretendido apesar da presença de falhas de hardware ou software e **Capacidade de recuperação**, que se refere à capacidade de uma aplicação recuperar os dados direta-

mente afetados, no caso de falha, e restabelecer ao estado desejado da aplicação.

As aplicações IoT geralmente são distribuídas em dispositivos móveis e servidores implantados na Cloud Computing/Fog Computing/ Edge Computing. Nesse contexto, uma aplicação IoT pode falhar por diferentes motivos: falhas de travamento, em que um dispositivo ou servidor para e precisa de uma reinicialização; falhas de omissão, em que um dispositivo ou servidor para de enviar e receber mensagens; falhas de temporização, em que uma resposta de dispositivo ou servidor está muio lenta [White et al. 2017].

Espera-se que as aplicações IoT evoluam continuamente para lidar com novos serviços. As arquiteturas tradicionais limitariam a capacidade de um sistema IoT de evoluir, já que as mudanças exigiriam o reinício do sistema [Dragoni et al. 2017]. Portanto, isso afeta a disponibilidade de aplicações IoT.

Avaliação de desempenho para aplicações IoT ainda é uma questão aberta [Udoh and Kotonya 2018]. Para obter confiabilidade, o sistema deve atender a padrões de desempenho, uma vez que isso pode afetar a disponibilidade das aplicações IoT. Nesse contexto, consideramos que essas falhas afetam a disponibilidade das aplicações IoT, onde arquiteturas tradicionais teriam dificuldades em lidar com esses problemas. Portanto, isso leva à necessidade de projetar esses aplicações com novas abordagens arquiteturais. Em estudos recentes, [de Santana et al. 2018] afirmaram que Microserviços foi aplicado em aplicações IoT e eles podem ser implantados em containers. Sistemas de gerenciamento de containers, como o Docker[1], e sistemas de orquestração, como o Kubernetes[2], controlam aplicações e provisionam dinamicamente seus recursos, que podem ser extremamente escalonáveis, confiáveis e reativos.

Nesse contexto, este capítulo apresenta uma arquitetura baseada em Microserviços [de Santana et al. 2019] para suportar o desenvolvimento de aplicações IoT reativas. Para prover o desenvolvimento dessas aplicações, nós também apresentamos uma plataforma baseada no Vert.x[3] que pode ser implantada em dispositivos (i.e com baixo poder computacional) e servidores que estejam localizados na borda da rede, na névoa ou nuvem.

3.2. Infraestrutura para Internet das Coisas

Antes da IoT, os dados produzidos na Internet, em sua grande maioria, eram dependentes de humanos. Este fato determina algumas características estão associadas a natureza humana, como limitação de tempo, atenção e acurácia. Por sua vez, essas características possuem diferenças em dados produzidos por coisas, as quais tem seu comportamento determinado por um programa e seus componentes eletrônicos. Kevin Ashton [Ashton 2009], o pesquisador que introduziu o termo "Internet of Things" em 1999, argumentou que tecnologia de sensores e *tags* RFID podem ser capazes de rastrear e processar informações reduzindo significativamente perdas e custos. Desde então, as tecnologias para implantar coisas na Internet foram desenvolvidas, como conectividade sem fio, protocolos de comunicação, hardware para dispositivos e plataformas para prototipagem e desenvolvimento de sistemas eletrônicos, criando assim muitas possibilidades de aplicações e soluções para IoT [Gérald 2010].

[1]https://www.docker.com/get-started

[2]https://kubernetes.io/pt/

[3]https://vertx.io/

Conceitos e tecnologias inspirados na IoT foram desenvolvidos em várias aplicações do mundo real, como, por exemplo: carros autônomos, gerenciamento eficiente de energia, gerenciamento de ambientes inteligentes, gerenciamento inteligente de tráfego e monitoramento ambiental.

De acordo com *The Internet of Things – Architecture* (IoT-A) [Bauer et al. 2013], do ponto de vista da IoT existem três tipos básicos de dispositivos:

- **Sensores:** fornecer informações, conhecimentos ou dados sobre a entidade física monitorada. Por exemplo, um dispositivo mede a temperatura de uma sala ou a câmera habilitada para reconhecimento de rosto são sensores. Os dados produzidos pelos sensores funcionam como entradas para sistemas IoT e também podem ser gravados para recuperação posterior;
- ***Tags*:** são aplicados para identificar dispositivos físicos em sistemas IoT. Geralmente, as tags são fisicamente anexadas, como códigos de barras, códigos QR e RFID. Eles são usados para melhorar a automatização da identificação de dispositivos IoT pelos sistemas IoT;
- **Atuadores:** podem alterar o estado físico dos dispositivos IoT, como ligar/desligar, ações de movimento ou alterar o estado da forma. Os atuadores, em geral, agem em ambientes ou coisas físicas, e sua ação geralmente promove mudanças no local/coisa da operação. Um dispositivo IoT capaz de ligar/desligar luzes ou braço mecânico em uma fábrica de automóveis são exemplos de atuadores.

O desenvolvimento da IoT depende do projeto de novas aplicações e modelos de negócios. Em estudos recentes, [Khan et al. 2012] e [Al-Fuqaha et al. 2015] dividiram a estrutura da IoT em cinco camadas. A Figura 3.1 ilustra essas camadas, que são descritas abaixo:

- **Camada de Percepção:** contém as "coisas" da IoT, que são sensores (por exemplo, sensores de temperatura), atuadores (por exemplo, relés) e tags (por exemplo, RFID), conforme ilustrado na Figura 3.1. Esses dispositivos são a base da infraestrutura da IoT, a partir da qual os sistemas interagem com o ambiente físico;
- **Camada de Conexão:** transfere as informações da camada Percepção para a camada de Middleware. Nos sistemas de IoT, geralmente a transmissão pode ser sem fio ou com fio. Além disso, podemos usar a tecnologia 4G, Bluetooth e infravermelho, dependendo do sensor, conforme mostrado em Figura 3.1;
- **Camada de Middlware:** fornece uma interface entre a camada de percepção (através da camada de Conexão) e o restante do sistema de IoT. Essa camada é responsável pelo gerenciamento de dispositivos IoT e pelos dados coletados. O middleware também fornece interface de acesso aos dados da camada superior da IoT;
- **Camada de Aplicação:** fornece o gerenciamento global do sistema de IoT, recebendo e enviando informações para dispositivos e usuários. A Figura 3.1 mostra a camada de Aplicação através de exemplos de aplicação como saúde inteligente,

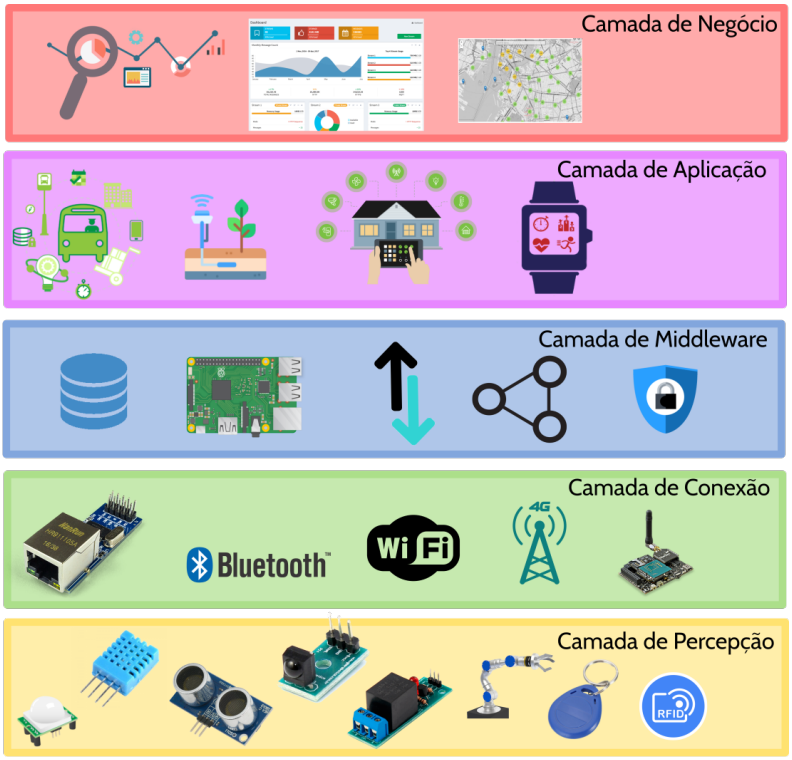


Figure 3.1. Representação de camadas da arquitetura IoT. Adaptado de [Andrade et al. 2018]

agricultura inteligente, casa inteligente, cidade inteligente, transporte inteligente, etc;

- **Camada de Negócio:** gerencia as atividades e serviços gerais do sistema IoT [Al-Fuqaha et al. 2015]. Como é ilustrado na Figura 3.1, essa camada é responsável por fornecer modelo de negócios, gráficos, fluxogramas e outras visualizações com base nos dados recebidos da camada Aplicação.

Os progressos na IoT resultaram na criação de vários ecossistemas paralelos, sem uma integração global [Sundmaeker et al. 2010]. Essa limitação dificulta o acesso a diferentes dispositivos e a criação de aplicações mais poderosos, integrando sistemas distintos. O Cluster Europeu de Pesquisa sobre a Internet das Coisas (IERC) publicou um relatório [Serrano et al. 2015a] analisando os principais desafios de pesquisa, melhores práticas, recomendações e próximos passos para proporcionar interoperabilidade na Internet das Coisas.

Além da necessidade de fornecer interoperabilidade, as soluções IoT geralmente requerem funcionalidades para registrar, anotar para gerenciar dados e dispositivos [Serrano et al. 2015b]. Além disso, os serviços IoT devem garantir a entrega e o uso desses dados para usuários, aplicações ou até outros serviços [Bassi et al. 2013]. Várias soluções baseadas em Computação em Nuvem foram propostas pela indústria e pela academia com foco na prestação de serviços e interoperabilidade. Na Seção 3.2.1 é discutido os principais aspectos das soluções de IoT baseadas na Computação em Nuvem.

A crescente demanda por mais eficiência no processamento local e mais maneiras de proteger os dados e a tecnologia IoT antes dos dados irem para a nuvem geraram novas alternativas de infraestrutura em sistemas IoT. Assim, alternativas surgiram buscando usar a capacidade de processamento, armazenamento e acesso dos dispositivos locais, na chamada Fog Computing ou Computação em Névoa. Na Seção 3.2.2, apresentamos características sobre as soluções de IoT baseadas no paradigma de Computação em Névoa. Por fim, algumas soluções IoT não utilizam conectividade externa para prover seu funcionamento, usando de modo exclusivo os dispositivos locais na chamada Computação na Borda (Edge Computing). Os detalhes sobre esse tipo de infraestrutura é apresentado na Seção 3.2.3.

3.2.1. Computação em Nuvem

Com evolução da Internet com alta demanda como segurança, armazenamento e processamento de dados, a Cloud Computing ou Computação em Nuvem transforma uma solução amplamente usada para esses problemas. O conceito de Computação em Nuvem refere-se a uma extensão da computação em grade, computação distribuída e computação paralela com um ambiente de virtualização e extensibilidade [Zhang et al. 2010].

As soluções de Computação em Nuvem fornecem um modelo que permite o acesso configurável dos recursos computacionais oferecidos como serviços [Borgia 2014]. Em geral, o gerenciamento desses recursos é controlado pelos usuários, onde eles pagam apenas o valor do uso efetivo, através do modelo *pay-as-you-go* [Cavalcante et al. 2016]. Esse tipo de solução é usado pela maioria das aplicações Web, pois fornece aspectos como alta disponibilidade, tolerância a falhas e escalabilidade de processamento e armazenamento.

Embora os recentes avanços nos dispositivos IoT, a computabilidade e o armazenamento sejam recursos limitados na maioria deles [Botta et al. 2016]. A associação do paradigma de Computação em Nuvem pode suportar esses tipos de limitação da IoT porque a nuvem dispõe de uma infraestrutura global de alta capacidade de computabilidade e armazenamento, como exemplo de processamento de uma análise de dados de grande volume de dados. Além disso, essa associação pode melhorar aspectos como privacidade, desempenho e confiabilidade de plataformas IoT.

Muitas iniciativas estão desenvolvendo arquiteturas e plataformas para a IoT usando o paradigma de Computação em Nuvem, em que o acesso e os dados gerados pelos sensores IoT são direcionados para a nuvem. Por exemplo, grandes provedores dessa tecnologia, como Amazon e Google, oferecem serviços em nuvem com suporte específico e especial aos sistemas de IoT [Cavalcante et al. 2016]. Além disso, existem várias soluções de plataformas em nuvem capazes de oferecer alta escalabilidade, armazenamento e processamento, bancos de dados distribuídos, processamento em tempo real, gerenciamento, monitoramento e implantação [Díaz et al. 2016].

A Figura 3.2 mostra uma visão geral do paradigma de Computação em Nuvem para a Internet das Coisas: Cloud of Things (CoT) [Distefano et al. 2012]. No paradigma CoT, os dados de coisas geralmente são coletados por um middleware de sensores e enviados para uma nuvem pública para processamento, armazenamento e entrega de serviços. Não há processamento de dados ou entrega de serviços na borda da rede, porque todos os

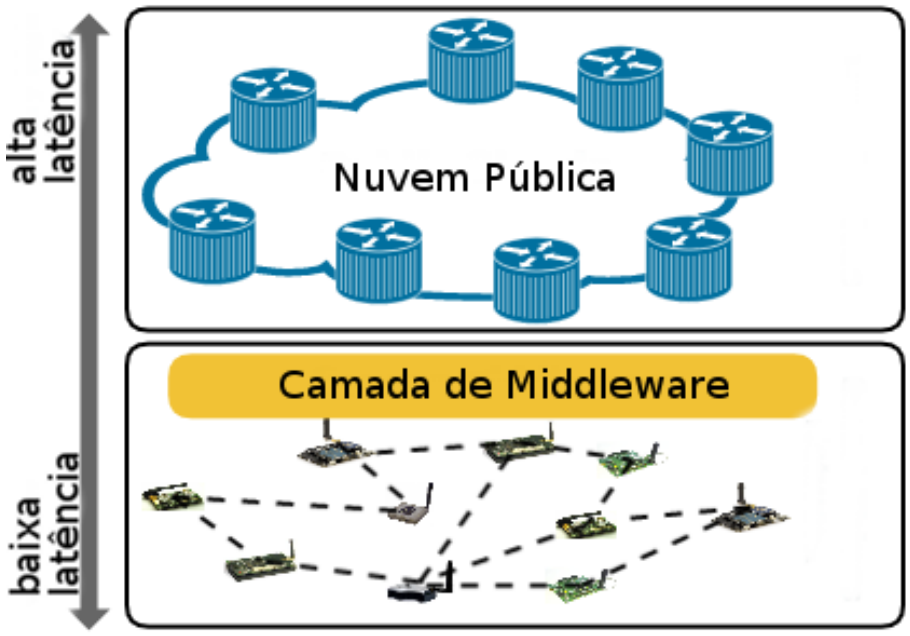


Figure 3.2. Visão Geral da Nuvem de Coisas

dados são processados por servidores remotos na nuvem.

Existem alguns desafios na integração da IoT e da computação em nuvem como a necessidade de melhorar a eficiência dos recursos da nuvem (por exemplo: rede, processamento e armazenamento), padronização de dados e serviços, segurança e privacidade de dados, preocupação com a confiabilidade (os dispositivos IoT podem ficar indisponíveis por diversas razões) e alta heterogeneidade dos ambientes IoT e de nuvem [Cavalcante et al. 2016]. De acordo com [Abdelshkour 2015] o paradigma CoT tem algumas limitações como: a conectividade com a nuvem é uma precondição, mas alguns sistemas de IoT precisam funcionar, mesmo quando a conexão está temporariamente indisponível; alta demanda por largura de banda, como resultado do envio de todos os dados pelos canais da nuvem; e, tempo de resposta lento (alta latência) e escalabilidade limitada como resultado da dependência de servidores remotos.

Nesse contexto, algumas alternativas foram propostas para evitar algumas limitações do paradigma da Computação em Nuvem. Na Seção 3.2.2, apresentamos o Fog Computing ou Computação em Névoa, que estende o paradigma de Computação em Nuvem usando a capacidade de processamento, armazenamento e comunicação dos dispositivos da rede local e tem sido adequado aos requisitos de sistemas IoT.

3.2.2. Computação em Névoa

Com o objetivo de criar alternativas para limitação da Computação em Nuvem, [Bonomi et al. 2012] propuseram o paradigma Fog Computing (ou Computação em Névoa), que, por definição, aproxima algumas operações de computação e armazenamento da borda da rede. Na Computação em Névoa, as operações são distribuídas entre dispositivos computacionais, transferindo parte da complexidade da nuvem para a borda.

Contudo, a adoção da Computação em Névoa não exclui a Computação em Nu-

vem. Esta traz características como baixa latência, reconhecimento de local, suporte à mobilidade, forte presença de aplicativos de streaming e em tempo real e escalabilidade para grande número de nós de nevoeiro [Bonomi et al. 2012]. Além disso, por motivos como requisitos de segurança ou privacidade e/ou indisponibilidade de um servidor em nuvem. A Computação em Névoa e a Computação em Nuvem são complementares em vários aspectos, pois os dispositivos de borda continuam a operar mesmo sem conectividade e, quando é possível, os dados são enviados para a nuvem. Além disso, usuários ou aplicativos locais podem acessar dispositivos e dados diretamente na rede interna e quando estes são remotos podem acessar através de servidores em nuvem, proporcionando melhor Qualidade de Experiência (QoE).

Em uma típica implementação da IoT, os dados coletados dos dispositivos são armazenados e processados em servidores na nuvem. Embora esse tipo de abordagem seja comumente usado, ele tem algumas limitações [Abdelshkour 2015]: a conectividade com a nuvem é uma pré-condição e alguns sistemas de IoT precisam funcionar, mesmo quando a conexão está temporariamente indisponível; alta demanda por largura de banda, como resultado do envio de todos os dados pelos canais da nuvem; tempo de resposta lento (alta latência) e escalabilidade limitada como resultado da dependência de servidores remotos hospedados em data centers centralizados.

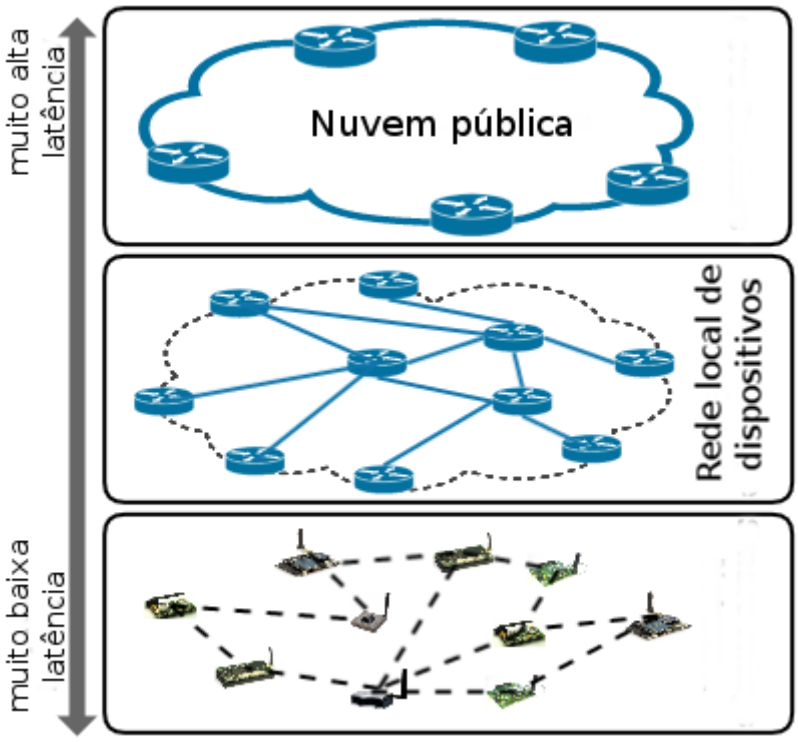


Figure 3.3. Visão geral da IoT na Computação em Névoa.

Na IoT, a Computação em Névoa visa aproveitar um pouco da complexidade da nuvem e aproximá-la de dispositivos, aplicações e/ou usuários, como uma "nuvem local e privada". A Figura 3.3 ilustra a visão geral de uma rede de equipamentos físicos (servidores, gateways, dispositivos etc.) com capacidades para processamento de dados locais e paralelos, prestação de serviços para IoT e habilitado para enviar dados resultantes desse

processamento para infraestruturas virtuais (nuvens públicas), a fim de atender aos requisitos e características da Computação em Névoa como descrito anteriormente.

Um caso que exemplifica tal associação é Névoa das Coisas (Fog of Things - FoT), na qual a Prazeres and Serrano [Prazeres and Serrano 2016] propuseram uma nova maneira de projetar e implementar plataformas IoT usando Computação em Névoa. O paradigma FoT vai além da Computação em Névoa em algumas direções, como:

- Usando toda a capacidade de processamento da borda da rede, executando o processamento de dados e a entrega de serviços em dispositivos, gateways (servidores muito pequenos) e pequenos servidores locais;
- Definindo perfis para gateways e servidores na borda da rede, a fim de definir os serviços de IoT a serem entregues;
- Distribuindo esses serviços IoT na borda da rede por meio de um middleware orientado a mensagens e serviços.

Por fim, no paradigma FoT, parte da capacidade de processamento de dados e das operações de entrega de serviços são processadas localmente em pequenos servidores combinado com operações em servidores na nuvem [Andrade et al. 2018].

3.2.3. Computação na Borda

A Computação em Borda ou Edge Computing refere-se às tecnologias que permitem que a computação e armazenamento seja executada na borda da rede, para que tais processos aconteça perto de fontes de dados [Shi and Dustdar 2016]. A adoção de tal paradigma prove benefícios como: baixa latência, mobilidade dos dispositivos, segurança e independência de conexão com a Internet.

Existem sistemas IoT que possuem sua funcionalidade e ciclo de vida envolvendo somente dispositivos locais. Esse tipo de solução é baseado na Computação na Borda restringe ao recursos para operação dos sistemas a servidores e dispositivos locais. Em alguns caso tal tipo de situação pode ocorrer de modo temporário, quando há perda de conectividade com a internet, na qual o sistema pode se adaptar e funcionar somente com seus recursos internos.

A Computação na Borda permite um grande número de aplicações, incluindo comunicação veicular, cidades inteligentes, smart grid, redes de sensores sem fio incorporadas a atuadores, monitoramento de tráfego rodoviário, monitoramento de tubulações, parques eólicos, sistema de semáforo inteligente, monitoramento ferroviário, sistemas de controle industrial e o aplicações em explorações de petróleo e gás [Bilal et al. 2018].

3.2.4. Contêiners

As aplicações IoT podem ser disponibilizadas na Nuvem/Névoa/Borda a partir da adoção de contêiners. Os ontêiners fornecem isolamento das aplicações, permitindo realizar atualizações e escalar as aplicações. As duas tecnologias de container a seguir (Kubernetes e Docker) tem sido adotadas na implantação de aplicações IoT.

3.2.4.1. Kubernetes

O Kubernetes é uma plataforma de automação para implantar, dimensionar e operar contêiners das aplicações por meio de clusters de host, e foram originalmente criados pelo Google [kubernetes 2019]. O Kubernetes trabalha com um conceito semelhante de Nós. Não há hierarquia de Nós: cada Nó um é uma instância capaz de executar tarefas em contêiners. Os Nós são controlados por um Nó controlador, responsável por monitorar os Nós e manter a lista de Nós em sincronia com as máquinas do cluster.

Para implantar as aplícações IoT no Kubernetes é possível utilizar o Openshift. O Openshift é uma plataforma de conteinerização. O Openshift possui entidades, tais como: **configuração de build**, **configurações de deployment**, **Pods**, **Serviços e Rotas**.

A configuração de build é a parte do processo que se constrói as imagens conteinerizadas que serão usadas pelo Openshift para instanciar os diferentes contêiners que compõem a aplicação. Esta "entidade" pode trabalhar com o próprio docker, por exemplo, construindo a imagem do Dockerfile.

A configuração de Deployment é a parte do processo em que se define a instância da imagem construída pelo build, definindo qual imagem deve ser criada e quantas instâncias devem ser mantidas funcionando, quando a implementação de um container deve ser feita, quantas réplicas devem existir, atuando como um controlador de réplicas programável para ser tanto ajustável manualmente quanto para realizar o auto-scaling e também serve para realizar checagens da 'saúde' das réplicas para que não exista uma incidência de containeres inoperantes.

Os Pods são um grupo de um ou mais contêiners, porém normalmente são compostos por somente um container. A orquestração, escala e administração dos Pods é delegada ao Kubernetes.

Os serviços nos permitem comunicar com os pods sem depender de seus endereços, mas usando o endereço virtual do serviço. Um serviço atua como um proxy na frente de um grupo de pods e podem implementar uma estratégia de balanceamento de carga.

Para instalar o Openshift utiliza-se o Minishift:

```
https://github.com/minishift/minishift
```

O Minishift requer um hypervisor para executar a máquina virtual que contém o OpenShift. Dependendo do sistema operacional utilizado, pode-se escolher entre algumas alternativas de hypervisors (consulte o guia de instalação do Minishift para obter detalhes). Para instalar o Minishift, basta baixar o arquivo mais recente para o sistema operacional utilizado pelo usuário na página de downloads do Minishift:

```
https://github.com/minishift/minishift/releases
```

Descompacte o arquivo no local de preferência e adicione ao PATH. Assim que estiver instalado basta usar o comando:

```
minishift start
```

A partir daí será possível conectar à sua instância do OpenShift no endereço

```
https://192.168.64.12:8443
```

Caso seja necessário validar o SSL faça login como developer/developer

Também há necessidade de um openshift client, disponível em:

```
https://github.com/openshift/origin/releases/tag/v3.11.0
```

Descompacte na área de preferência e adicione o binário ao PATH, conecte-se ao openshift com o comando:

```
oc login https://192.168.64.12:8443 -u developer -p developer
```

Para criar um projeto use os seguintes comandos:

```
oc new-project nome-projeto

oc policy add-role-to-user admin developer -n nome-projeto

oc policy add-role-to-user view -n nome-do-projeto -z default
```

E acesse-o em:

```
open https://192.168.64.12:8443/console/project/<nome do projeto>
```

E pode-se observar a tela de projeto. Usando um plugin do maven, se criam os pods com o comando:

```
mvn fabric8:deploy -Popenshift
```

O primeiro build toma um pouco mais de tempo, mas as próximas versões serão mais rápidas pois muitos dados estarão em cache.

3.2.4.2. Docker

O Docker é um projeto de código aberto e seu objetivo é criar e manter contêiners [Bernstein 2014]. É responsável por armazenar vários serviços isoladamente do Sistema Operacional, como: servidor web, banco de dados, aplciações, entre outros. Seu back-end é baseado em Linux Contêiners.

Os containers do Docker são criados a partir de imagens. Uma imagem pode incluir apenas os fundamentos do sistema operacional ou pode consistir em uma pilha de aplicações pré-criadas. Ao construir as imagens, cada comando executado (por exemplo, apt-get install) forma uma nova camada. Os comandos podem ser executados manual ou automaticamente a partir do script Dockerfiles.

Dockerfile é um script composto por vários comandos listados sucessivamente, para que seja possível executar ações em uma imagem base para criar uma nova imagem. O Dockerfile é usado para organizar artefatos e simplificar o processo de implantação.

A opção de utilizar o Docker para implantar Microserviços vem ganhando espaço. Isso se deve à fácil escalabilidade, isolamento e facilidade de compartilhamento das imagens utilizadas pela plataforma (Se houver alguma dúvida com relação aos termos utilizados, consulte:

```
https://docs.docker.com/get-started/
```

Por quê utilizar o Docker em vez de uma VM com Open Shift? Existem diversas respostas para essa pergunta, algumas podem ser encontradas em: Mas elas revolvem basicamente em torno de três pontos: 1- Limitações de Hardware: Enquanto uma VM utiliza um sistema operacional "convidado" ou "hóspede" com acesso virtual ao sistema operacional "anfitrião" pelo Hypervisor, os containers do docker rodam nativamente no Linux, compartilhando o kernel com o sistema operacional "anfitrião" e rodando processos discretos, sem utilizar muita memória, se tornando uma opção mais leve do que as VM's que geralmente consumem bastante memória.

2- Facilidade de Cooperação: Um daemon do Docker pode ser conectado tanto no mesmo sistema quanto em outros daemons em outros lugares, permitindo assim que mais pessoas trabalhem num projeto só de seus computadores. Quado isso acontece, utiliza-se o Docker Swarm, onde existem diversos trabalhadores e alguns gerentes, e todos os membros do projetos são Docker daemons que se comunicam pela API do Docker, deixando assim o trabalho mais dinâmico e funcionando um pouco como um Github, onde se poderiam dar push e pull de várias imagens do projeto.

3- Facilidade de Remodelar/Reconstruir o projeto: Além da facilidade de comunicação e teste do projeto entre seus desenvolvedores, os containers do Docker são efêmeros e facilmente destruídos ou remodelados, então ao encontrar um erro ao invés de precisar interromper toda a maquina para consertá-lo, só precisaríamos interromper o container específico que contém a imagem problemática, consertá-la e rodar novamente outro container, atualizado.

É de suma importância que sejam instalados tanto o docker quanto o docker compose, que seja utilizado um ambiente rodando alguma distribuição do linux (neste capítulo foi utilizado o ubuntu 18.04) e possuir o git instalado.

Primeiramente é necessário que haja uma aplicação para realizar o deployment no docker.

```
https://github.com/Rck-Sanchez/microservices-in-docker
```

Para baixar o repositório basta utilizar o comando:

```
git clone https://github.com/Rck-Sanchez/microservices-in-docker.git
```

Tutorial de instalação para o docker (versão gratuita):

Primeiro, atualize sua lista atual de pacotes com o comando:

```
sudo apt update
```

Em seguida, instale alguns pacotes de pré-requisitos que permitem que o apt utilize pacotes via HTTPS:

```
sudo apt install apt-transport-https ca-certificates
curl software-properties-common
```

Então adicione a chave GPG para o repositório oficial do Docker em seu sistema:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg sudo apt-key
    add -
```

Adicione o repositório do Docker às fontes do APT:

```
sudo add-apt-repository deb arch=amd64
https://download.docker.com/linux/ubuntu bionic stable"
```

A seguir, atualize o banco de dados de pacotes com os pacotes Docker do repositório recém adicionado:

```
sudo apt update
```

Certifique-se de que a instalação se dará a partir do repositório do Docker em vez do repositório padrão do Ubuntu:

```
apt-cache policy docker-ce
```

Espera-se uma saída como esta, embora o número da versão do Docker possa estar diferente:

```
Output of apt-cache policy docker-ce
docker-ce Installed (none) Candidate 18.03.1-ce-3-0-ubuntu
Version table 18.03.1-ce-3-0-ubuntu 500 500
https://download.docker.com/linux/ubuntu bionic/stable amd64 Packages
```

Instale o Docker:

```
sudo apt install docker-ce
```

O Docker agora deve ser instalado, o daemon iniciado e o processo ativado para iniciar na inicialização. Verifique se ele está sendo executado:

```
sudo systemctl status docker
```

A saída deve ser semelhante à seguinte, mostrando que o serviço está ativo e executando:

```
Output docker.service-Docker Application Container
Engine Loaded
loaded /lib/systemd/system/docker.service enabled
vendor preset: enabled) Active active (running)
since Thu 2018-07-05 15:08:39 UTC 2min 55s ago
Docs https://docs.docker.com Main PID: 10096 (dockerd)
Tasks: 16 CGroup: /system.slice/docker.service 10096
/usr/bin/dockerd -H fd:// 10113 docker-containerd
--config /var/run/docker/containerd/containerd.toml
```

Instalação do docker compose, necessário para rodar mais de um Microserviço em paralelo:

Primeiro checa-se a versão atual com o comando:

```
sudo curl -L https://github.com/docker/compose/releases/
download/1.21.2/
docker-compose- uname -s -uname -m -o

/usr/local/bin/docker-compose
```

Então veremos as permissões:

```
sudo chmod +x /usr/local/bin/docker-compose
```

Verificamos a instalação:

```
docker-compose --version
```

O resultado deve ser parecido com:

```
docker-compose version 1.21.2, build a133471
```

Os Dockerfiles são os arquivos que guiam o que o docker deve fazer com a imagem para colocá-la num container, logo é uma das partes mais importantes do projeto.

O Dockerfile deve ser criado como um arquivo de texto, o que pode ser feito em qualquer editor, e cada parte do Microserviço deve possuir um Dockerfile diferente, especificando o que cada uma dessas partes deve fazer. Por exemplo, na pasta disponibilizada no github existem: Dockerfile-msm e Dockerfile-mscm. Ao abri-los podemos ver a estrutura:

```
FROM frolvlad/alpine-java
WORKDIR ./files
EXPOSE 8081
COPY ./files/hello-microservice-1.0-SNAPSHOT.jar /var/lib/docker
ADD ./files/hello-microservice-1.0-SNAPSHOT.jar apa.jar
CMD ["java", "-jar", "apa.jar"]
```

O comando FROM especifica uma imagem para o docker usar como base, pré programadas com algumas funções, nesse caso com o jdk 8. O comando WORKDIR especifica onde que estão os arquivos a serem trabalhados. O comando EXPOSE diz em qual porta o mundo externo terá acesso ao container. O comando COPY adiciona o programa ao path de execução do docker. O comando ADD nesse caso está sendo utilizado somente para reduzir o tamanho do nome do arquivo .jar, mas sua função é parecida com a do comando COPY. O comando CMD é o responsável por executar o .jar dentro do container. Depois de seguir todos os passos anteriores e após fazer os Dockerfiles, é importante que se realize o comando docker build microservices-repo para criar os containers e verificar se não há nenhum erro com os seus respectivos Dockerfiles.

Para um Microserviço funcionar, deve existir um fluxo de informações que o usuário deve ser capaz de acessar a partir dele, e esse fluxo requer comunicação entre as partes que compõem a arquitetura do Microserviço. Assim sendo, temos as partes do Microserviço funcionando, porém cada uma por si só, isoladas.

Para criarmos elos entre as partes deve-se utilizar um outro produto do Docker, o Docker compose. Essa ferramenta estabelece ligações entre as partes dos microserviços além de estabelecer uma ordem de operação, se necessário for. Na pasta disponibilizada no github, pode-se observar o arquivo docker-compose.yml, este seria o equivalente ao Dockerfile do docker compose, sendo assim imprescindível para o deployment do Microserviço. É importante salientar aqui que a identação neste documento é imprescindível para o funcionamento correto do docker compose. O documento docker-compose.yml tem diversas diferenças em relação ao Dockerfile, como se pode observar no trecho a seguir:

```
version: '2.2'
services:
    microservicemessage:
```

```
        container_name: microservicemessage
        build:
            context: .
            dockerfile: Dockerfile-msm
        image: microservicemessage:latest
        expose:
            - 8080
        ports:
            - 8080:8080

    microservicemessageconsumer:
        container_name: microservicemessageconsumer
        build:
            context: .
            dockerfile: Dockerfile-mscm
        image: microservicemessageconsumer:latest
        expose:
            - 8081
        ports:
            - 8081:8081
        links:
            - microservicemessage:microservicemessage
        depends_on:
            - microservicemessage
```

Em version, coloca-se a versão instalada do docker-compose na máquina, pode-se verificá-la com o comando:

```
docker-compose version
```

Em services, declaramos quais serviços compõem e após nomeá-los dizemos o nome do container em que a imagem se encontra. O comando build é quem propriamente cria os elos entre os microserviços, lendo o que está no contexto, ele localiza os dockerfiles selecionados e suas respectivas imagens e cria, por meio do campo links, uma comunicação entre os microserviços. Em caso de existir alguma espécie de ligação entre um programa e outro, como acontece no exemplo, é necessário que se adicione o link, que diz de qual serviço o que está sendo declarado precisa, e estabelece uma ordem de iniciação dos containers com o comando depends-on, pois se o serviço depende de outro para funcionar, logicamente o serviço só pode operar quando seus pré-requisitos estiverem funcionando.

Após cumprir com a criação de todos os arquivos necessários, basta ir até o terminal, utilizar o comando cd para entrar na pasta microservices-repo onde estão as Dockerfiles do seu microserviço e, neste diretório, utilizar o comando:

```
docker-compose build
```

Que faz o processo de building dos containers.

Após isso, basta utilizar o comando

```
docker-compose up
```

e esperar até que apareçam as mensagens de succeeded in deploying verticle. Quando isso acontecer, seus programas já estão rodando e você deve ser capaz de acessá-los em localhost:8080.

3.3. Programação na IoT

Ainda em 2014, quando o setor de desenvolvimento de sistemas apontou os Microserviços como uma arquitetura promissora para soluções para problemas de escalabilidade, disponibilidade e implantação, [Namiot and Sneps-Sneppe 2014] já considerava os Microserviços (consulte a seção 3.3.3) como uma adoção natural para o desenvolvimento de aplicações em um ecossistema IoT. De acordo com [Taveras Núñez 2017], a implementação de soluções no ambiente da IoT requer conceitos avançados de programação, como multithreading, propagação de alterações e computação elástica. Portanto, esses conceitos podem ser fornecidos pela Programação Reactiva (consulte a Seção 3.3.1) e pelo Sistema Reativo (veja a Seção 3.3.2).

3.3.1. Programação Reativa

De acordo com [Bainomugisha et al. 2013], a programação reativa é um paradigma de programação preocupado com a observação de fluxos de dados e a propagação de mudanças. [Bonér 2017] afirma que a programação reativa é fundamental para o design de Microserviços, pois possibilita a criação de serviços eficientes, responsivos e estáveis.

Na programação reativa, os estímulos são os dados que transitam no fluxo, chamados de fluxos. Com a programação reativa, os componentes individuais de um sistema podem ter melhor eficiência e desempenho por meio de execuções assíncronas.

Na IoT, [Taveras Núñez 2017] afirma que esse paradigma representa um método adequado para orquestrar cálculos com base em eventos assíncronos, como os fluxos de dados dos sensores presentes na IoT.

A programação reativa é um modelo de desenvolvimento orientado pelo fluxo e propagação de dados. Na programação reativa, o estímulos são os dados que transitam no fluxo, chamados fluxos. Existem muitas maneiras de implementar um modelo de programação reativa. Este tipo de programação é importante pois, a partir dela, criam-se passagens assíncronas de dados e, ao orquestra-las, o programador é capaz de criar aplicações que são um pouco mais resistentes a erros e a partir dessa assincronicidade são criados os sistemas reativos, uma ferramenta poderosa para criar ambientes que usam o conceito de Microserviços de maneira reativa e, justamente por isso, possuem qualidades que são bastante atrativas para os desenvolvedores.

A Listagem 1 exemplifica um trecho de código que adota programação reativa. Nesse exemplo, linha 8 a 18, Nesse trecho, o código está observando um *Observable* e é notificado quando os valores transitam no fluxo.

3.3.2. Sistema Reativos

Sistemas reativos, por sua vez, são um estilo arquitetônico utilizado para construir sistemas distribuídos. Utilizar sistemas reativos é interessante pois, a partir deste estilo, é possível se alcançar responsividade, criando sistemas que mantém-se funcional mesmo sob intensa requisição ou até mesmo, sob a existência de erros.

A Figura 3.6 ilustra os princípios para a construção de um sistema reativo:

- Responsivo, refere-se à capacidade de um sistema responder consistentemente no

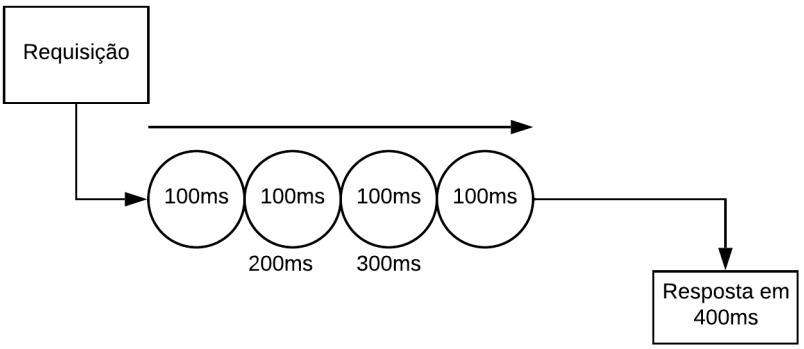


Figure 3.4. Diagrama representando o comportamento de um sistema síncrono

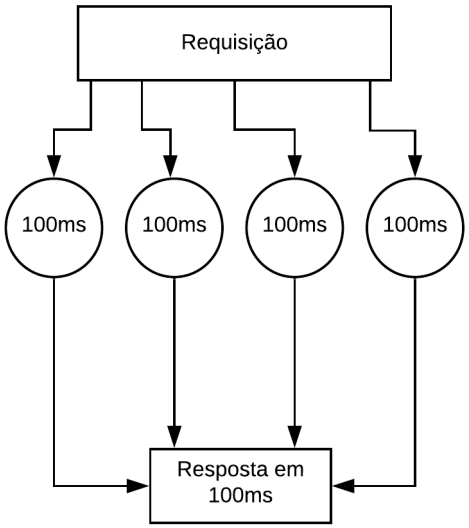


Figure 3.5. Diagrama representando o comportamento de um sistema assíncrono

menor tempo possível.

- Resiliência, esse princípio se aplica a sistemas que exigem alta disponibilidade. Um sistema resiliente responde mesmo na presença de falhas.
- Elasticidade, refere-se à capacidade de um sistema reagir adequadamente a variações de carga. Portanto, aumentar ou diminuir os recursos dependerá do número de solicitações ao sistema.

Em um sistema reativo, os componentes enviam e recebem mensagens assíncronas (consulte a Figura 3.6). Para dissociar remetentes e receptores, as mensagens são enviadas para um endereço virtual. Portanto, para que um componente receba uma mensagem, é necessário que ele esteja registrado nesse endereço virtual. Um endereço é um identificador do destino das mensagens e é representado por uma sequência ou URL. As in-

```
1  package io.vertx.book.rx;
2  import rx.Observable;
3  public class ObservableExample {
4
5  public static void main(String[] args) {
6  Observable<Integer> observable = Observable.range(0, 21);
7
8  observable.subscribe(
9      data -> {
10         System.out.println(data);
11     },
12     error -> {
13       error.printStackTrace();
14     },
15       () -> {
16             System.out.println("No more data");
17             }
18         );
19     }
20 }
```

Listing 1: Exemplo de um código contendo programação Reativa.

terações das mensagens assíncronas promovem duas propriedades: *Elasticidade*, que se refere à capacidade de escalar horizontalmente e *Resiliência*, que se refere à capacidade de permanecer responsivo em caso de falha e restauração. Além disso, melhora o tempo de resposta de um sistema, como pode ser observado nas Figuras 3.4 e 3.5.

3.3.3. Microserviços

Os Microserviços visam construir, gerenciar e projetar arquiteturas de pequenas unidades autônomas e são baseados na filosofia UNIX [Fowler and Lewis 2014]: os programas devem executar apenas uma tarefa e executar bem; os programas devem poder trabalhar juntos; os programas devem usar uma interface universal. Essas ideias levam a um design de componente reutilizável, suportando a modularização. O ponto principal é que os serviços são implantados no ambiente de produção independentemente um do outro, o que é uma das principais diferenças com a maioria das soluções tradicionais (e.g SOA) existentes.

A Figura 3.7 compara a arquitetura monolítica com a arquitetura de Microserviços. Na arquitetura monolítica, a lógica de negócios é processada em um único processo. Por esse motivo, uma atualização em qualquer parte da aplicação requer que todos as aplicações monolíticas sejam reimplementados. Em uma arquitetura de Microserviços, cada lógica de negócios é criada como um serviço independente. Assim, cada Microsserviço pode ser facilmente gerenciado em tempo de execução, incluindo várias atividades como implantação, instanciação e replicação.

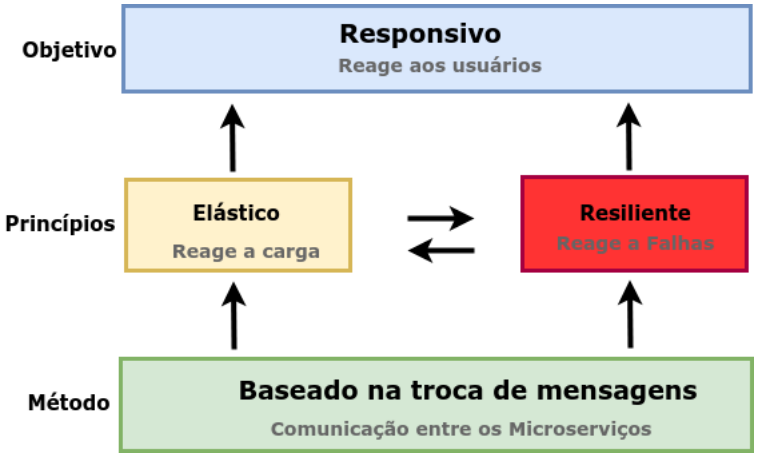
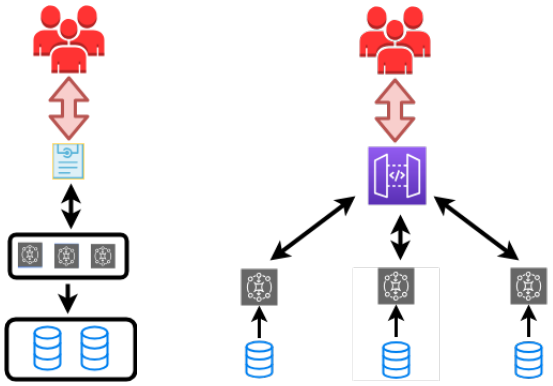


Figure 3.6. Princípios arquiteturais de um sistema reativo ([Bonér et al. 2014]).



(a) Monolito (b) Microserviços

Figure 3.7. Arquitetura Monolítica versus Arquitetura de Microserviços [Fowler and Lewis 2014].

Newman [Newman 2015] lista alguns benefícios, discutidos abaixo, ao adotar os Microserviços como uma solução no desenvolvimento de aplicações.

- **Technologia Heterogênea:** cada parte da aplicação pode ser implementada com diferentes tecnologias. Portanto, se uma parte da aplicação precisar melhorar sua qualidade de serviço, é possível decidir usar uma pilha de tecnologia diferente que seja mais adequada para atingir os níveis de QoS necessários. Por exemplo, na Figura 3.7 (lado b), cada aplicativo pode ser construído com diferentes tecnologias para atender a cada objetivo.

- **Dimensionamento:** com os Microserviços, é possível dimensionar partes da aplicação de acordo com a necessidade. Assim, é possível executar outras partes da aplicação em um dispositivo com menos poder computacional.

- **Facilidade de Implementação:** com os Microserviços, é possível alterar um único serviço e implantá-lo independentemente do restante do sistema. Se ocorrer um

problema, ele pode ser isolado rapidamente para um serviço individual, facilitando a recuperação rápida.

- **Alinhamento Organizacional :** com os Microserviços, é possível alinhar melhor a arquitetura da aplicação com a estrutura organizacional.

- **Composabilidade:** Um dos principais problemas nas arquiteturas orientadas a serviços e nos sistemas distribuídos é a possibilidade de melhorar a reutilização das aplicações. Com os Microserviços, é possível uma funcionalidade a ser consumida de diferentes maneiras para diferentes fins.

3.4. Arquitetura de Microserviços reativos para aplicações IoT

De acordo com o que foi discutido na Introdução, o crescente número de dispositivos na IoT permitirá o desenvolvimento de aplicações em vários domínios, como saúde, automação industrial e transporte. Essas aplicações de IoT possuirão os requisitos de QoS que devem ser concedidos em diferentes camadas da arquitetura IoT, nas quais a confiabilidade tem sido uma preocupação prioritária de sistemas IoT [White et al. 2017].

Neste contexto, a proposta de uma arquitetura de Microserviços confiáveis para o desenvolvimento de aplicações de IoT tem o objetivo de contribuir para melhorar a disponibilidade de aplicações quando comparado com soluções tradicionais em um ecossistema de IoT. A disponibilidade também pode ser definida como a capacidade do sistema de executar suas funcionalidades sob determinadas condições em um instante de tempo [Birolini 2013].

A Arquitetura de Microserviços discutida neste Capítulo foi projetada para orientar e apoiar o desenvolvimento de aplicações IoT resilientes e elásticas (ou seja, Responsivas). Esses recursos exigem um projeto de arquitetura que priorize o uso de mensagens assíncronas para garantir baixo acoplamento, transparência da localização e isolamento de comunicação entre diferentes partes da aplicação.

3.4.1. Arquitetura

Em primeiro lugar, a arquitetura define que uma aplicação IoT consiste de um conjunto de Microserviços reativos que podem ser distribuídos em dispositivos e servidores localizados na borda da rede, na névoa ou na nuvem. Os Microservices Reativos melhoram a autonomia, elasticidade e resiliência das aplicações. Em segundo lugar, nós fornecemos um broker para comunicação Máquina a Máquina. Em nossa proposta, esse broker foi projetado como um aplicação OSGI. Em terceiro lugar, adotamos containeres para aumentar o grau de elasticidade e reconfiguração das aplicações IoT. A Figura 3.8 ilustra a visão geral da proposta. Nessa arquitetura, a camada de aplicação possui duas partes principais: Microservices reativos, que guia o desenvolvimento das aplicações reativas e containers que está relacionada a infraestrutura necessária para a implementação e implantação das aplicações IoT. As características de cada componentes são discutidas a seguir.

Microserviços Reativos fornece uma arquitetura para o desenvolvimento de aplicações capazes de lidar com problemas de falha e perda de desempenho sem afetar o

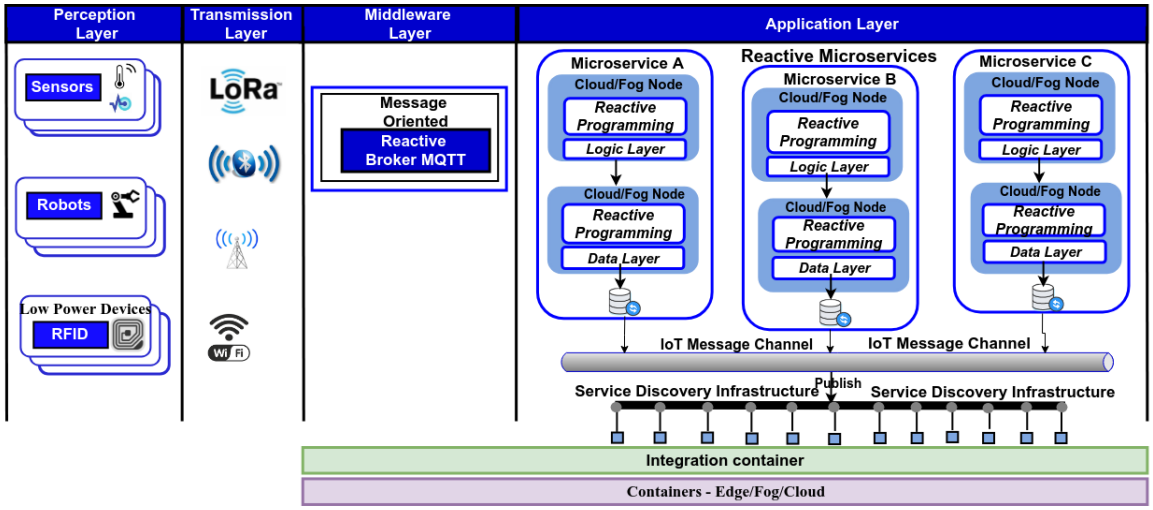


Figure 3.8. Visão Geral da Arquitetura.

comportamento de um aplicação inteira. Dividimos a camada lógica da arquitetura em duas partes: *Core* e *Utilities*. No *Core*, temos os seguintes componentes:

- **IoT Message Channel** fornece um meio de comunicação para que diferentes Microserviços possam se comunicar de maneira fracamente acoplada.
- **Service Discovery Infrastructure** fornece uma infraestrutura para a publicação e descoberta de serviços.
- **Reactive broker** fornece o ponto de comunicação com sensores e atuadores por meio de mensagens de publish/subscribe.
- **Circuit breaker** fornece um meio de evitar falhas em cascata.
- **Timeout and Bulkhead** permitir baixo acoplamento entre serviços.
- **Health check** fornece acesso ao status do Microsserviço: UP or Down.

A parte *Utilities* da arquitetura é usada em nossa proposta para fornecer suporte aos estudos de caso a serem implementados. Em Utilitários, temos os seguintes componentes:

- **Streamming** Fornece análise de dados em tempo real.
- **RESTful** Fornece acesso a dispositivos IoT.
- **Security** Fornece uma camada para autenticação e autorização para serviços.

3.4.2. Tecnologia

Para implementar cada componente dessa arquitetura (i.e. IoT Message Channel, Reactive broker, Circuit breaker,Timeout and Bulkhead, RESTful, Security) utilizamos a plataforma ilustrada na Figura 3.10. Esta plataforma é baseada no Vert.x e no ServiceMix.

3.4.2.1. Vert.x

O Vert.x[4] é um toolkit para criar sistemas reativos distribuídos no topo da Java Virtual Machine e adota um modelo de desenvolvimento assíncrono não bloqueiante. Como um toolkit, o Vert.x pode ser usado em muitos contextos: em uma aplicação standalone ou incorporado em um aplicação Spring[5]. O Vert.x e seu ecossistema são apenas arquivos jar usados como qualquer outra biblioteca: basta colocá-los em seu classpath e estará pronto para utilização. O Vert.x não fornece uma solução "all-in-one", mas fornece os blocos de construção para que desenvolvedores possa criar sua própria solução.

Uma parte do ecossistema Vert.x é mostrado na Figura 3.9. É possível selecionar qualquer um desses componentes (Figura 3.9) além do núcleo Vert.x para criar seus sistemas distribuídos. Por exemplo, para manipular conexões com clientes MQTT ou até mesmo criar um broker MQTT o componente responsável é o Vert.x MQTTT.

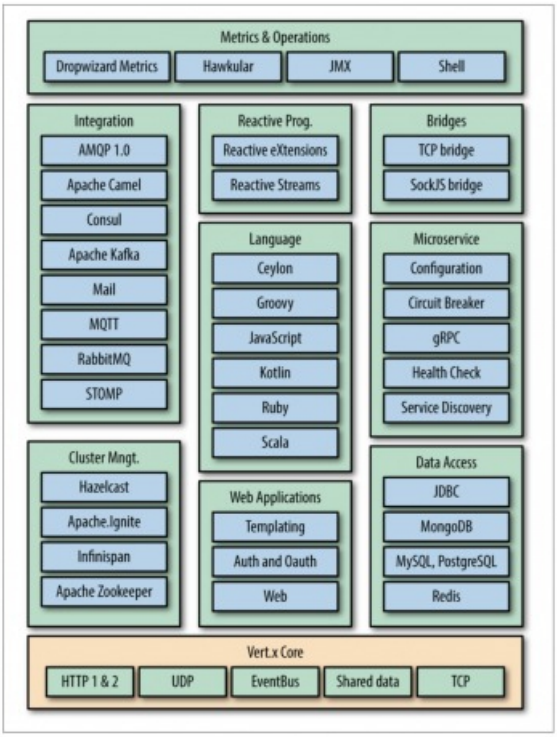


Figure 3.9. Uma parte do ecossistema Vertx. Adaptado [Bonér et al. 2014]

Uma das vantagens do Vert.x em comparação com outras tecnologias (por exemplo, Akka[6], Ratpack[7]) é o fornecimento de um conjunto de componentes capazes de projetar aplicações reativas em diferentes linguagens. Além disso, o Vert.x tem melhor desempenho do que o Akka e outros frameworks em alguns benchmarks, como em TechEmpower[8].

[4]https://vertx.io/
[5]https://spring.io/
[6]https://akka.io/
[7]https://ratpack.io/
[8]https://www.techempower.com/benchmarks/

```
1
2  //Exemplo de inclusao de dependencia no Maven
3
4  <dependency>
5     <groupId>io.vertx</groupId>
6     <artifactId>vertx-mqtt</artifactId>
7     <version>3.8.1</version>
8  </dependency>
9
10 //Exemplo de inclusao de dependencia no Gradle
11
12 compile io.vertx:vertx-mqtt:3.8.1
```

Código 3.1. Exemplos de inclusao de uma dependencia Vertx no Maven e Gradle

É possível utilizar o Vert.x de diferentes formas. Com o Maven ou Gradle é necessário apenas adicionar ao gerenciador de dependências o artefato a ser incluído no projeto. Por exemplo, na Listagem 3.1 é incluído a dependência para utilização do Vert.x MQTT Server e Vert.x MQTT Client.

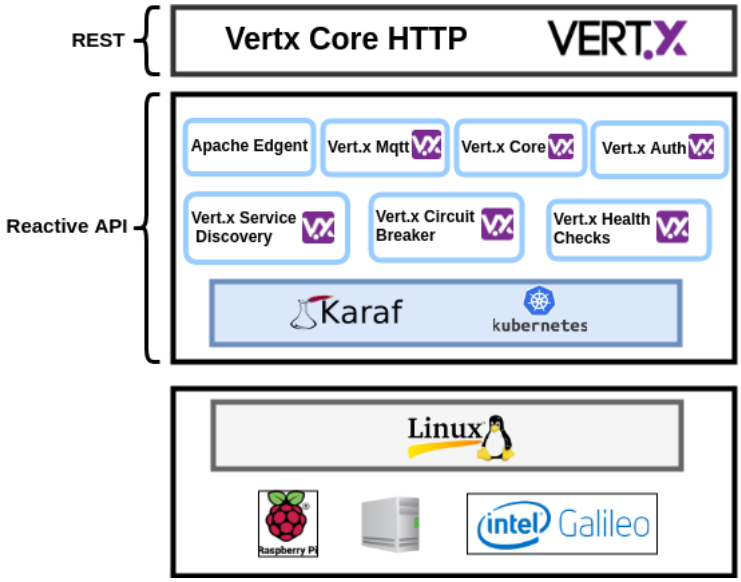


Figure 3.10. Reactive Microservices IoT Platform.

3.4.2.2. ServiceMix

O Apache ServiceMix é um aplicação de código aberto, implementada em Java, no qual serviços nativos da arquitetura ESB/OSGi oferecem toda a infraestrutura necessária para o suporte dos outros serviços da plataforma SOFT-IoT. Com o ServiceMix é possível implantar serviços (também chamados de bundles) em tempo de execução e permite que estes compartilhem dados e objetos através de relacionamentos e dependências.Figure 6.20. Componentes do Apache ServiceMixA Figura 6.20 representa o ServiceMix e os

seus principais módulos. O ServiceMix é leve e portável, tendo como requisito básico o suporte ao Java 1.7. Além disso,possui suporte ao Spring Framework e Blueprint e é compatível com o Java SE ou comum servidor de aplicativos Java EE. O Karaf é o núcleo principal do ServiceMix e ofereceo conceito de recursos, onde coleções de pacotes podem ser instalados como um grupoem um ambiente OSGi em execução. Sobe o Karaf, o ServiceMix usa o ActiveMQ parafornecer serviço de mensagens, CXF para suporte serviços RESTful, Cellar para comuni-cação e interações entre diferentes instalações do ServiceMix e Camel para integração etroca de dados através de rotas. Por fim, o ServiceMix contém módulos adicionais como H2 Database, que gerencia o sistema de banco de dados relacional baseado em arquivos.A plataforma SOFT-IoT foi baseada na versão 6.10 do ServiceMix. Para seu usoe instalação em um ambiente Linux é necessário somente baixar o pacote do programa, o qual pode ser encontrado em: `https://servicemix.apache.org/downloads/servicemix-6.1.0.html11http://servicemix.apache.org/`

Para inicializar o ServiceMix em sistemas Linux é preciso executar dentro do diretório do base o seguinte arquivo:

```
1  $ ./bin/servicemix
```

O ServiceMix pode ser gerenciado através de uma aplicação Web chamado webconsole. Nessa aplicação é possível gerenciar por meio de uma interface gráfica, funções como instalação, atualização, remoção e informações de bundles e também funções de debug. Para instalar o webconsole no ServiceMix basta executar o seguinte comando no terminal:

```
1  karaf@root()> feature:install webconsole
```

Por padrão aplicação Web inicializada pelo webconsole é configurada na porta 8181 com login e senha `karaf` e pode ser acessada pelo endereço:

`http://localhost:8181/system/console`

A Figura 3.11 mostra um recorte do webconsole. Observe que é possível através dele ter um panorama geral dos bundles em execução e também alterar seus status. O webconsole, além disso, permite instalar novos bundles e depurar o estado e eventuais erros das aplicações em execução.

3.4.3. Simulação para Internet das Coisas

Nas seções anteriores, entre outras coisas, foram abordadas as principais características sobre Internet das Coisas, Computação em Névoa e Microserviços. Com base nessas informações, apresenta-se nesta seção o processo de modelagem de ambientes que reproduzam cenários de Internet das Coisas em um ambiente de emulação/simulação. Para tal, utilizam-se ambiente para emulação de redes, tecnologias, protocolos e formatos de dados amplamente utilizados no contexto de Internet das Coisas. O objetivo é modelar cenários, realistas o suficiente, para que possibilitem a sua utilização como forma de testar e/ou validar novas soluções no âmbito de Internet das Coisas e/ou Névoa das Coisas. Na modelagem são implementados os componentes sensores, Gateway IoT e Gateway Virtualizado, todos, em conjunto com tecnologias e padrões (MQTT, ServiceMix, JSON, Plataforma Amazon EC2) já utilizados em equipamentos reais.

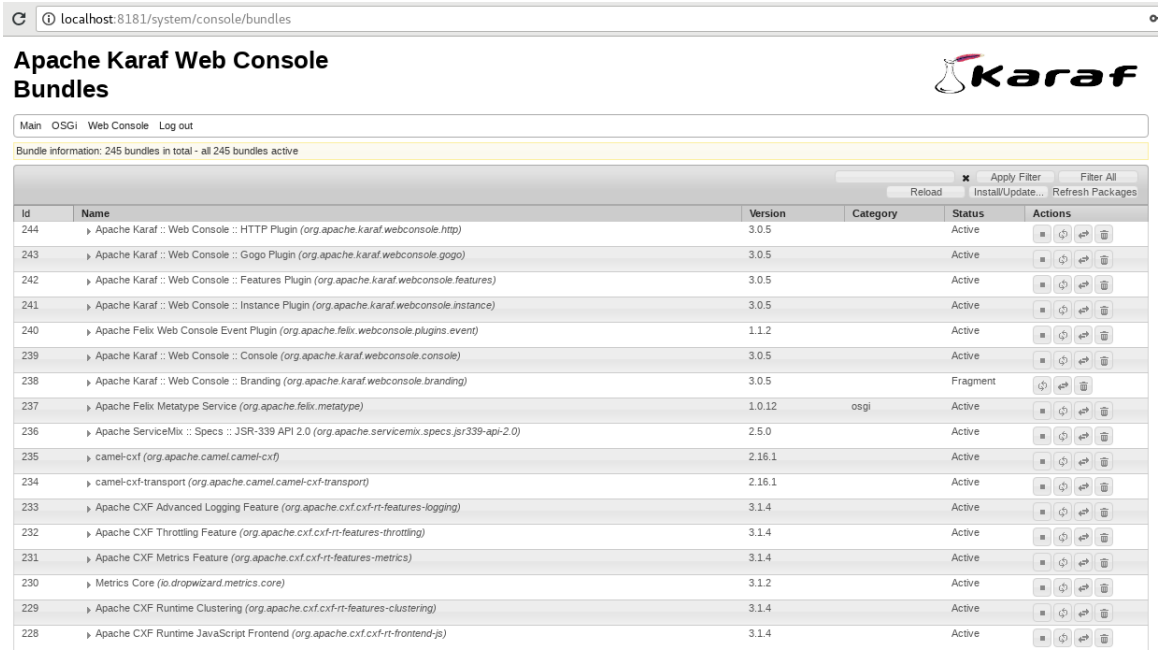


Figure 3.11. Webconsole do Apache ServiceMix (Karaf)

3.4.3.1. Emulador de Redes Mininet

O Mininet [9] é um emulador de de redes que possibilita a criação de *hosts* virtuais, *switches* e links. O emulador Mininet é um projeto [10] de código aberto e disponibilizado de forma gratuita. Essas características, associadas à flexibilidade na criação de redes, posicionam o Mininet como facilitador no desenvolvimento, prototipagem, aprendizagem, teste e depuração de novas soluções em IoT. Algumas vantagens de utilizar o Mininet: (i) possibilita o teste de infraestruturas de redes de forma simples e de baixo custo; (ii) permite testes complexos de topologia sem a necessidade de equipamentos de redes físicos; (iii) fornece API na linguagem de programação Python para criação e experimentação de infraestruturas de redes.

O Mininet utiliza a virtualização baseada em processos para a criação e execução dos elementos que constituem a rede. Com isso, todos os dispositivos de rede são dotados de recursos individuais mesmo compartilhando o mesmo *kernel* linux. O uso do mecanismo de virtualização "linux *network namespaces*" possibilita ao Mininet prover interfaces de redes e tabelas de roteamento/encaminhamento individuais, características fundamentais para a modelagem e implementação dos dispositivos de rede. As interfaces de rede no Mininet são conectadas através de links ethernets virtuais para as diversas instâncias possíveis de *switches* (ver Figura 3.12), dentre elas, o *switch* padrão do Mininet e o *Open vSwitch* [11] (*switch* de software multicamada adequado para para funcionar como virtual *switch* em ambientes de máquinas virtuais).

[9] Mininet: *An Instant Virtual Network* , disponível em: http://mininet.org/

[10] *Emulator for rapid prototyping of Software Defined Networks*, disponível em: https://github.com/mininet/mininet

[11] *What Is Open vSwitch?*, disponível em: http://docs.openvswitch.org/en/latest/intro/what-is-ovs/

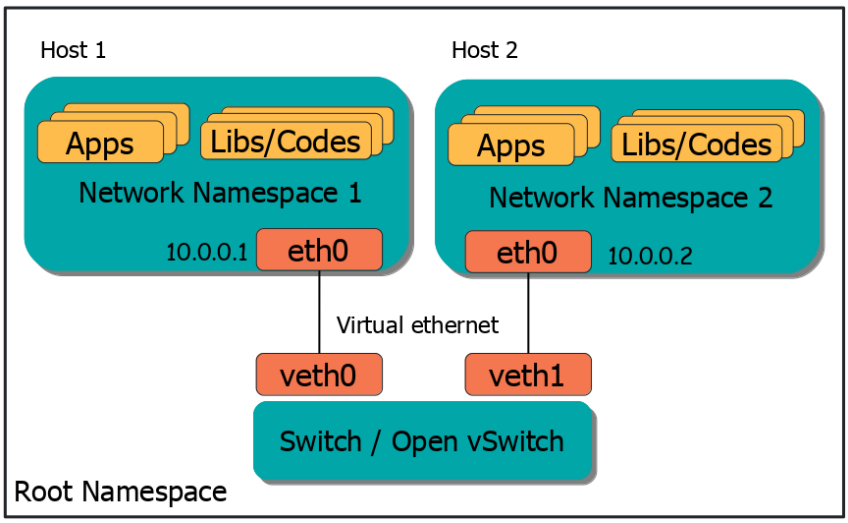


Figure 3.12. Mininet (*network namespaces*)

Com essas características, o Mininet fornece uma maneira simples para modelagem realista de ambientes que envolvem infraestrutura de redes. Os dispositivos que compõem a rede no Mininet executam código real, incluindo aplicativos e bibliotecas feitas para Linux, bem como a pilha do kernel e rede. Sendo assim, o código desenvolvido para ser executado em instâncias no Mininet (Controlador ou *hosts*) pode ser transferido para um dispositivo real com mínimas possibilidades de mudanças. Tal comportamento denota o Mininet como um emulador apropriado para implementação de um ambiente baseado no paradigma de Névoa das Coisas.

3.4.3.2. Modelagem da Internet das Coisas

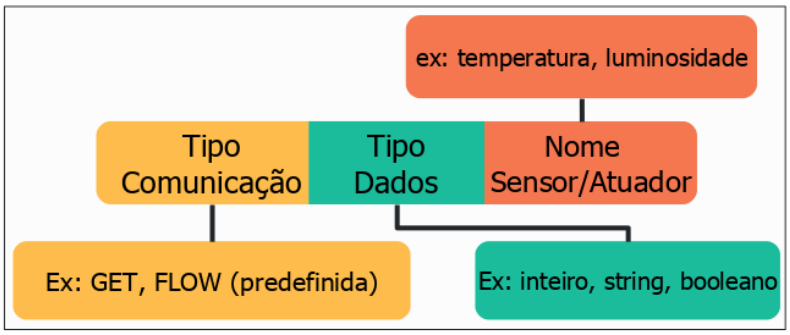
Baseado nos conceitos de Internet da Coisas, Computação em Nuvem e Névoa, descritos anteriormente, considera-se como componentes na modelagem: (i) Sensores; (ii) Gateways IoT; (iii) Nuvem / Gateways IoT virtualizados. Ressalta-se que a modelagem apresentada nesta Seção engloba os elementos básicos da Névoa das Coisas sob os aspectos de comportamento, comunicação e tecnologias utilizadas, de maneira que em um ambiente real de Névoa das Coisas (dispositivos físicos/reais) essas mesmas características tenham alto grau de similaridade com a modelagem proposta nesta Seção.

3.4.3.3. Protocolo de Comunicação TATU (*The Acessible Thing Universe Protocol*)

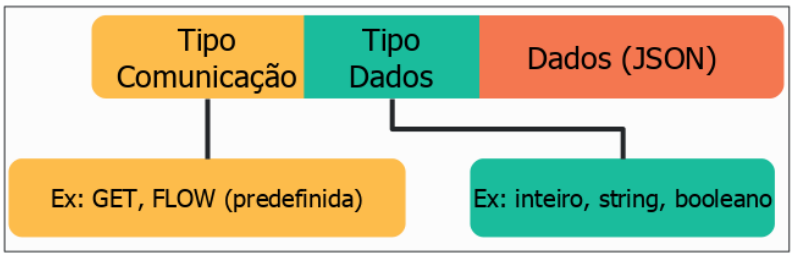
Embora o protocolo de comunicação MQTT seja leve e simples para ser implementado/executado em dispositivos IoT de capacidade limitada, existe a necessidade, dado a variedade das tarefas IoT e quantidade de informações geradas, de estabelecer padrões para a troca de mensagens em ambientes IoT. Nesse contexto, o protocolo de comunicação TATU [12], desenvolvido no contexto de IoT, Computação em Nuvem e Névoa,

[12]TATU: *The Acessible Thing Universe*, disponível em: https://github.com/WiserUFBA/TATUDevice

padroniza a troca de informações entre dispositivos IoT através de notações seguindo o formato JSON. O protocolo TATU pode ser utilizado de acordo com o tipo de comunicação/fluxo adotado nos dispositivos IoT, e atualmente estão implementados os modelos: (i) baseados em requisições GET/SET; (ii) baseados em requisições predefinidas (*flow* no TATU) [Prazeres and Serrano 2016].



(a) Requisição do tipo "pedido" no protocolo TATU



(b) Requisição do tipo "resposta" no protocolo TATU

Figure 3.13. Requisições no protocolo TATU.

A Figura 3.13 mostra os componentes de uma requisição no protocolo TATU, separadas em "pedido" e "resposta". Para realizar um "pedido", com o propósito de receber amostras com informações sobre uma grandeza (sensores) ou interagir com o ambiente (atuadores), deve-se especificar três campos (ver Figura 3.13(a)): (i) tipo de comunicação IoT; (ii) tipo de dados esperado na resposta; (iii) nome do sensor/atuador que deve atender/responder o pedido. Na requisição TATU do tipo "resposta", mostrada na Figura 3.13(b), adiciona-se um novo campo, chamado de dados ou carga útil. Esse campo contém informações no formato JSON sobre o sensor/atuador e a informação/resposta requerida pelo *FoT-Gateway* ou aplicação. O Código 3.2 mostra exemplos de mensagens construídas de acordo com o protocolo TATU. São mostradas mensagens da comunicação GET e FLOW (predifinidas) para as opções de pedido (requisição) e resposta.

Uma vez que o MQTT funciona fundamentalmente sob a abordagem de publicação/inscrição, as requisições TATU não são enviadas diretamente ao destino. Dessa forma, é necessário a organização do ambiente IoT para que os dispositivos realizem a inscrição nos endereços e tópicos corretos. A Figura 3.14 exibe o esquema necessário para realizar uma comunicação entre sensor e Gateway IoT, processo que pode ser resumido em quatro etapas (ver Figura 3.14): (1) sensor (MQTT *Client*) se inscreve no seu tópico no Gateway (MQTT *Broker*) que o gerencia; (2) Gateway IoT publica localmente no tópico do sensor desejado uma requisição TATU do tipo pedido; (3) o sensor recebe a

```
1
2  //Requisicoes baseadas em GET
3  GET INFO sensorType
4  //Resposta para Requisicoes GET
5  GET INFO RESPONSE {"code":"post","HEADER":{"requisition":"GET",
6  "name":"name_device"},"BODY":{"sensorType":"off"}
7
8  //Requisicoes predefinidas
9  FLOW INT sensorType {"collect":500, "publish":1000}
10 //Resposta p/ requisicoes predefinidas (cada 1000 milissegundos)
11 FLOW INT RESPONSE {"code":"post","HEADER":{"requisition":"FLOW",
12 "name":"device_name"},"BODY":{"sensorType":[30,31]}
```

Código 3.2. Exemplos do protocolo TATU

requisição TATU publicada pelo Gateway IoT na etapa anterior; (4) o sensor processa a informação recebida e publica a resposta no Gateway IoT.

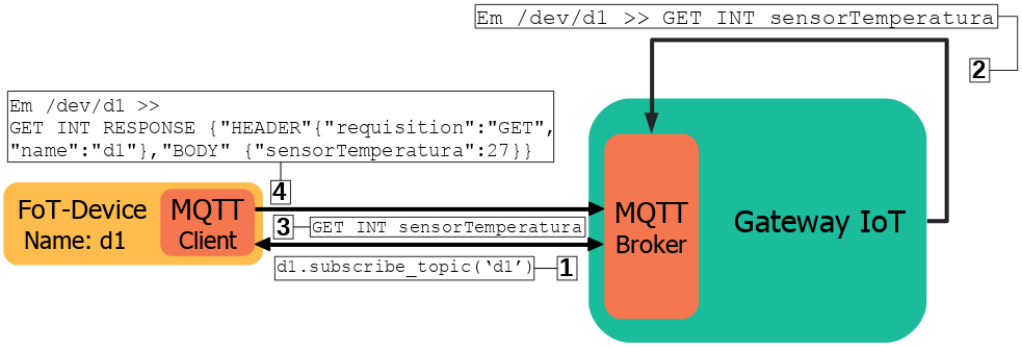


Figure 3.14. Exemplo de comunicação entre *FoT-Gateway* e *FoT-Device*

3.4.3.4. Modelagem do Sensor (virtual)

O sensor é o componente básico de um ambiente baseado no paradigma FoT. Para modelagem e, por conseguinte, simulação de um sensor virtual, alguns aspectos básicos precisam ser atendidos: (i) simulação de coleta de amostras (pseudoamostras) ou utilização de *dataset* de dados obtidos a partir de dispositivos reais e armazenamento de dados de curto período; (ii) implantação do protocolo de comunicação responsável por controlar o fluxo de informações entre sensores / Gateways / Nuvem.

O comportamento de um sensor é realizado por um serviço criado na linguagem de programação Python (ver Figura 3.15). É de sua responsabilidade a criação e armazenamento temporário de pseudoamostras. Para a identificação e construção de mensagens padronizadas de acordo com o protocolo TATU, realiza-se a importação e utilização da biblioteca que implementa as funções do protocolo TATU. A fim de realizar a comunicação, i.e. transporte de mensagens entre sensor e Gateway / Nuvem, utiliza-se o protocolo MQTT. Portanto, esse serviço executado em hosts do Mininet associado as bibliote-

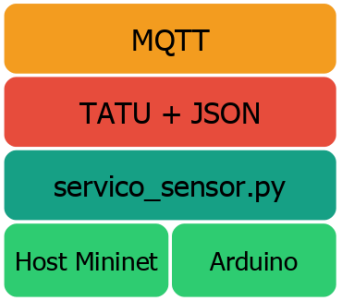


Figure 3.15. Modelo *FoT-Device*

cas e tecnologias utilizadas no processo de execução, incorporam ao host comportamentos de um sensor.

3.4.3.5. Modelagem do Gateway IoT

O Gateway IoT é o nó básico de gerência na Internet das Coisas. Gateways implementam suas funcionalidades através de serviços IoT (armazenamento, segurança, enriquecimento semântico, dentre outros) implementados seguindo especificações OSGi. Para a modelagem de um Gateway no Mininet alguns requisitos básicos devem ser atendidos: (i) implantação de framework que implementa as especificações OSGi (Apache ServiceMix); (ii) implantação dos serviços IoT no framework OSGi; (iii) implantação do protocolo de comunicação responsável por controlar o fluxo de informações entre sensores / Gateways / Nuvem.

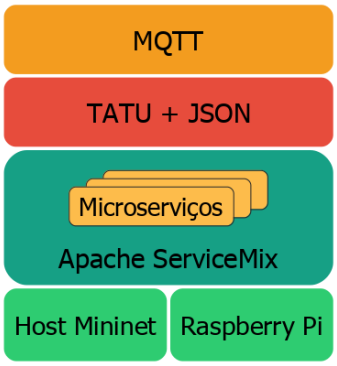


Figure 3.16. Modelo *FoT-Gateway*

A união dos componentes mostrados na Figura 3.16 caracterizam um host Mininet com o comportamento básico de um Gateway no cenário de Internet das Coisas. A gerência dos dispositivos é realizada pelos serviços IoT inclusos no Apache ServiceMix. As informações geradas pelos sensores são repassadas para os respectivos Gateways através do protocolo MQTT, já com o padrão definido pelo TATU. A partir disso, os Gateways poderão armazenar/tratar as informações recebidas.

3.4.3.6. Modelagem da Nuvem (Gateways IoT Virtualizados)

Os *FoT-Gateways* virtualizados servem como nó de gerência e realização de tarefas que exigem maior poder computacional. Podem ser disponibilizados em nível local (Névoa), através de servidores, ou em instâncias na Nuvem, conforme o proposto nesta Seção. Ao utilizar Gateways IoT disponibilizados na Nuvem, abre-se a possibilidade para a criação e desenvolvimento de estratégias que utilizem cenários híbridos (Nuvem e Névoa).

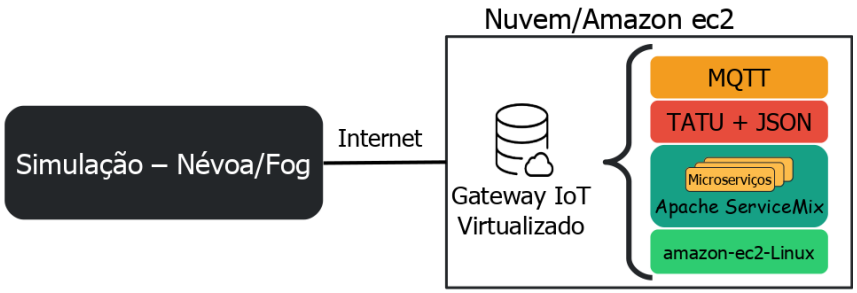


Figure 3.17. Modelo *FoT-Gateway* na Nuvem

A modelagem e implantação do Gateway IoT em um ambiente de Nuvem (ver Figura 3.17) é similar ao próprio Gateway implantado na Névoa (Mininet). No entanto, modifica-se os *hosts* do Mininet por instâncias de máquinas virtuais / Sistemas Operacionais disponibilizados na Nuvem. Para disponibilizar os Gateways em ambiente de Computação em Nuvem neste trabalho, utiliza-se a plataforma *Amazon Elastic Compute Cloud* [13] (*Amazon* EC2), que, entre outras coisas, possibilita a criação de instâncias virtuais de Sistemas Operacionais com total controle sobre instalação e gerenciamento dos seus programas.

3.5. Estudo de Caso

Para demonstrar de forma prática o desenvolvimento de aplicações IoT com Microserviços reativos um estudo de caso foi desenvolvido. Nesse estudo uma aplicação IoT foi desenvolvida para monitoramento das variáveis temperatura, umidade, som e luminosidade de um laboratório. Esse tipo de aplicação é definida na literatura como ambiente inteligente (do inglês *Smart Environment*). Um *Smart Environment* é uma sala ou espaço com sensores e/ou atuadores. Os sensores/atuadores e aplicações conectados em rede fazem a sala perceber o estado físico e as atividades internas do ambiente. Em um ambiente inteligente, as rotinas do usuário e os processos de informação podem interagir perfeitamente entre si.

Este estudo de caso foi desenvolvido tendo como base a arquitetura apresentada na Figura 3.18. Como sensores do ambiente foram utilizados o Sonoff SC apresentado na Seção 3.5.0.1. O equipamento do gateway IoT é o Raspberry Pi descrito na Seção 3.5.0.2. O estudo de caso utiliza uma arquitetura de Computação em Névoa combinada com a Computação em Nuvem. Para o ambiente de Névoa e Nuvem foram desenvolvidos Microserviços reativos e utilizadas tecnologias reativas com os seguinte objetivos:

[13] *Amazon Elastic Compute Cloud* , disponível em: https://aws.amazon.com/pt/ec2/

- **Microserviço Reativo de Análise de Dados**: Microserviço que realiza as análises de dados para tomadas de decisões no ambiente.
- **Microserviço Reativo de Coleta de Dados**: Microserviço responsável por receber e armazenar os dados produzidos pelos sensores.
- **Microserviço Reativo de Acesso a dados**: Microserviço que disponibiliza os dados coletados pelos sensores para acesso de aplicações por meio de serviços RESTFul.
- **Canal de Mensagens IoT**: Componente da arquitetura que permite a comunicação entre os Microserviços de Análise, Coleta e Acesso aos dados.
- **Broker Reativo**: Componente da arquitetura que utiliza a comunicação assíncrona pub/sub para troca de mensagens entre os Microserviços e os sensores do ambiente.

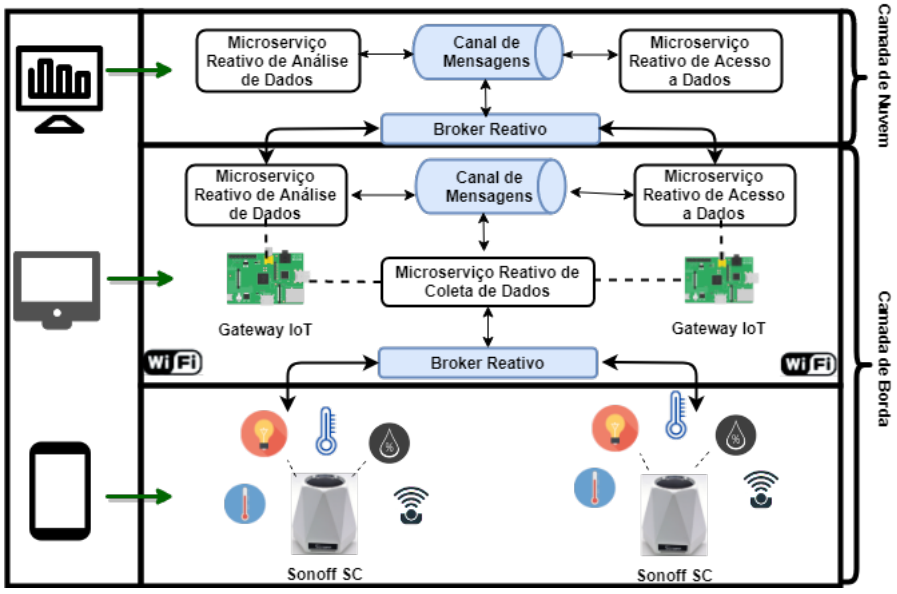


Figure 3.18. Arquitetura do Estudo de Caso.

3.5.0.1. Sonoff SC

O Sonoff SC é um nó de sensores desenvolvido pela ITEAD[14] que possui cinco sensores, são eles: temperatura, umidade, som, poeira e luminosidade. Os sensores que compõem o Sonoff SC são exibidos na Figura 3.19. A especificação dos sensores do Sonoff é a seguinte: o sensor de temperatura e umidade é o DHT11; o sensor de luminosidade é o LDR; o sensor de poeira é o Sharp GP2Y1010AU0F; para captação de som é utilizado um microfone.

O nó de sensores Sonoff SC utiliza o microcontrolador ATmega328. Esse microcontrolador do Sonoff foi desenvolvido pela Atmel Corporation e possui baixo custo.

[14]https://www.itead.cc/

Contudo, apresenta um baixo poder de processamento por causa do seu microprocessador de 8 bits. No Sonoff SC ilustrado na Figura 3.19, o ATmega328 é responsável pela conexão entre o ESP8266 e os sensores.

O ESP8266 é um módulo WiFi denominado como um SoC (System on a Chip). Esse módulo possui o protocolo TCP/IP integrado que pode dar acesso à rede WiFi para qualquer microcontrolador. O ESP8266 também possui um microprocessador de 32 bits com memória disponível. Com esse módulo WiFi é possível tanto hospedar uma aplicação quanto dar suporte às funções de rede WiFi de outro processador. Cada unidade possui uma programação com um firmware de conjunto de comandos AT. O módulo ESP8266 é uma placa com custo baixo e contínua expansão da comunidade.

Figure 3.19. Componentes Sonoff-SC.

3.5.0.2. Raspberry Pi

O Raspberry Pi, Figura 3.20, é um computador do tamanho de um cartão de crédito, desenvolvido no Reino Unido pela Fundação Raspberry Pi[15], foi criado para promover o ensino de ciência da computação nas escolas. O primeiro modelo tornou-se mais popular do que o esperado, visto que as vendas foram além do seu mercado-alvo, sendo adquiridos para usos distintos como na robótica.

Figure 3.20. Raspberry Pi Modelo A (Esquerda) e Modelo B (Direita).

[15]https://www.raspberrypi.org/

Os números mostram a aceitação do público ao dispositivo. De acordo com a Fundação Raspberry Pi, mais de 5 milhões de dispositivos foram vendidos antes de fevereiro de 2015, tornando-se o computador britânico mais comercializado. Em novembro de 2016, foram vendidos 11 milhões de unidades, atingindo 12,5 milhões em março de 2017, tornando-se o terceiro computador de propósito geral mais vendido.

O Raspberry Pi, como qualquer outro computador, usa um sistema operacional (SO). A opção Linux, chamada Raspbian, é grátis e de código aberto, reduzindo o preço da plataforma e tornando-o mais fácil de usar [Maksimović et al. 2014]. Existem outras opções de SO disponíveis [Richardson and Wallace 2012]. O fato do Raspberry Pi ser atrativo consiste em possuir uma ampla gama de aplicação.

O Raspberry Pi além de ser pequeno, possui uma série de vantagens como o baixo custo quando comparado aos demais dispositivos no mercado [Maksimović et al. 2014]. Uma redução no custo, em alguns casos, poderá resultar na capacidade de comprar mais dispositivos, tornando possível, por exemplo, implantar uma rede de maior densidade para coletar mais dados. Outro aspecto relevante está relacionado a energia. O componente principal do Raspberry Pi é a CPU que é responsável por realizar as instruções de um programa de computador através de operações matemáticas e lógicas. O processador ARMbased BCM2835, que é barato, poderoso, e não consome muita energia, é o motivo pelo qual o Raspberry Pi é capaz de operar apenas na fonte de alimentação 5V/1A fornecida pela porta micro-USB [Maksimović et al. 2014].

3.5.1. Implementação

- **Microserviço Reativo de Análise de Dados**:

  Este Microserviço habilita a análise de dados na borda da rede. O Microserviço de análise de dados pode ser verificada em:

```
https://github.com/WiserUFBA/reactsmartlab/blob/master/
   NetBeansProjects/
iot-reactive-application/src/main/java/
controller/AnaliseDadosController.java
```

- **Microserviço Reativo de Coleta de Dados**:

  Estes Microserviços capturam dados dos dispositivos conectados. Os Microserviços responsáveis pela coleta de dados podem ser verificados em:

```
https://github.com/WiserUFBA/reactsmartlab/blob/
master/NetBeansProjects/
iot-reactive-application/src/main/java/model/Sensor.java
```

```
https://github.com/WiserUFBA/reactsmartlab/
blob/master/NetBeansProjects/
iot-reactive-application/src/main/java/
controller/ReactiveController.java}
```

- **Microserviço Reativo de Acesso a dados**: Este Microserviço é responsável em expor os dados dos sensores. O Microserviço de exposição dos dados pode ser verificado em:

```
https://github.com/WiserUFBA/reactsmartlab/blob/master/
NetBeansProjects/
iot-reactive-application/src/main/java/
controller/AcessoDadosController.java
```

- **Canal de Mensagens IoT**:

  O componente Canal de Mensagens IoT permite que diferentes partes de um aplicação ou serviço se comuniquem de maneira pouco acoplada. As mensagens são enviadas aos endereços e os consumidores se registram nesses endereços para receber as mensagens. O canal de mensagens da IoT também é clusterizado, o que significa que ele pode enviar mensagens pela rede entre remetentes e consumidores distribuídos. Além disso, o Canal de Mensagens IoT envia mensagens para as instâncias disponíveis e, assim, equilibra a carga entre os diferentes nós registrados no mesmo endereço.

  A utilização do canal de mensagens IoT pode ser verificada em:

```
https://github.com/WiserUFBA/reactsmartlab/
blob/master/NetBeansProjects/
iot-reactive-application/src/main/java/
controller/ReactiveController.java}
```

  Um exemplo de utilização dessa canal de mensagens é ilustrado pelo código abaixo:

```
vertx.eventBus().send("webmedia",  hndlr.topicName() +
hndlr.payload().toString() );
```

- **Broker Reativo**:

  O Microserviço MQTT Broker fornece um servidor que é capaz de lidar com conexões, comunicação e troca de mensagens com clientes MQTT remotos. Na plataforma SOFT-IoT o broker MQTT é implemetado no módulo **soft-iot-vertx-mqtt-broker**[16] uma implementação baseada em OSGI como um *bundle* do ServiceMix.

  Este módulo é responsável por habilitar comunicações com os dispositivos conectados. Além das vantagens de usar uma arquitetura modular, o uso da API Vert.x MQTT Server torna possível escalar o agente MQTT reativo de acordo com, por exemplo, o número de núcleos do sistema e, assim, possibilitar a escalabilidade horizontal.

  Antes instalar o Microserviço soft-iot-vertx-mqtt-broker é necessário introduzir ao ServiceMix alguns módulos do Vert.x. As dependências estão localizadas no endereço:

```
https://github.com/WiserUFBA/soft-iot-vertx-mqtt-broker/
tree/master/src/main/resources/dependencies
```

  As dependências são bibliotecas .jar que devem ser incluídas no ServiceMix, podendo ser introduzidas através do webconsole, conforme pode ser visto na Figura 3.11 através do botão `Install/Update`. Além da instalação das dependências é

[16]https://github.com/WiserUFBA/soft-iot-vertx-mqtt-broker

necessário também incluir arquivos referente ao certificado de segurança do soft-iot-vertx-mqtt-broker. Tais arquivos estão disponíveis em:

```
https://github.com/WiserUFBA/soft-iot-vertx-mqtt-broker/tree/
    master/
src/main/resources/certificates
```

O certificado de segurança possui um arquivo de configuração disponível em:

```
https://github.com/WiserUFBA/soft-iot-vertx-mqtt-broker/
blob/master/src/main/resources/configuration/
br.ufba.dcc.wiser.soft_iot.gateway.brokers.cfg
```

Os arquivos do certificado de segurança, incluindo o arquivo de configuração devem ser armazenados em:

```
<diretorio_servicemix>/etc
```

Para instalação do soft-iot-vertx-mqtt-broker é necessário executar os seguintes comandos no terminal do ServiceMix:

```
karaf@root()> bundle:install mvn:br.ufba.dcc.wiser.soft_iot/
soft-iot-vertx-mqtt-broker/1.0.0
```

3.6. Considerações Finais

Este capítulo introduziu o conceito de Microserviços reativos, cujo objetivo é permitir a construção de aplicações na IoT que sejam performativos, resilientes e escalonáveis, o que é possível pela troca de mensagens assíncronas. Como prova de conceito, implementamos uma solução para análise de dados das temperaturas e humidade de uma laboratório de pós graduação na Universidade Federal da Bahia. Nesse experimento é possível mostrar as características de resiliência e elasticidade fornecidas nos Microserviços.

References

[Abdelshkour 2015] Abdelshkour, M. (2015). IoT, from cloud to fog computing. `http://blogs.cisco.com/perspectives/iot-from-cloud-to-fog-computing`. [Online; acessado 04 de Setembro de 2019].

[Al-Fuqaha et al. 2015] Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., and Ayyash, M. (2015). Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376.

[Andrade et al. 2018] Andrade, L., Lira, C., Mello, B., Andrade, A., Coutinho, A., Greve, F., and Prazeres, C. (2018). Soft-iot platform in fog of things. In *Proceedings of the 24th Brazilian Symposium on Multimedia and the Web*, WebMedia '18, pages 23–27, New York, NY, USA. ACM.

[Ashton 2009] Ashton, K. (2009). That 'internet of things' thing. *RFiD Journal*, 22:97–114.

[Bainomugisha et al. 2013] Bainomugisha, E., Carreton, A. L., Cutsem, T. v., Mostinckx, S., and Meuter, W. d. (2013). A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34.

[Bassi et al. 2013] Bassi, A., Bauer, M., Fiedler, M., Kramp, T., Kranenburg, R. v., Lange, S., and Meissner, S., editors (2013). *Enabling Things to Talk: Designing IoT Solutions with the IoT Architectural Reference Model*. Springer, Heidelberg.

[Bauer et al. 2013] Bauer, M., Boussard, M., Bui, N., Carrez, F., Jardak (SIEMENS, C., De Loof (ALUBE, J., Magerkurth (SAP, C., Meissner, S., Nettsträter (FhG IML, A., Olivereau, A., Thoma (SAP, M., Joachim, W., Stefa (CSD/SUni, J., and Salinas, A. (2013). Internet of things – architecture iot-a deliverable d1.5 – final architectural reference model for the iot v3.0.

[Bernstein 2014] Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84.

[Bilal et al. 2018] Bilal, K., Khalid, O., Erbad, A., and Khan, S. U. (2018). Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers. *Computer Networks*, 130:94 – 120.

[Birolini 2013] Birolini, A. (2013). *Reliability engineering: theory and practice*. Springer Science & Business Media.

[BLOOM 2013] BLOOM, Z. (2013). How we deploy 300 times a day.

[Bonér 2017] Bonér, J. (2017). *Reactive Microsystems The Evolution of Microservices at Scale*. O'Reilly Media, Gravenstein Highway North, Sebastopol.

[Bonér et al. 2014] Bonér, J., Farley, D., Kuhn, R., and Thompson, M. (2014). The reactive manifesto.

[Bonomi et al. 2012] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. (2012). Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA. ACM.

[Borgia 2014] Borgia, E. (2014). The internet of things vision: Key features, applications and open issues. *Computer Communications*, 54:1 – 31.

[Botta et al. 2016] Botta, A., de Donato, W., Persico, V., and Pescapé, A. (2016). Integration of cloud computing and internet of things: A survey. *Future Generation Computer Systems*, 56:684 – 700.

[Cavalcante et al. 2016] Cavalcante, E., Pereira, J., Alves, M. P., Maia, P., Moura, R., Batista, T., Delicato, F. C., and Pires, P. F. (2016). On the interplay of internet of things and cloud computing: A systematic mapping study. *Computer Communications*, 89-90:17 – 33. Internet of Things : Research challenges and Solutions.

[CI 2015] CI, C. (2015). Db performance issue incident report for circleci.

[da Rosa Righi et al. 2018] da Rosa Righi, R., Correa, E., Gomes, M. M., and da Costa, C. A. (2018). Enhancing performance of iot applications with load prediction and cloud elasticity. *Future Generation Computer Systems*, 1(1):1–13.

[Díaz et al. 2016] Díaz, M., Martín, C., and Rubio, B. (2016). State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing. *Journal of Network and Computer Applications*, 67:99 – 117.

[de Santana et al. 2018] de Santana, C. J. L., de Mello Alencar, B., and Prazeres, C. V. S. (2018). Microservices: A mapping study for internet of things solutions. In *2018 IEEE International Symposium on Network Computing and Applications (NCA)*, pages 1–4, Cambridge, MA, USA. IEEE.

[de Santana et al. 2019] de Santana, C. J. L., de Mello Alencar, B., and Prazeres, C. V. S. (2019). Reactive microservices for the internet of things: A case study in fog computing. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 1243–1251, New York, NY, USA. ACM.

[Distefano et al. 2012] Distefano, S., Merlino, G., and Puliafito, A. (2012). Enabling the cloud of things. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pages 858–863.

[Dragoni et al. 2017] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham.

[for Standardization/International Electrotechnical Commission et al. 2011] for Standardization/International Electrotechnical Commission, I. O. et al. (2011). Iso/iec 25010-systems and software engineering—systems and software quality requirements and evaluation (square)—system and software quality models. *Authors, Switzerland*, 1(15):1–15.

[Fowler and Lewis 2014] Fowler, M. and Lewis, J. (2014). Microservices, 2014. *URL: http://martinfowler. com/articles/microservices. html*, 1(1):1–1.

[Francesco et al. 2017] Francesco, P. D., Malavolta, I., and Lago, P. (2017). Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 21–30, Gothenburg, Sweden. IEEE.

[Gérald 2010] Gérald, S. (2010). *The Internet of Things: Between the Revolution of the Internet and the Metamorphosis of Objects*. Publications Office of the European Union.

[Humble and Farley 2011] Humble, J. and Farley, D. (2011). *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley Boston, EUA.

[Khan et al. 2012] Khan, R., Khan, S. U., Zaheer, R., and Khan, S. (2012). Future internet: The internet of things architecture, possible applications and key challenges. In *2012 10th International Conference on Frontiers of Information Technology*, pages 257–260.

[kubernetes 2019] kubernetes (2019). What is kubernetes. *https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/*.

[Lasse Lueth 2018] Lasse Lueth, K. (2018). State of the iot 2018: Number of iot devices now at 7b – market accelerating. *Press Release. Online unter: https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/*.

[Maksimović et al. 2014] Maksimović, M., Vujović, V., Davidović, N., Milošević, V., and Perišić, B. (2014). Raspberry pi as internet of things hardware: performances and constraints. *design issues*, 3:8.

[Namiot and Sneps-Sneppe 2014] Namiot, D. and Sneps-Sneppe, M. (2014). On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27.

[Newman 2015] Newman, S. (2015). *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.".

[Prazeres and Serrano 2016] Prazeres, C. and Serrano, M. (2016). Soft-iot: Self-organizing fog of things. In *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 803–808.

[Richardson and Wallace 2012] Richardson, M. and Wallace, S. (2012). *Getting started with raspberry PI*. " O'Reilly Media, Inc.".

[Serrano et al. 2015a] Serrano, M., Barnaghi, P., Carrez, F., Cousin, P., Vermesan, O., and Friess, P. (2015a). Iot semantic interoperability: Research challenges, best practices, recommendations and next steps. *IERC: European Research Cluster on the Internet of Things,*.

[Serrano et al. 2015b] Serrano, M., Quoc, H. N. M., Phuoc, D. L., Hauswirth, M., Soldatos, J., Kefalakis, N., Jayaraman, P. P., and Zaslavsky, A. (2015b). Defining the stack for service delivery models and interoperability in the internet of things: A practical case with openiot-vdk. *IEEE Journal on Selected Areas in Communications*, 33(4):676–689.

[Shahid and Aneja 2017] Shahid, N. and Aneja, S. (2017). Internet of things: Vision, application areas and research challenges. In *International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pages 583–587, Palladam, India. IEEE.

[Shi and Dustdar 2016] Shi, W. and Dustdar, S. (2016). The promise of edge computing. *Computer*, 49(5):78–81.

[Sundmaeker et al. 2010] Sundmaeker, H., Guillemin, P., Friess, P., and Woelfflé, S., editors (2010). *Vision and Challenges for Realising the Internet of Things*. Publications Office of the European Union, Luxembourg.

[Taveras Núñez 2017] Taveras Núñez, P. M. (2017). *A Reactive Microservice Architectural Model with Asynchronous Programming and Observable Streams as an Approach to Developing IoT Middleware*. PhD thesis, PUCMM. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Última atualização em - 2018-05-09.

[Trojak 2018] Trojak, M. (2018). Ci/cd at scale.

[Udoh and Kotonya 2018] Udoh, I. S. and Kotonya, G. (2018). Developing iot applications: challenges and frameworks. *IET Cyber-Physical Systems: Theory Applications*, 3(2):65–72.

[White et al. 2017] White, G., Nallur, V., and Clarke, S. (2017). Quality of service approaches in iot: A systematic mapping. *Journal of Systems and Software*, 132:186 – 203.

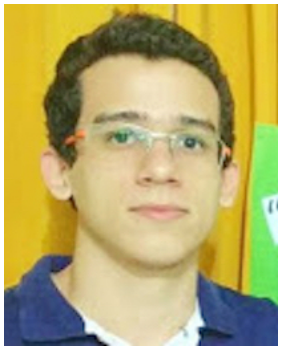
[Zhang et al. 2010] Zhang, S., Zhang, S., Chen, X., and Huo, X. (2010). Cloud computing research and development trend. In *2010 Second International Conference on Future Networks*, pages 93–97.

Biografia Resumida dos Autores

Cleber Santana, é doutorando em Ciência da Computação pela Universidade Federal da Bahia (UFBA). Obteve o título de Mestre em Sistemas e Computação (2014) e graduado em Processamento de Dados pela Faculdade Ruy Barbosa. É pesquisador do grupo WISER-UFBA/NUMAC-IFBA, atuando em projetos relacionados a Internet das Coisas, Microserviços, Web Semântica, Computação em Névoa e Educação. Cleber é membro da IEEE Communications Society, da Communications Society e possui publicações em conferências nacionais e internacionais. Desde 2013 é professor do Instituto Federal da Bahia e possui interesse em pesquisa nas áreas de Web Semântica, Serviços Web, Microserviços, Internet das Coisas, Computação em Névoa, Inteligência Artificial e Informática na Educação.

Leandro Andrade, é doutorando em Ciência da Computação pela Universidade Federal da Bahia (UFBA). Obteve o título de Mestre em Ciência da Computação (2014) e também o de Bacharel em Ciência da Computação (2012) pela UFBA. É pesquisador do grupo WISER-UFBA, atuando em projetos relacionados a Internet das Coisas, Computação em Névoa e Big Data. Realizou doutorado sanduíche no The Insight Centre for Data Analytics (NUI Galway - Irlanda). Leandro é membro da Sociedade Brasileira de Computação (SBC), da Communications Society e

possui publicações em conferências nacionais e internacionais. É professor substituto da UFBA, vinculado ao Departamento de Ciência da Computação. Possui interesse em pesquisa nas áreas de Internet das Coisas, Computação em Névoa, Serviços Web, Web Semântica, Big Data, Aprendizado de Máquina, Informática na Educação e Software Livre.



Brenno de Mello é mestrando em Ciência da Computação da Universidade Federal da Bahia (UFBA). Atualmente, é participante do grupo de pesquisa WISER, pesquisando principalmente os seguintes temas: Internet das Coisas, Mineração de Fluxo de Dados, Fog Computing, Smart Water. Mello possui graduação em Análise e Desenvolvimento de Sistemas pelo Instituto Federal da Bahia (2016). Tem experiência na área de Ciência da Computação, com ênfase em Sistemas de Informação, atuando como Analista de Sistemas.

José Sampaio, é graduando em Ciência da Computação pela Universidade Federal da Bahia (UFBA). Possui bolsa de iniciação científic apela FAPESB e faz parte do grupo WISER-UFBA, atuando em projetos relacionados a Internet das Coisas, Computação em Névoa e Microserviços Reativos. Possui interesse em pesquisa nas áreas de Internet das Coisas, Computação em Névoa, Serviços Web e Big Data.

Ernando Batista é Mestre em Ciência da Computação pela Universidade Federal da Bahia (UFBA), Engenheiro de Computação e Bacharel em Ciências Exatas pela Universidade Federal do Recôncavo da Bahia (UFRB). É pesquisador no Laboratório e Grupo de Pesquisa CNPq WISER (Web, Internet and Intelligent Systems Research Group) e Professor no Instituto Federal de Educação, Ciência e Tecnologia da Bahia (IFBA). Possui interesse em pesquisa nas áreas de Internet das Coisas, Computação em Névoa, Serviços Web e Redes Definidas por Software.

Cássio Prazeres é Doutor em Ciências - Área de Ciências de Computação e Matemática Computacional - pela Universidade de São Paulo (2009), é professor Associado I na Universidade Federal da Bahia (UFBA) nas áreas Internet/Web e é orientador permanente no Programa de Pós-graduação em Ciência da Computação (PGCOMP-UFBA). Prazeres é membro: da Sociedade Brasileira de Computação (SBC); do ACM SIGWEB (Special Interest Group on Hypertext the Web); do IEEE Computer Society Technical Committee on Services Computing; do IEEE Smart Cities Technical Community; do IEEE Internet of Things Technical Community; e do W3C Web of Things Community Group. É co-fundador e líder do Laboratório e Grupo de Pesquisa CNPq WISER (Web, Internet and Intelligent Systems Research Group). Tem interesse em pesquisas envolvendo tópicos de: Internet of Things, Web of Things, Web Services, Semantic Web, Microservices, Fog Computing, Fog of Things,

Web of Data. Em 2015, Prazeres realizou estágio pós-doutoral como professor visitante no DERI (Digital Enterprise Research Institute) na National University of Ireland (Galway) nas áreas de Internet das Coisas e Web Semântica.