IOT – INTERNET DAS COISAS

AULA 4

Prof. Gian Carlo Brustolin

CONVERSA INICIAL

Até aqui, conhecemos os conceitos básicos e duas importantes aplicações dessa tecnologia. Iniciamos agora uma abordagem um pouco mais profunda tecnicamente nos próximos momentos, o que incluirá o estudo de técnicas de programação voltadas a IoT, tecnologias de redes e conceitos de segurança para IoT. Apesar do foco mais técnico, não é objetivo aqui o mergulho em códigos computacionais ou em programação de dispositivos, mas sim orientar o estudante na aplicação das técnicas estudadas em outros contextos do IoT.

Dessa forma, neste capítulo, que abordará aspectos de programação, não devemos esperar códigos e APIs. Abordaremos, sim, aspectos de engenharia de *software*, que particularizam o desenvolvimento para estes dispositivos, a exemplo da computação de névoa e microsserviços. O objetivo geral será sempre conduzir o estudante para a compreensão das particularizações, ou seja, indicar como os conhecimentos desenvolvidos em seus estudos anteriores podem ser utilizados em projetos envolvendo IoT. Deixaremos, entretanto, indicadas as referências para revisão ou aprofundamento dos tópicos.

Ao final deste capítulo, você terá uma visão técnica geral de desenvolvimento para IoT e, como de hábito neste estudo, conhecendo as inevitáveis incertezas e desafios ligados ao assunto.

Vamos então dar início a este empolgante estudo.

TEMA 1 – INTRODUÇÃO AO DESENVOLVIMENTO PARA IoT

Quando imaginamos o desenvolvimento de *software* para IoT, diante do que já conhecemos sobre a tecnologia, construímos uma ideia de modularidade ou de segmentação. Existem, de fato, vários desenvolvimentos em IoT, cujo escopo e complexidade, variarão, conforme a IoT-A escolhida. Como discutimos, em outro momento, em uma arquitetura mais simples, composta de objetos inteligentes mais complexos, de três camadas, teremos códigos para a programação do objeto em si, *back end*, para o tratamento da interconexão com o servidor e o *front end* de controle. A complexidade cresce ainda um pouco mais, quando desejamos (e normalmente o fazemos) que o *front end* possa ser acessado ou residir remotamente.

IoT-As que permitem objetos mais simples, desprovidos de algumas dos blocos funcionais não essenciais, delegam a códigos intermediários a complementação das funcionalidades ausentes. Dessa forma, podemos esperar complexidade crescente no desenvolvimento, conforme simplificamos os dispositivos da camada sensorial.

Além da questão da IoT-A, conforme o uso pretendido, o tratamento parcial dos dados pode ocorrer mais próximo aos objetos (*Edge* e *Fog Computing*) ou mais próximo à camada de negócios (*Cloud Computing*). Essa escolha impactará, igualmente, na engenharia de desenvolvimento. Vamos inicialmente estudar essas estruturas.

1.1 Computação em nuvem e CoT

A diversidade de usos e de objetos IoT resultaram na criação de vários ecossistemas sem integração. Além dessa necessidade de interoperabilidade, as implementações IoT, de grande porte, requerem funcionalidades, para gerenciar dados e dispositivos e garantir a entrega e o uso correto desses dados. Soluções, baseadas em computação em nuvem, foram as primeiras a serem propostas para equacionar a prestação de serviços e a interoperabilidade.

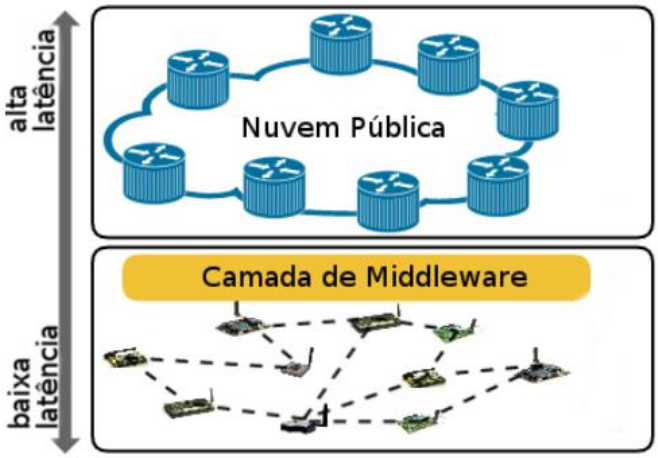
Nessa aproximação, a nuvem age como um transdutor e controlador. Ao transitar os dados de cada objeto para a nuvem, esta poderá garantir a segurança dos dados e a comunicação entre objetos.

A computação em nuvem, antes de sua aplicação ligada ao IoT, foi imaginada para permitir o uso de recursos computacionais oferecidos como serviços. Se a implantação é externa à rede privada do usuário, o modelo *pay-as-you-go* (pagamento por uso) permite flexibilidade e custo linear, proporcional ao crescimento da rede. A computação em nuvem, escalável e resiliente, se tornou a solução preferida pela maioria das aplicações *web*

Com a sedimentação das tecnologias ligadas ao IoT, a nuvem se tornou a opção ideal, para complementar a baixa capacidade computacional, inerente aos objetos inteligentes. Como já estudamos, objetos para domótica têm forte tendência ao uso de nuvem, tanto para operação isolada, com apoio de servidores do fabricante, quanto para uso integrado, via plataformas como Google e Amazon.

Esse paradigma, batizado CoT (*Cloud of Things* ou *nuvem de objetos*, em português), ilustrado na Figura 1, tem algumas vantagens interessantes, que discutimos quando falamos em domótica. A conexão dos objetos se dá ao *middleware* (direta ou indiretamente, conforme a IoT) e este se conecta diretamente à nuvem.

Figura 1 – CoT



Fonte: Santana *et al*., 2019, p. 81.

Como você já deve estar imaginando, a CoT tem algumas limitações severas que impedem o uso desse paradigma como único (Santana *et al*., 2019). Em soluções CoT, a conectividade com a nuvem, se perdida, impede o funcionamento do objeto. Outra fragilidade se refere à demanda por largura de banda, na nuvem, visto que todos os objetos a ela se conectam diretamente. Embora as demandas individuais sejam baixas, a copiosidade de objetos torna o trânsito total significativo, impactando no tempo de resposta, gerando alta latência.

Sistemas que devam funcionar permanentemente ou com baixa latência não serão aderentes à CoT. Uma solução que aproxima as facilidades de nuvem do objeto é a computação de neblina.

1.2 Computação de Neblina ou Névoa e FoT

Proposta por Bonomi *et al*. (2012) e Bar-Magen (2013), a arquitetura de computação de névoa prevê a realização de parte do processamento na rede

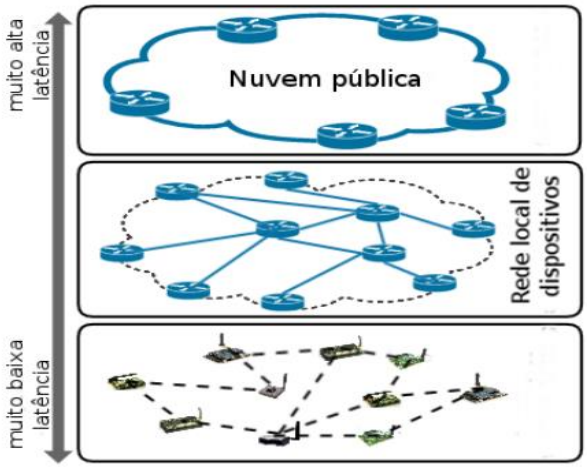
local. A princípio, o paradigma de computação em névoa (*Fog Computing*) é aplicável, como arquitetura, para qualquer rede de computadores. Quando consideramos o vertiginoso crescimento do tráfego de sistemas e redes IoT, entretanto, que em muitos casos já supera a demanda humana, percebemos o real potencial da névoa. Levar todos os dados para a nuvem e lá selecionar os que devem voltar para a camada de percepção e atuação parece uma arquitetura pouco lógica, diante dessa imensa geração de dados.

Nos primeiros estudos de aplicação em IoT do conceito de névoa (*FoT*, *Fog of Things*), imaginou-se o processamento realizado por alguns dispositivos de maior capacidade pertencentes à rede de objetos. Em uma implementação ZigBee, por exemplo, os nós roteadores poderiam assumir certo processamento local. A ideia evoluiu para que um concentrador tivesse capacidade de processamento e, consequentemente, de tratamento de dados e, eventualmente de decisão. A conexão com a nuvem se faz a partir desses concentradores.

O conceito de computação em neblina é, no entanto, distribuído, prevendo o uso da capacidade computacional onde ela for encontrada. Assim, uma rede de ativos composta de roteadores, *gateways*, servidores etc., com capacidade de processamento local, pode estar associada à camada de percepção.

O FoT não necessariamente elimina a necessidade da conexão à nuvem, mas presta os serviços locais possíveis e envia dados processados para as infraestruturas virtuais. A proximidade com os objetos facilita a obtenção de baixa latência. A Figura 2 ilustra o paradigma de neblina.

Figura 2 – Computação de neblina



Fonte: Santana *et al*., 2019, p. 82.

A consequência desse ganho de latência se reflete em possibilidades de uso em tempo real, mobilidade e escalabilidade. Há, ainda, um ganho lateral de resilência, resultante da autonomia dos dispositivos, em relação à conectividade com a nuvem. Se existir a necessidade de complementariedade de serviços, providos pela nuvem, os dados podem ser armazenados em neblina para envio posterior.

1.3 Computação de borda

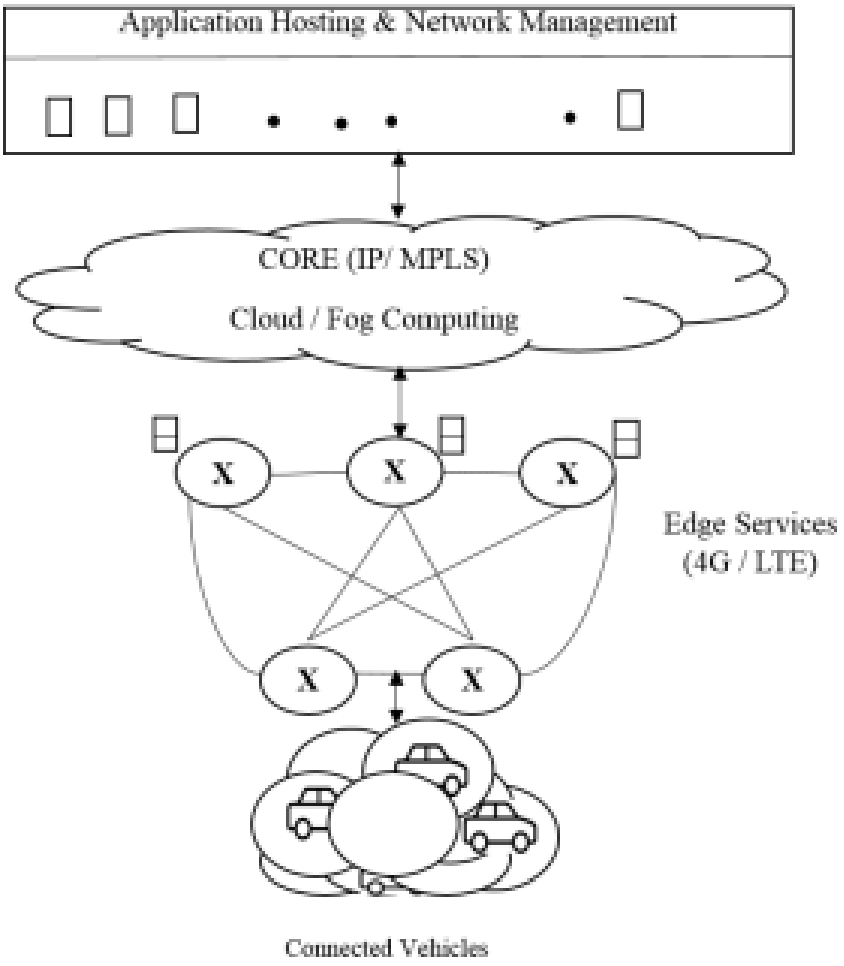
A computação de borda (*Edge Computing*, em inglês) é o paradigma arquitetural que devolve a capacidade de processamento para os objetos. Nessa arquitetura, no entanto, não retornamos apenas a uma LAN de pequenos processadores, como inicialmente podemos imaginar, embora isso seja possível em determinados usos.

Essa arquitetura prevê, de fato, a possibilidade de computação distribuída na borda. Além da virtual independência de conexão com a internet ou nuvem, outras facilidades se tornam viáveis, como baixa latência, mobilidade dos dispositivos e segurança de borda.

O tempo de processamento de soluções, envolvendo decisão centralizada em nuvem ou neblina, se tornou problemático para aplicações em *Vanet* (*Vehicular Ad Hoc Networks*), ou seja, quando veículos autônomos necessitavam tomar decisões em milissegundos (El-Sayed *et al*., 2017)

A arquitetura não abandona a camada de neblina, de forma análoga ao que comentamos em FoT, dando a ela funções de provimento de serviços para os objetos. Parte do processamento, entretanto, volta a ser realizado nos objetos. A Figura 3 ilustra a topologia.

Figura 3 – Topologia *Edge* para IoT



Fonte: El-Sayed *et al.*, 2017, p. 1709.

O conceito de computação de borda em IoT possibilitou, além de veículos inteligentes, implementações como monitoramento de tráfego, semáforos inteligentes, *smart grid* e monitoramento de tubulações de gás, por exemplo.

1.4 Introdução à programação para IoT

Entendida a diversidade de paradigmas de computação, associada à miríade de objetos, com variados níveis de inteligência e arquitetura, podemos construir nosso pensamento, quanto aos métodos de programação necessários para esses objetos.

O problema inicial repousa sobre o próprio projeto de dispositivos: embora existam padrões de interfaces de rede, que estudaremos, nascem, frequentemente, objetos de baixa interoperabilidade. Esse problema, todavia, se

agrava exponencialmente quando tentamos desenvolver aplicações para gerenciá-los, uma vez que, ainda não existem padrões consolidados para as interfaces de *software*.

Todavia algumas características devem ser mantidas em foco sempre que projetamos um *software* para IoT. A primeira delas é o gerenciamento de energia: os programas devem levar em conta que boa parte dos dispositivos precisa sobreviver por anos, sustentado por uma pequena bateria autônoma. Os sistemas operacionais dos objetos normalmente assumem o gerenciamento de energia da conectividade e do processamento, mas os serviços não são conhecidos do SO, e algumas decisões, como desligamento das requisições ao objeto, só podem ser tomadas pelo sistema demandante.

Uma segunda característica importante é a *latência*. Mesmo em arquiteturas de borda, o acesso ao dispositivo é feito em janelas determinadas de tempo para permitir o adormecimento. Assim, certa tolerância à latência deve ser prevista. Outra característica, ligada à conectividade, são os objetos que operam em modo não conectado, transmitindo os dados em intervalos de tempo constantes, de forma independente de requisição. Esse modo de operação de dispositivos simples pode gerar problemas na camada de aplicação, com a chegada de dados de forma assíncrona às necessidades dessa camada. Será, então, necessário depurar e corrigir os dados.

Alguns dados em sensores multifuncionais ou em redes com sensores de tecnologias diversas, com a mesma função, podem demandar metadados que devem ser considerados nos algoritmos de depuração, para construir um *dataset* adequado.

Para agregar um pouco mais de complexidade, a IoT tem uma característica ubíqua: um objeto pode estar em mobilidade e gerar dados em redes diversas. A camada de aplicação precisa estar preparada para a ubiquidade.

Namiot e Sneps-Sneppe (2014, p. 25) enfrentam esse problema de forma sistemática. Como muitos sensores não suportam instruções ou comandos impedindo o interfaceamento por APIs (*Application Program Interfaces*), estratégias envolvendo DPIs (*Data Program Interfaces*) parecem sem mais genéricas, por permitirem o acesso também a esses objetos simples. Com uso

de DPIs, hipoteticamente seria possível padronizar o acesso a todos os sensores bem como a respostas destes que poderiam gerar um arquivo JSON simples.

Dessa forma, a aplicabilidade de programação em entidades, sejam APIs ou DPIs, sedimenta a ideia do uso de microsserviços reativos, como método prioritário de desenvolvimento para IoT. Logo iremos explicar melhor este conceito. Para que possamos aprofundar um pouco mais este assunto, precisamos, antes, conhecer alguns SOs típicos, residentes em objetos inteligentes.

TEMA 2 – SISTEMAS OPERACIONAIS PARA IoT

Após visitarmos as topologias computacionais e compreendermos seu impacto sobre a programação, precisamos conhecer os sistemas operacionais que controlam os dispositivos IoT. Sabemos antecipadamente que existem níveis diversos de inteligência ligados a esses dispositivos e, por tal motivo, devemos esperar sistemas operacionais simples e leves e, em casos extremos, resumidos a temporizadores e transdutores.

Excetuando-se esse caso extremo, como os objetos inteligentes possuem recursos limitados, os SOs devem consumir poucos recursos. Os dois principais sistemas operacionais dedicados para objetos inteligentes são Contiki e TinyOS, além do Android e algumas versões de Linux que podem ser orientadas à IoT.

Vamos agora conhecer esses sistemas operacionais de forma genérica.

2.1 Contiki OS e Contiki-NG

Os primeiros sistemas operacionais para IoT exigiam do programador bom conhecimento de linguagens de baixo nível, como Assembly e normalmente demandavam algumas incursões no *opcode* hexadecimal. O Contiki surgiu como uma opção a estes SOs rústicos, em 2006. Possui bibliotecas que permitem a operação com várias soluções de conectividade, além da alocação e controle de memória. Do ponto de vista de conectividade, o Contiki foi o primeiro SO compatível com o protocolo IP e capaz de tratar endereçamento IP V6 (µIPv6). Este SO, em seu desenvolvimento original, tem código extremamente enxuto com apenas 100KB e necessita de parcos 10KB de memória volátil para operação.

Seu *kernel* foi desenvolvido em código aberto, utilizando a linguagem C, tornando-o leve e portável.

Saiba mais

O sistema operacional Contiki está disponível para estudo no *link* a seguir:

GIT HUB. The Contiki Open Source OS for the Internet of Things. **Git Hub**, S.d. Disponível em: <https://github.com/contiki-os>. Acesso em: 5 out. 2022.

O *kernel* é orientado a eventos tornando a execução eficiente em termos de energia e rápida, em relação a eventos externos. Outra característica desse OS é permitir a conexão com APIs cooperativas, multitarefa.

Para permitir testes de carga desse SO foi desenvolvido um simulador de rede de dispositivos, batizado de Cooja (Contiki OS Java), que permite simular nós com distinta memória e número de interfaces.

Em 2017, uma nova versão do SO, batizada de Contiki-NG foi publicada (Meriksson, 2022). Essa versão permite o controle de objetos com processadores 32 *bits* e suporta padrões recentes de protocolos como o IEEE 802.15.4 TSCH, 6LoWPAN, 6TiSCH, RPL, CoAP, MQTT, and LWM2M.

A nova versão também é agnóstica em relação ao *hardware*, mas há *drivers* disponíveis para adaptá-lo a eletrônicas específicas. Apesar de conter várias atualizações de segurança, o uso do C como base de programação mantém certas vulnerabilidades, ligadas a alguns problemas clássicos do C, as quais comentaremos em outro momento.

A seguir, um exemplo do inevitável código “Olá Mundo” para Contiki. Observe que, por ser um sistema multitarefa, é necessário *startar* os processos desejados:

```
#include"contiki.h"
#include<stdio.h>

PROCESS(olaMundoProcess,"Ola mundo");
AUTOSTART_PROCESSES(&olaMundoProcess);

PROCESS_THREAD(olaMundoProcess,ev,data)
{
```

```
        PROCESS_BEGIN ( ) ;
                p r i n t f ( " Ola Mundo \ n " ) ;
        PROCESS_END ( );
}
```

2.2 TinyOS

Esse sistema operacional baseia sua arquitetura em entidades computacionais independentes, ditas componentes, ligadas aos serviços oferecidos pelo objeto. Esses componentes, que operam como classes de APIs, se dividem em três tipos básicos, a saber: *comandos* (requisições para execução de serviço), *tarefas* (serviços internos do processador) e *eventos* (que sinalizam o estado de um serviço).

Existem bibliotecas de APIs que facultam a programação modular e reaproveitável para esse SO.

Saiba mais

Assim como Contiki, este SO tem código aberto, que pode ser encontrado acessando o *link* a seguir:

TINYOS. Disponível em: <https://github.com/tinyos>. Acesso em: 5 out. 2022.

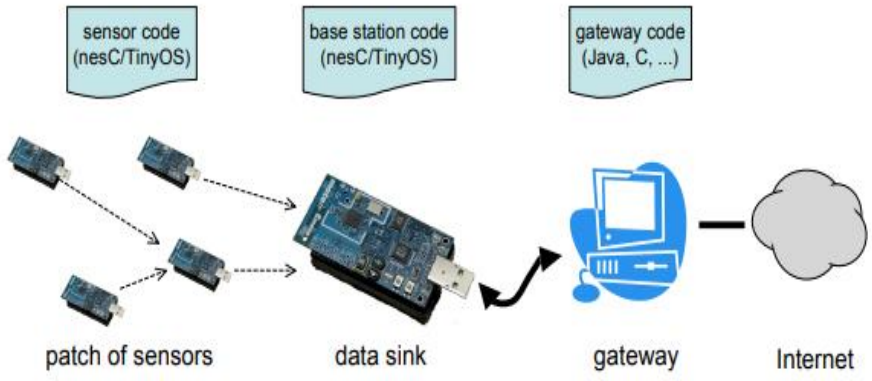
Há também um simulador de *hardware* que permite exercitar o uso do TinyOS mesmo na ausência de HW, o código deste simulador, chamado de Tossim, bem como instruções de uso podem ser encontradas acessando o *link* a seguir:

TINYOS. Tossim. **Tinyos**, S.d. Disponível em: <http://tinyos.stanford.edu/tinyos-wiki/index.php/TOSSIM>. Acesso em: 5 out. 2022.

Esse simulador é, de fato, uma biblioteca do SO.

TinyOS foi desenvolvido em nestC, uma variação do C, com foco em sistemas embarcados, LPWAN e redes de sensores. Imaginou-se uma rede de sensores extremamente simples conectados a um *gateway*, mas parte do HW necessário aos sensores reside em uma eletrônica centralizada, como se vê na Figura 4. Esse tipo de IoT-A já foi descrito e tem a vantagem de reduzir o custo total da rede, permitindo potências de rádio transmissão mais baixas.

Figura 4 – IoT-A para sensores com TinyOS



Fonte: Levis; Gay, 2009, p. 4.

O foco desse sistema operacional é o consumo extrabaixo de energia na camada de percepção (nós sensores). O uso de APIs, associado a essa estratégia, permite o adormecimento dos nós por até 99% do tempo de operação (Levis; Gay, 2009, p. 7).

O desenvolvimento de uma aplicação pode ser feito em linguagens de mais alto nível e traduzidas para nestC. Dada a simplicidade dos nós para os quais o SO foi desenvolvido, todavia, essa estratégia pode resultar em APIs pouco otimizadas, que consumirão recursos excessivos dos nós. Após concluído o código em nestC, a implantação em um nó é feita pelo *download*, com uso da interface USB, suportada por um computador com o IDE Tiny instalado.

2.3 Android

Esse SO não foi, como os anteriores, desenvolvido para objetos IoT e sim para dispositivos móveis de maneira geral. Como as tecnologias para esses dispositivos, principalmente em telefonia celular, estão bastante sedimentadas, é de se esperar que o ambiente, em torno do Android, também seja estável e rico em opções. De fato, há muitas APIs, bibliotecas e mesmo *midlewares* para ele.

Esse SO é baseado em um *kernel* Linux empilhado sob bibliotecas nativas do Android e uma máquina virtual (*Android Runtime*) que gerencia a execução das APIs. Acima dessa camada está o *framework* de serviços que isola a base da pilha, fornecendo serviços de alto nível para os desenvolvedores. Por

não ser um sistema operacional dedicado a LPWANs e objetos IoT, apresenta limitações de uso, exigindo dispositivos de maior complexidade e disponibilidade de recursos.

2.4 Linux

O desenvolvimento do mundo IoT atraiu adeptos dos sistemas operacionais Linux, tradicionalmente leves em relação a outros SOs, para uso em máquinas computacionais. Surgiram então distribuições Linux adaptadas ou adaptáveis ao IoT, como o Ubuntu Core, uma versão do Ubuntu otimizada para sistemas embarcados.

Saiba mais

Baseada em *containers*, essa distribuição apresenta alta resiliência e boa segurança, permitindo a configuração remota dos objetos (mais informações podem ser obtidas em:

UBUNTU. Disponível em: <https://ubuntu.com/core>. Acesso em: 5 out. 2022.

Saiba mais

Outra distribuição digna de citação é a RIOT, desenvolvida por acadêmicos, *hobbistas* e empresas, tem foco em IoT de baixo recurso, como os SOs Contiki e Tiny, mas, por ser uma versão do Linux, permite o desenvolvimento de APIs em várias linguagens. O *site* acessível no *link* a seguir possui informações sobre as facilidades e implantação deste SO:

RIOT. Disponível em: <https://www.riot-os.org/>. Acesso em; 5 out. 2022.

Cabe ainda citar o Raspian, baseado no Debian, mas nesse caso com foco em uma placa IoT específica, a Raspberry. Esse SO é bastante popular entre os desenvolvedores e prototipadores por possuir milhares de APIs, já compiladas, de fácil uso, disponíveis e convenientemente documentadas.

TEMA 3 – COMPUTAÇÃO ORIENTADA A SERVIÇOS

Comentamos, sem justificar convenientemente, que uma boa aproximação para programação de dispositivos IoT se dá por meio do uso de microsserviços reativos. Embora possamos intuitivamente compreender a expressão é importante que lhe demos uma abordagem acadêmica. Vamos então aos fundamentos da programação reativa e de microsserviços.

3.1 Introdução a SOC

SOC (*Service-Oriented Computing*) ou *computação orientada a serviços*, é um paradigma de programação, que tem por base a disponibilização, como serviços, de funcionalidades aplicacionais independentes.

Classicamente, dividem-se as funcionalidades da SOC em três camadas, iniciando nos serviços propriamente ditos, como camada de base. Sobre essa camada se estrutura a comunicação e a agregação entre os serviços, composta por um conjunto de sobresserviços chamados *web services*. O gerenciamento dos serviços compõe a camada superior.

A camada de agregação permite a construção de macrosserviços compostos pelos serviços de base. Essa agregação pode se dar por duas soluções principais: *web services* SOAP (*Simple Object Access Protocol*) e *web services restful*. No primeiro caso, as interações ocorrem por chamadas entre os serviços de base formatadas em XML. A segunda solução, mais recente, segue a arquitetura *Representational State Transfer* (REST), que cria métodos padronizados (PUT, POST, GET e DELETE) de troca de mensagens entre serviços, utilizando HTTP no paradigma *stateless* (não há troca ou armazenamento de contexto, que permanece no cliente).

3.2 Programação reativa

Programação reativa é uma estratégia de programação SOC que leva em conta os fluxos de dados durante a operação do código, permitindo serviços eficientes, responsivos e elásticos, estáveis mesmo sobre alta densidade de requisições. Os primeiros códigos baseados em FRP (*Functional Reactive Programming)* foram criados para controle de robôs. Esse tipo de controle precisa lidar com atuadores e sensores contínuos, como motores e componentes discretos, a exemplo dos processadores. Dessa forma, a comunicação não pode

ser plenamente conectiva, ou seja, os dados e comandos precisam ser gerados de forma relativamente assíncrona a sua leitura, pelo processador, ou execução, pelos atuadores.

Nesse paradigma, esperam-se execuções assíncronas de serviços motivadas pelos fluxos de dados. Voltando ao exemplo do robô, as ações serão provenientes de decisões baseadas em leituras do meio. Essas leituras podem se assíncronas em relação à execução do programa de controle.

De forma geral, pode-se dizer que, nesse tipo de programação, são geradas passagens assíncronas de dados entre serviços. A liberação do programador para esse paradigma não síncrono permite a estabilidade de aplicação, mesmo em ambientes pouco previsíveis.

Sistemas reativos não são propriamente uma novidade. A arquitetura de *software* em ambientes distribuídos baseia-se sobre programação orientada a serviços. De acordo com esse paradigma, é necessário dissociar transmissor e receptor de uma mensagem, criando-se um espaço de armazenamento virtual das mensagens. Esse espaço é, de fato, um endereço distribuído nos servidores ou objetos que compõem a rede. Naturalmente a escolha do armazenador deve estar ligada à resiliência relativa intrínseca deste.

Como em redes IoT a assincronicidade é importante para permitir o adormecimento dos objetos, a programação reativa é um modelo de desenvolvimento bastante aceitável.

3.3 Microsserviços

Segundo Namiot e Sneps-Sneppe (2014, p. 25), a ideia de um microsserviço é o particionamento de uma API em uma coleção de pequenos segmentos de código, independentes, que contenham um serviço ou parte segregável de um serviço. Cada serviço conterá seus processos, mantendo a independência e a autonomia. Um mecanismo leve de comunicação, ao estilo do HTTP, será implementado para permitir a conexão entre serviços, pelo armazenamento das mensagens necessárias a serem trocadas. Por esse método, a camada de gerenciamento dos microsserviços será independente e agnóstica em relação aos serviços gerenciados. A arquitetura de microsserviços é baseada também em SOC, embora, no caso dos microssserviços, ocorra uma

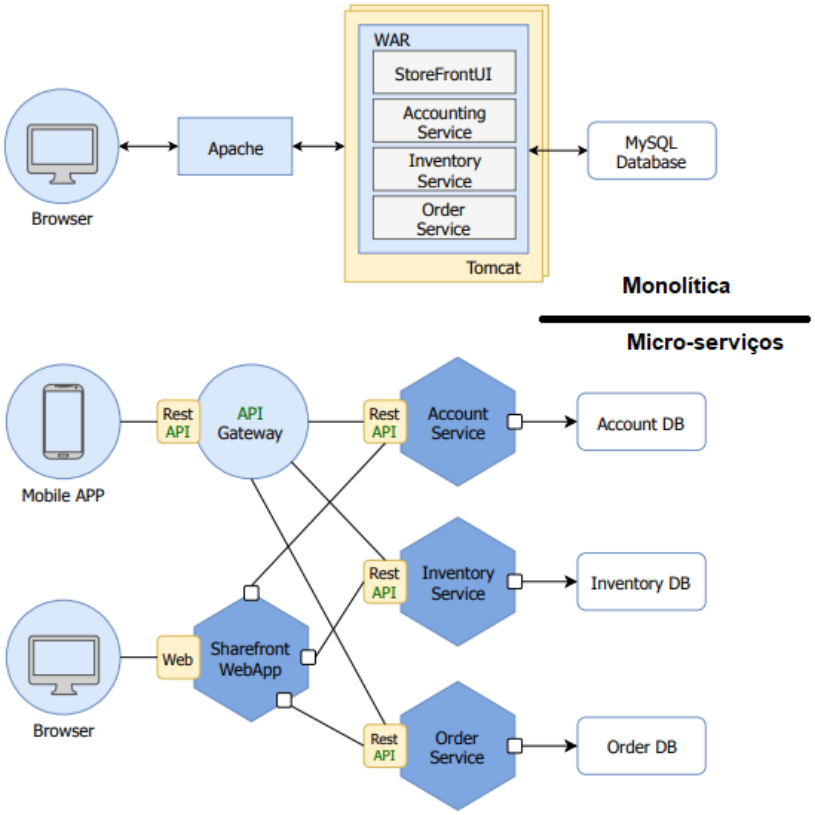
redução a apenas um serviço independente, e não a uma coleção de serviços, como em uma API.

Há várias vantagens no uso dessa estratégia, inclusive quanto à manutenção do código. Afirma Carrusca (2018, p. 2) que

> uma aplicação composta por microsserviços permite corrigir um problema localizado num dado microsserviço, sem que para isso toda a aplicação fique indisponível. Isso facilita também a adição de novas funcionalidades, pois é necessário apenas desenvolver e realizar o *deploy* de um novo microsserviço sem que para isso existam conflitos com os já presentes.

Vamos comparar, para melhor entendimento, um exemplo de aplicação monolítica, em relação a outra, estruturada em microsserviços, sugerido por Carrusca (2018, p. 10). Suponha uma aplicação de *e-commerce* que, ao receber um pedido de usuário, verifica o estoque, o crédito do usuário e finalmente faz o despacho. A Figura 5 mostra essa aplicação no paradigma monolítico e, em seguida, com uso de serviços.

Figura 5 – Arquitetura monolítica *versus* microsserviços



Fonte: Carrusca 2018, p.10-11.

Observe que, no modelo de serviços, a responsividade e a elasticidade são maiores, mas há também desvantagens associadas a essa arquitetura. A *performance* pode ser um problema se um serviço utilizar chamadas a outros serviços encadeados, resultando em latência alta. *Microsserviços reativos* contornam esse problema pela atualização assíncrona dos dados, na camada de comunicação. Essa estratégia agrega resiliência ao sistema. Naturalmente, um novo problema surge dessa solução: *manter a consistência dos dados*. A camada de gerenciamento deverá implementar controles para contornar a inconsistência temporal, derivada da assincronicidade da comunicação entre serviços.

TEMA 4 – ARQUITETURA DE MICROSSERVIÇOS REATIVOS PARA IOT

Já sabemos que microsserviços reativos são um bom paradigma de programação para IoT, contemplando estruturas de computação em borda, névoa ou nuvem transparentemente. Vamos agora estudar como uma arquitetura de microsserviços reativos pode ser concebida eficientemente

4.1 Comunicação

Como já discutimos, uma aplicação IoT, no paradigma de microsserviços, será composta por um conjunto de microsserviços (μS) reativos, distribuídos em dispositivos e servidores, localizados na borda da rede, na névoa ou na nuvem. Podemos dividir a comunicação, nessa arquitetura, em duas possibilidades: *externa* e *interna*.

4.1.1 Comunicação externa

Para que os μS se comunique com um cliente, a camada de agregação pode implementar essa conexão diretamente, serviço a serviço, ou criando um *gateway*, que fornecerá a comunicação também como um serviço.

Na primeira aproximação, o cliente faz o pedido diretamente ao μS, utilizando um *endpoint* público. Como a granularidade dos μS é grande, um cliente muito provavelmente precisará realizar inúmeras requisições, a vários μS, para obter os dados pretendidos.

Na segunda técnica, cria-se um *gateway*, como ponto único de acesso aos µSs. Esse *gateway* oculta, para o cliente, a arquitetura interna do sistema, fornecendo uma API para cada tipo de cliente, que agregará os dados necessários. Essa aproximação simplifica o código do cliente e padroniza o acesso à camada de percepção.

4.1.2 Comunicação interna

Para que os µSs se comuniquem entre si, será necessário implementar um modelo de interação entre processos. Assim como em redes de computadores, a interação pode ocorrer de *um para um* ou de *um para muitos*. O modelo pode ser implementado ainda prevendo a comunicação assíncrona, por *buffers* – nesse caso o µS solicitante fará uma *notificação* ao µS destino, ou uma *publicação*, no caso de solicitação a múltiplos µSs. A comunicação síncrona também pode ser utilizada no modelo pedido/resposta, mas essa estratégia é pouco prática para IoT, pelos motivos já expostos.

4.2 Gestão de dados

Percebemos que o uso de µS reativo pode ter por consequência uma queda na consistência dos dados. Um dado de uma amostragem anterior pode ainda estar presente quando se inicia uma nova coleta por uma API.

Requisitar o *refresh* dos dados não é uma boa aproximação, uma vez que gera tráfego desnecessário e consumo de energia dos dispositivos, obrigados a sincronizar as informações. Uma estratégia de publicação voltada a eventos, reativa, é uma solução boa, como já percebemos. Um serviço somente publicará seus dados em caso de alteração nas suas tabelas internas de dados. Os outros microsserviços, interessados nesses dados, os subscreverão, executando procedimentos motivados pela atualização, se necessário. Esse processo é realizado por um componente denominado *Message Broker*, que fará então um *query* na tabela do serviço.

Um mecanismo alternativo similar é o *Event Store*. Nesse caso, é criada uma tabela com apenas as alterações sofridas (eventos).

4.3 Descoberta de serviços

Em arquiteturas de μSs reativos, podem surgir novos componentes, ou as localizações dos μS podem ser alteradas. Um componente de gerenciamento, chamado *Service Registry*, constrói uma base de dados, com a identificação e localização de cada μS.

Quando a descoberta de um μS precisa ser feita por um cliente externo, dois são os procedimentos possíveis: contato direto com *Service Registry* ou uso de um *Load Balencer*, que interfaceará esse contato. Neste último caso, a este componente se dará também a faculdade de, caso existam serviços semelhantes, balancear a carga de requisições. Naturalmente, neste segundo caso, há necessidade da criação e manutenção desse componente de balanceamento, que deverá possuir alta disponibilidade, uma vez que se tornará o caminho crítico da disponibilidade do sistema.

4.4 *Containers*

Em computação de borda, uma boa estratégia para conter o uso de recursos computacionais de um microsserviço é a virtualização por *containers*.

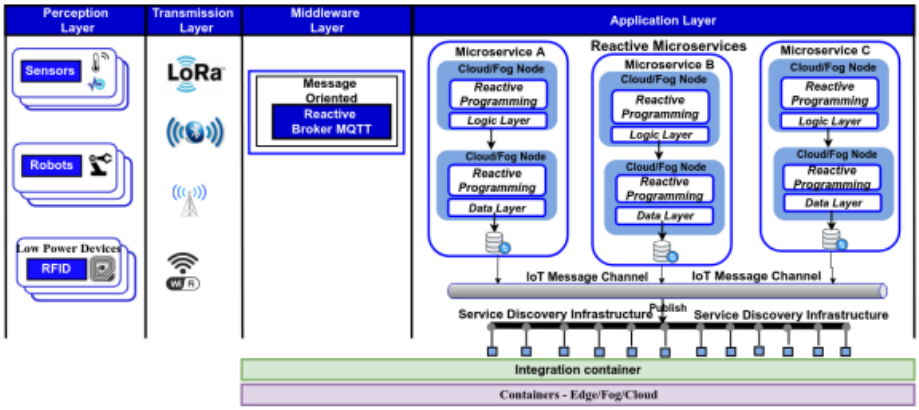
A ideia de um *container* é a concentração, em num único pacote, de todas as dependências, em termos de bibliotecas e definições, para a execução de um serviço ou aplicação. Essa técnica potencialmente reduz o uso de recursos da máquina, já que restringe a operação do serviço ao próprio container. Esse recurso é especialmente útil quando o desenvolvimento do código do serviço é feito em linguagens interpretadas.

A máquina que executa o *container* deverá naturalmente rodar um motor de *container*, como *Mesos Containerizer* ou o famoso *Docker*. Quando desejamos rodar vários *containers*, em uma mesma máquina, sistemas de orquestração, como o *Kubernetes*, controlam a execução de cada serviço e cuidam do provisionamento dinâmico de recursos, buscando-os em outras máquinas, inclusive, quando necessário

Essa estratégia é especialmente importante para microsserviços, permitindo a abstração dos serviços em relação à máquina e o controle efetivo do uso de recursos desta, fato bastante importante em redes IoT.

Uma visão geral da arquitetura de microsserviços, proposta por Santana *et al*. (2019), pode ser vista na Figura 6:

Figura 6 – Visão geral de arquitetura de microsserviços para IoT



Fonte: Santana *et al*., 2019, p. 95.

TEMA 5 – SDN E IOT

Redes de computadores (ou objetos) com arquiteturas definidas dinamicamente por *software* (SDN – *Software Defined Networks*) são um assunto bastante complexo, que certamente não poderá ser descrito suficientemente neste estudo. Por outro lado, o esforço pela criação de um processo de programação, para dispositivos e redes IoT, elástico, responsivo e resiliente, pode ser bastante simplificado, se a rede à qual esse dispositivo se conecta puder sofrer adaptações dinâmicas.

Dito de outra forma, se a percepção do objeto quanto à estrutura da rede se mantém constante, mesmo diante da entrada de dispositivos e ativos novos ou mesmo diante de mudanças de arquitetura física, o esforço para criação de aplicações tolerantes cai substancialmente. Controlar o uso e o compartilhamento de recursos da rede e de segurança é um exemplo de problemas que podem ser resolvidos por redes SDN, desonerando a programação dos serviços (Bera; Misra; Vasilakos, 2017).

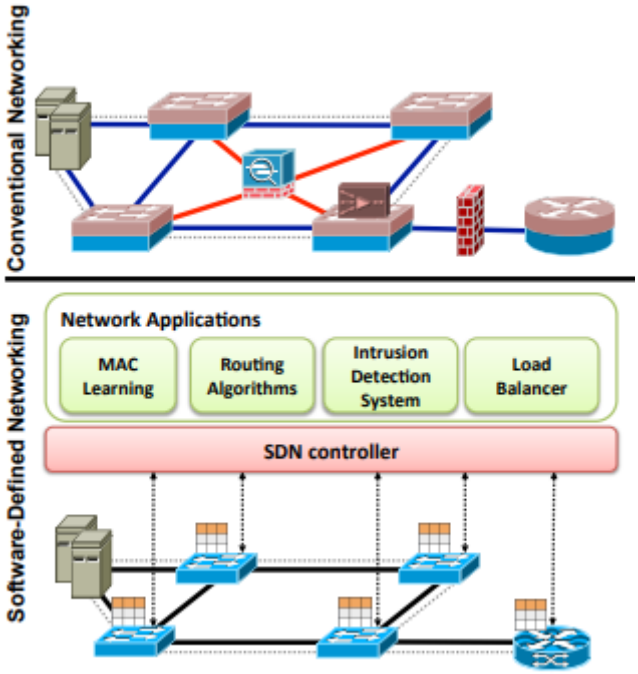
Dessa forma, vamos iniciar um estudo superficial sobre esse tema, que envolve tecnologias de ponta, tanto no aspecto de rede, quanto de dispositivos.

5.1 SDN

Redes definidas por SW são aquelas nas quais o encaminhamento dos pacotes, ou plano de dados, é desassociado do controle da arquitetura da rede,

ou plano de controle. Dessa forma, a configuração dos ativos de rede não é mais feita manualmente, a cada inserção na rede, mas controlada por *software*, de forma centralizada. A ilustração a seguir demonstra as diferenças entre aproximações.

Figura 7 – Rede tradicional *versus* SDN



Fonte: Kreutz *et al*., 2014, p. 5.

Embora, pela observação da figura, possamos imaginar a centralização física do controle da rede, a centralização é apenas lógica. Isso é explicável em função da existência de redes complexas e destruídas entre geografias.

Redes configuradas e controladas por SDN tratam problemas clássicos de congestionamento e vulnerabilidades de segurança do TCP com alta eficiência, adaptando a rede ao tráfego e à segurança, dinamicamente. A criação, por exemplo, de redes virtuais e de dutos seguros dentro da rede, que originalmente demanda a interferência, mesmo que remota, nos equipamentos, passa a ser delegada a uma inteligência centralizada de gerenciamento. O acréscimo de técnicas estatísticas de ML incorpora IA a rede, abstraindo completamente o plano de dados do seu gerenciamento humano. Exemplos de aplicações de ML em SDN são: predição de QoS e QoE (*quality of experience*), classificação de tráfego e gerenciamento dinâmico de segurança.

Saiba mais

Neste nosso estudo, não nos aprofundaremos além deste ponto, mas o estudo do artigo a seguir pode fornecer o aprofundamento se desejado.

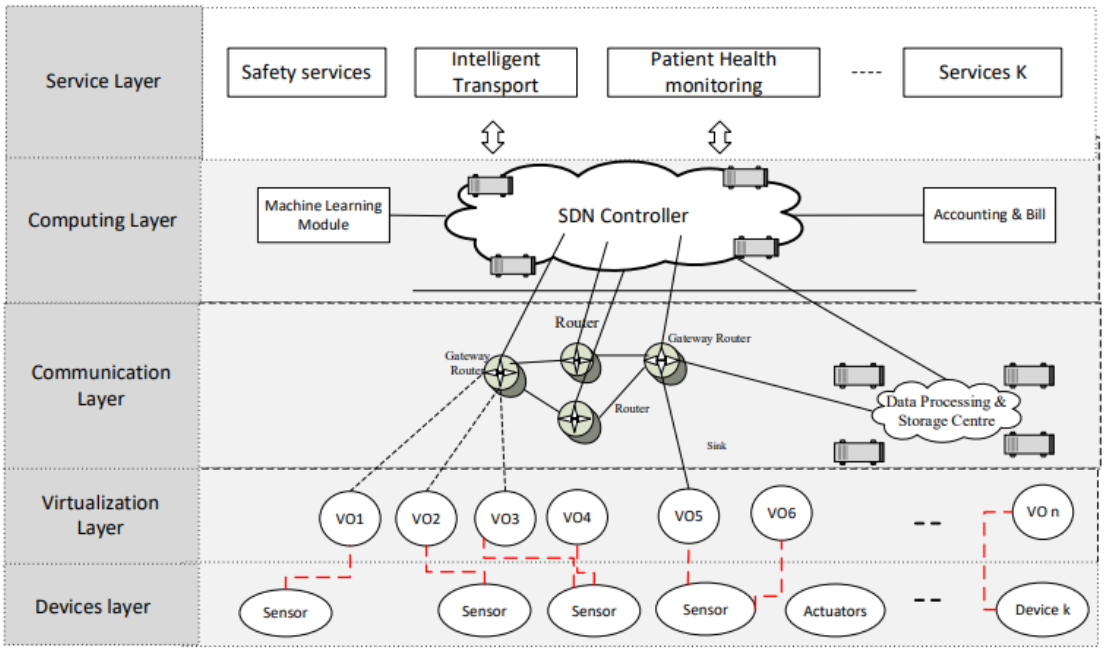
KREUTZ, D. *et al*. Software-defined networking: a comprehensive survey. **Proceedings of the IEEE**, v. 103, n. 1, p. 14-76, 2014.

5.2 SDN e IoT

Redes de objetos IoT sofrem uma série de limitações originadas nos próprios objetos ou nos protocolos heterogêneos de rede. Estudos envolvendo redes de sensores, a exemplo de Seol, Shin e Kim (2015), anteriores à sedimentação do conceito de cidades inteligentes, já propunham a criação de uma *software-defined wireless sensor network* (SDWSN), como forma de contornar problemas de interoperabilidade, delegando o controle dos *gateways* a uma inteligência central.

Em cidades inteligentes, o trânsito de dados, proveniente da borda em direção à nuvem, é considerável e assimétrico. Em relação ao tempo, configurações fixas de rede exigem o dimensionamento pelo pico de transmissão, levando a um investimento não ótimo em ativos e conectividade. A Figura 8 ilustra a contribuição que redes definidas por SW podem fazer às estruturas de IoT.

Figura 8 – SDN em IoT



Fonte: Ghaffar *et al*. 2021, p. 15.

A conexão entre borda e nuvem e a harmonização da heterogeneidade não são as únicas contribuições possível de sistemas SDN; a elasticidade, a mobilidade e a segurança também se vêm facilitadas com o emprego de inteligência no gerenciamento de rede.

Saiba mais

Infelizmente, não podemos nos aprofundar além deste ponto neste estudo, mas os artigos a seguir podem fornecer o aprofundamento se desejado:

BERA, S.; MISRA, S.; VASILAKOS, A. V. Software-defined networking for internet of things: a survey. **IEEE Internet of Things Journal**, v. 4, n. 6, p. 1994-2008, 2017. Disponível em: <https://bit.ly/3e68BS3>. Acesso em: 5 out. 2022.

GHAFFAR, Z. *et al*. A topical review on machine learning, software defined networking, internet of things applications: Research limitations and challenges. **Electronics**, v. 10, n. 8, p. 880, 2021. Disponível em: <https://www.mdpi.com/2079-9292/10/8/880>. Acesso em; 5 out. 2022.

FINALIZANDO

Neste capítulo, conhecemos os métodos recomendados de desenvolvimento de *software* voltados para IoT. Esses paradigmas complexos de programação, naturalmente, são aventados para aplicações de larga escala para as quais não apenas a programação precisa ser minuciosamente planejada, mas também as estruturas de rede. Considerando a heterogeneidade das tecnologias em torno dos objetos IoT, as redes que os atendem precisam ser dotadas de boa dose de inteligência e flexibilidade. Boa parte dessa heterogeneidade advém, entretanto, de protocolos particulares de conectividade entre objetos. Em outro momento, vamos conhecer algumas dessas soluções.

REFERÊNCIAS

BAR-MAGEN, J. Fog computing: introduction to a new cloud evolution. In: CASALS, J. F. F. (ed. lit.); NUMHAUSER, P. (ed. lit.). **Escrituras silenciadas**: paisaje como historiografía. Alcalá: Editorial Universidad de Alcalá, 2013. p. 111-126.

BERA, S.; MISRA, S.; VASILAKOS, A. V. Software-defined networking for internet of things: a survey. **IEEE Internet of Things Journal**, v. 4, n. 6, p. 1994-2008, 2017.

BONOMI, F. *et al*. Fog computing and its role in the internet of things. In: GERLA, M.; HUANG, D. Proceedings of the first edition of the MCC workshop on Mobile cloud computing. In: SIGCOMM '12 CONFERENCE, Helsinki: : ACM SIGCOMM, 17 ago. 2012, p. 13-16.

CARRUSCA, A. V. **Gestão de micro-serviços na Cloud e Edge**. Dissertação (Mestrado em Engenharia Informática) – Faculdade de Ciências e Tecnologia, Lisboa, 2018.

EL-SAYED, H. *et al*. Edge of things: The big picture on the integration of edge, IoT and the cloud in a distributed computing environment. **IEEE Access**, v. 6, p. 1706-1717, 2017.

GHAFFAR, Z. *et al*. A topical review on machine learning, software defined networking, internet of things applications: Research limitations and challenges. **Electronics**, v. 10, n. 8, p. 880, 2021.

LEVIS, P.; GAY, D. **TinyOS programming**. Cambridge: Cambridge University Press, 2009. Disponível em: <http://csl.stanford.edu/~pal/pubs/tos-programming-web.pdf>. Acesso em: 5 out. 2022.

KREUTZ, D. *et al*. Software-defined networking: a comprehensive survey. **Proceedings of the IEEE**, v. 103, n. 1, p. 14-76, 2014.

MERIKSSON, J. Version 4.8. **Contik-Ng**, 14 jul. 2022. Disponível em: <https://github.com/contiki-ng/contiki-ng/releases>. Acesso em: 5 out. 2022.

NAMIOT, D.; SNEPS-SNEPPE, M. On iot programming. **International Journal of Open Information Technologies**, v. 2, n. 10, p. 25-28, 2014.

SANTANA, C. *et al*. Teoria e prática de microsserviços reativos: um estudo de caso na Internet das Coisas. In: XXV SIMPÓSIO BRASILEIRO DE SISTEMAS MULTIMÍDIA E WEB: MINICURSOS. **Anais**... Sociedade Brasileira de Computação, 2019.

SEOL, S.; SHIN, Y.; KIM, W. Design and realization of personal IoT architecture based on mobile gateway. **International Journal of Smart Home**, v. 9, n. 11, p.133-144, 2015.