UNIVERSIDADE FEDERAL DE ITAJUBÁ - UNIFEI
PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA ELÉTRICA

Serviço de reconfiguração de dispositivos IoT.

Miguel Marques de Paiva Esper

Itajubá, 14 de fevereiro de 2023

UNIVERSIDADE FEDERAL DE ITAJUBÁ - UNIFEI
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Miguel Marques de Paiva Esper

Serviço de reconfiguração de dispositivos IoT.

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica como parte dos requisitos para obtenção do Título de Mestre em Ciências em Engenharia Elétrica.

Área de Concentração: Microeletrônica

Orientador: Prof. Dr. Danilo Henrique Spadoti

14 de fevereiro de 2023
Itajubá

UNIVERSIDADE FEDERAL DE ITAJUBÁ - UNIFEI
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Serviço de reconfiguração de dispositivos IoT.

Miguel Marques de Paiva Esper

Dissertação aprovada por banca examinadora em 14 de Dezembro de 2022, conferindo ao autor o título de **Mestre em Ciências em Engenharia Elétrica.**

Banca Examinadora:
Prof. Dr. Gabriel Lobão Vasconselos Fré
Prof. Dr. Edvard Martins de Oliveira
Dr. Reinaldo Lima de Abreu

Itajubá
2022

Miguel Marques de Paiva Esper
Serviço de reconfiguração de dispositivos IoT/ Miguel Marques de Paiva Esper. – Itajubá, 14 de fevereiro de 2023-
112 p.

Orientador: Prof. Dr. Danilo Henrique Spadoti

Dissertação (Mestrado)
Universidade Federal de Itajubá - UNIFEI
Programa de pós-graduação em engenharia elétrica, 14 de fevereiro de 2023.

1. IoT. 2. Azure. I. Spadoti, Danilo Henrique. II. Universidade Federal de Itajubá. III. Serviço de reconfiguração de dispositivos IoT

CDU 07:181:009.3

Miguel Marques de Paiva Esper

Serviço de reconfiguração de dispositivos IoT

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica como parte dos requisitos para obtenção do Título de Mestre em Ciências em Engenharia Elétrica.

Trabalho aprovado. Itajubá, 14 de Dezembro de 2022:

Prof. Dr. Danilo Henrique Spadoti
Orientador

Prof. Dr. Gabriel Lobão Vasconselos Fré

Prof. Dr. Edvard Martins de Oliveira

Dr. Reinaldo Lima de Abreu

Itajubá
14 de fevereiro de 2023

Agradecimentos

A princípio, agradeço ao Prof. Dr. Gabriel Lobão Vasconcelos Fré pelo convite em fazer parte do Laboratório de Telecomunicações, incentivo no ambiente acadêmico, amizade e paciência ao longo dessa caminhada.

Ao Prof. Dr. Danilo Henrique Spadoti, pela aceitação em sua equipe de pesquisa, pela parceria nesse aprendizado e por sempre estar presente no decorrer do trabalho e também no âmbito pessoal.

Ao Dr. Reinaldo Lima de Abreu e Msc. Tales Henrique Carvalho pela disponibilidade e ajuda nas longas conversas sobre o futuro do trabalho e resolução de impasses encontrados.

Ao Msc. Lucas Victor Benjamin Vasconcelos Fré, pela amizade, conselhos e prestatividade na obtenção dos documentos acadêmicos em períodos de pandemia.

À Unifei pelo ambiente propicio ao desenvolvimento desse trabalho e à CAPES pela bolsa de estudos.

Ao meu irmão, Felipe Marques de Paiva Esper, por sempre estar presente apesar da distancia física e pelo incentivo constante em buscar o melhor sempre, uma de suas palavras "Você usa o mesmo tempo e esforço para criar uma coisa pequena ou uma coisa gigantesca". Ao meu pai, Roberval Silva Esper, por sempre me ensinar a correr atrás do que queremos, nada cai do céu. À minha mãe, Magda Marques de Paiva Esper, que mesmo não estando mais entre nós, é presente diariamente com sua alegria, entusiasmo e bom humor.

"A menos que modifiquemos a nossa maneira de pensar, não seremos capazes de resolver os problemas causados pela forma como nos acostumamos a ver o mundo".

(Albert Einstein)

Resumo

Esse trabalho apresenta um modelo para aprimorar o envio de telemetria de dispositivos IoT (*Internet of Things*), utilizando a ferramenta de Dispositivo Gêmeo presente na plataforma de serviços de servidores em nuvem da Microsoft, chamada Azure. O aprimoramento é feito através de um servidor, desenvolvido em Python, que interpreta a telemetria recebida, define o estado atual do dispositivo e atualiza as configurações do dispositivo emissor por meio do Dispositivo Gêmeo. O resultado para o dispositivo em seus parâmetros normais de operação foi uma redução de 91% na quantidade de pacotes, consequentemente no volume de dados trafegados na rede. Já fora de seus parâmetros normais, a quantidade de pacotes aumentou 145% e consequentemente o volume de dados trafegados pela rede também.

Palavras-chaves: IoT. Azure. Dispositivo Gêmeo. Hub IoT

Abstract

This work presents an optimization model for sending telemetry from IoT devices, using the Twin Device tool present in Microsoft's cloud server services platform, called Azure. The optimization is done through a server, developed in Python, which interprets the telemetry received, defines the current state of the device and updates the settings of the device and emits it through Twin Device. The result in normal operating states was a drastic decrease in the number of packets and consequently a smaller volume of data trafficked on the network, already in a critical state, the amount of packets increased and consequently a considerably greater volume trafficked through the network.

Key-words: IoT. Azure. Device Twin. Hub IoT

Lista de ilustrações

Figura 1 – Sistema típico IoT. 4
Figura 2 – Processamento de borda. 6
Figura 3 – Processamento de borda colaborativo. 6
Figura 4 – Estrutura do dispositivo gêmeo. 13
Figura 5 – Ciclo de vida de uma atividade. 19
Figura 6 – Exemplo de código em fluxo desenvolvido em Node-RED. 21
Figura 7 – Wireshark 22
Figura 8 – Fluxograma do serviço - Recebendo uma nova mensagem. 23
Figura 9 – Aplicativo Android. 27
Figura 10 – Aplicativo Android ao iniciar. 28
Figura 11 – Aplicativo Android - Comparação dos parâmetros. 29
Figura 12 – Fluxograma do painel de visualização. 38
Figura 13 – Painel quando serviço é iniciado. 39
Figura 14 – Painel quando o serviço recebe mensagem. 40
Figura 15 – Dispositivo Normal sem otimização. 43
Figura 16 – Dispositivo Normal com otimização. 44
Figura 17 – Dispositivo Crítico com temperatura alta sem otimização. 45
Figura 18 – Dispositivo Crítico com temperatura alta com otimização. 46
Figura 19 – Dispositivo Crítico com temperatura baixa sem otimização. 47
Figura 20 – Dispositivo Crítico com temperatura baixa com otimização. 48
Figura 21 – Dispositivo Super Crítico com temperatura alta sem otimização. 49
Figura 22 – Dispositivo Super Crítico com temperatura alta com otimização. 50
Figura 23 – Dispositivo Super Crítico com temperatura baixa sem otimização. 51
Figura 24 – Dispositivo Super Crítico com temperatura baixa com otimização. 52
Figura 25 – Gráfico comparando otimização no dispositivo normal. 53
Figura 26 – Pacotes enviados - Dispositivo Normal. 53
Figura 27 – Volume de dados - Dispositivo Normal. 54
Figura 28 – Gráfico comparando otimização no dispositivo crítico em baixa temperatura. 55
Figura 29 – Gráfico comparando otimização no dispositivo crítico em alta temperatura. 55
Figura 30 – Gráfico comparativo o envio de 20 pacotes com e sem otimização em estado crítico. 56
Figura 31 – Gráfico Volume de dados - dispositivo critico. 57
Figura 32 – Gráfico comparando otimização no dispositivo super crítico em alta temperatura. 58

Figura 33 – Gráfico comparando otimização no dispositivo super crítico em baixa temperatura. 58
Figura 34 – Gráfico comparando o estado crítico ao super crítico. 59
Figura 35 – Kit de desenvolvimento do NVIDIA Jerson Nano. 60
Figura 36 – Disposição dos testes. 62
Figura 37 – Experimento - Dispositivo Normal - Sem Serviço. 63
Figura 38 – Experimento - Dispositivo Normal - Com Serviço. 63
Figura 39 – Experimento - Dispositivo Normal - Comparação. 64
Figura 40 – Experimento - Dispositivo crítico - Sem Serviço. 64
Figura 41 – Experimento - Dispositivo Crítico - Com Serviço. 65
Figura 42 – Experimento - Dispositivo Esfriando - Com Serviço. 66

Sumário

1 INTRODUÇÃO 1
1.1 Revisão Bibliográfica 4
1.2 Objetivo 8
1.3 Organização do trabalho 8

2 FERRAMENTAS E TECNOLOGIAS 9
2.1 Gêmeo Digital 9
2.2 Linguagem JavaScript - JSON 10
2.3 Azure 11
2.3.1 Hub Iot 12
2.3.2 Dispositivo Gêmeo 12
2.4 Python 15
2.5 Android 16
2.6 Node-RED 19
2.7 Wireshark 21

3 DESENVOLVIMENTO DE UM SERVIÇO DINÂMICO 23
3.1 Dispositivo 24
3.2 Serviço dinâmico 29
3.3 Painel de informações 38

4 SIMULAÇÃO 41
4.1 Configurações 41
4.2 Dispositivo normal 42
4.3 Dispositivo crítico 44
4.4 Dispositivo Super Crítico 48
4.5 Resultados 52

5 EXPERIMENTAÇÃO 60
5.1 NVIDIA® Jetson Nano™ 60
5.2 Metodologia 61
5.3 Experimentos 62

6 CONCLUSÃO 67
6.1 Trabalhos futuros 67

APÊNDICES 69

APÊNDICE A – CÓDIGO EM PYTHON DO SERVIDOR INTELIGENTE 70

APÊNDICE B – CÓDIGO EM JAVA (ANDROID) QUE SIMULA O DISPOSITIVO IOT 79

APÊNDICE C – CÓDIGO NODE-RED DO PAINEL DE VISUALIZAÇÃO 92

APÊNDICE D – CÓDIGO EM PYTHON DA EXPERIMENTAÇÃO NO DISPOSITIVO NVIDIA JETSON 101

APÊNDICE E – TESTE WIRESHARK 104

REFERÊNCIAS 108

1 Introdução

Este Capítulo, inicialmente, apresenta uma visão geral sobre tema Internet Das Coisas (IoT),do inglês *"Internet of things"*, e apresenta a evolução de diferentes tecnologias. Também, são discutidos os problemas referentes a quantidade de informações que trafegam pelas redes e os gargalos que os atuais sistemas apresentam.

O termo *"Internet of things"*, ou IoT, remete à qualquer objeto físico conectado a *Internet* que pode trocar algum tipo de informação. Os objetos podem ser dispositivos estáticos ou objetos vivos, como: pessoas, animais e plantas. Nos próximos três parágrafos estão descritas três definições do termo IoT encontrado na literatura.

- Internet das Coisas (IoT - do inglês *"Internet of things"*) se refere à rede de dispositivos de uso cotidiano equipados com uma inteligência onipresente. Essa onipresença será amplificada com a integração de cada objeto via sistemas embarcados, gerando uma rede de objetos que se comunicam entre si e com seres humanos [1].
- Rede global e infraestrutura de serviço de densidade variável e conectividade com recursos de autoconfiguração baseados em protocolos e formatos padrões e interoperáveis que consistem em coisas heterogêneas que possuem identidade, atributos físicos e virtuais, e são integrados de forma perfeita e segura na Internet. Objetos e coisas inteligentes se tornarão participantes ativos em negócios, informativos e processos sociais onde serão ativos para interagir e comunicar entre si e com o ambiente [2].
- Um paradigma onde os objetos do cotidiano podem ser equipados com recursos de identificação, detecção, rede e processamento que permitirá que eles se comuniquem uns com os outros e com diferentes dispositivos e serviços pela Internet para atingir algum objetivo [3].

Diversas soluções estão sendo desenvolvidas, de modo a contribuir para diferentes segmentos do mercado de tecnologia. Destacam-se: segurança e monitoramento de ambientes, levantamento estatísticos de qualidade do ar e outros fatores energéticos [4, 5], controle de semáforos e sistemas de sinalização, iluminação pública [6, 7], gestão de resíduos em grandes centros [8], controle de qualidade do solo em lavouras e rastreamento de rebanhos, compartilhamento de veículos e gestão de transporte público. Deste modo, sistemas capazes de operar em todos estes segmentos, devem apresentar algumas características específicas, como o uso de banda estreita(100Hz do SigFox até 200KHz do NB-IoT [9]), otimização no consumo de energia (para dispositivos energizados por baterias que tem acesso restrito para sua troca [10]), baixa latência (40ms para o LoRaWan até 10ms

para o NB-IoT [11]), garantia de disponibilidade e confiabilidade [12], para que os permitam trabalhar com o grande volume de informação gerada, com intuito de catalogar e armazenar de forma ordenada e livre de redundâncias.

Para se conectar os bilhões de usuários e dispositivos em todo o mundo utiliza-se a *Internet*, que é uma rede mundial que utiliza protocolos padrões de comunicação baseados em TCP/IP. A Internet é uma rede pública, em que os pontos de acesso tem alcance entre eles, ou seja, ao incluir um ponto de acesso novo, todos os existentes terão acesso ao mesmo. Uma aplicação de IoT pode estar conectada a Internet, o acesso público à ela ficará ativo, ou pode ser feita uma rede privada, sem acesso à rede externa.

As redes classificadas como LPWAN (Low Power Wide Area Network - Rede de longa distância de baixa potência) apresentam características especificas, tais como: pequena largura de banda, longo alcance, alta penetração, alta confiabilidade e disponibilidade e baixo consumo de energia [13].

Além disso, deve-se prever o suporte a um número elevado, na grandeza de bilhões, de dispositivos interconectados com acesso de forma simultânea aos pontos de acesso a rede, que pode ser um *gateway*, um roteador, ou até mesmo uma antena de operadora. Ao estabelecer sistemas que sejam capazes de gerenciar quantidades elevada de dispositivos, com inteligência artificial que os permitam lidar com altos volumes de informação [14, 7, 15, 16].

Diversas tecnologias de rede foram desenvolvidas, testadas e implementadas em projetos de IoT, alguma delas são: NB-IoT, LoRaWAN, BLE, ZigBee, SigFox, dentre outras. Pode-se, resumidamente, descrevê-las como [6]:

Narrowband-IoT ou NB-IoT é utilizada em dispositivos de baixo consumo nas redes celulares, é baseada no esquema de modulação espalhamento espectral por sequência direta ou *Direct-Sequence Spread Spectrum* (DSSS). Possui uma banda utilizável de 180kHz, com taxa de dados de 200 kbps [17].

LoRaWAN é composto por três classes de dispositivos diferentes para requisitos de energia e de latência de *downlink* diferentes, atendendo requisitos específicos de aplicação. LoRaWAN é um protocolo de controle de acesso, desenvolvido para suportar redes de grande escala, na casa de bilhões, contendo LoRa (Long Range ou longo alcance) como sua camada física, ou seja, seu chip [18, 19].

Bluetooth Low Energy (BLE), ou *Bluetooth Smart*, é uma derivação do Bluetooth destinado a dispositivos com recursos limitados de energia. O BLE faz parte da pilha do Bluetooth v4.0 e da pilha do Bluetooth v4.2. Foi desenvolvido como protocolo de rede de área pessoal(PAN) para transmitir parte de dados entre frequências com taxa de transferência limitada baixo consumo por *bit* [20].

ZigBee, desenvolvido pelo ZigBee Alliance, é um padrão de rádio comunicação de

curto alcance para dispositivos embarcados e é desenvolvido com base em um protocolo de rede local (LAN). Foi desenvolvido inicialmente para controle e automação predial. Este padrão compreende alguns benefícios, como: baixa consumo, segurança avançada, robustez e alta escalabilidade com grandes quantidades de nós [21].

SigFox é uma operadora global de redes IoT. Sua cobertura fica entre o WiFi e o celular, utiliza as tecnologias de *Ultranarrow band* (UNB).Foi desenvolvido para lidar com baixas taxas de transferência, entre 10 e 1000 bps. A implantação é semelhante as redes celulares, com antenas em torres. O envio de dados do servidor para os sensores ou dispositivos (capacidade de *downlink*) é limitado e a interferência de sinal é um problema [22, 18].

Segundo a Cisco, uma das maiores empresas de soluções em redes e comunicações, em 2018 existia 6,1 bilhões de dispositivos IoT. Com o crescimento exponencial, a previsão é que em 2023 haverão 14,7 bilhões de dispositivos IoT conectados à Internet. Isso quer dizer que cada individuo da população global terá 1,8 dispositivos. Aplicações para casa conectada, como automação residencial, segurança residencial e câmeras de vigilância irão representar 48% dos dispositivos IoT [23].

Um sistema típico IoT é desenvolvido por três principais componentes [24], exposto na Figura 1:

- Dispositivo IoT, também chamado de nó, é responsável por coletar dados através de sensores e enviar essas informações ao servidor. Em alguns casos, o dispositivo pode realizar algum processamento antes de enviar ao servidor. Por exemplo, um sistema de controle de temperatura em uma sala com servidores, o nó pode, somente, enviar o dado de temperatura ao servidor, pode acionar um ar condicionado se a temperatura estiver alta ou um aquecedor caso a temperatura estiver baixa;

- Servidor IoT é responsável por receber e tratar os dados recebidos dos clientes (os dispositivos IoT). O servidor pode enviar comandos e dados aos dispositivos;

- Interface de usuário é normalmente um ambiente *cloud*, um sistema accessível via computador com conexão a Internet, ou móvel, aplicativo disponível para celulares *smartphones*, em que os usuários possam ler esses dados dos dispositivos e requisitar alguma ação perante eles.

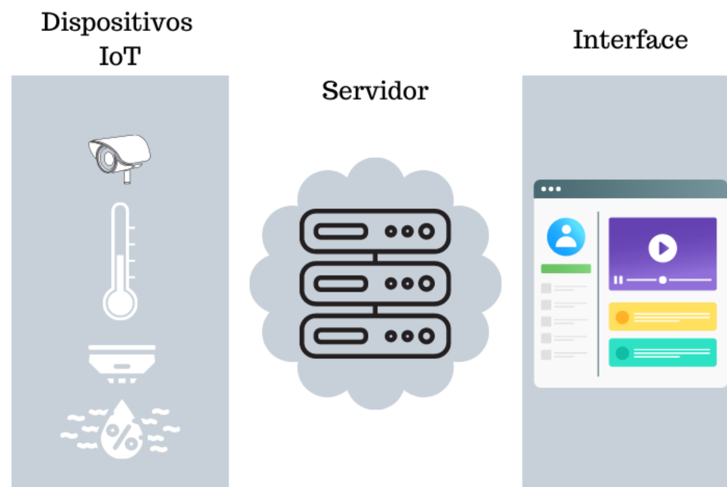

Figura 1 – Sistema típico IoT.

Neste cenário, a motivação e objetivo deste trabalho é reduzir o gargalo existente entre a comunicação Servidor-Dispositivo. O desenvolvimento do serviço inteligente capaz de receber informações dos dispositivos, independentemente da rede utilizada, pode aumentar a eficiência da rede de comunicação. O dispositivo IoT pode ser inteligente ou não, com a possibilidade de ter conexão com várias redes ou não. O servidor pode receber telemetria de dispositivos com a inteligência de receber a nova configuração que o servidor atribuir a ele ou dispositivos que não possuam essa inteligência, nesse caso ao receber mensagem desse dispositivo, o servidor tratará ela normalmente só que o dispositivo não sera otimizado. O servidor será capaz de interpretar que aquele pacote de informações originou de um dispositivo único. O servidor terá inteligencia para determinar como deverá reconfigurar os parâmetros de envio do dispositivo, usando os dados da mensagem enviada e a sua configuração atual. Isso poderá diminuir a complexidade do desenvolvimento de aplicações do lado servidor e possibilitar o desenvolvimento de dispositivos inteligentes com mais de uma rede disponível. O código do servidor terá possibilidade de adicionar informações de forma mais simples, baseada em códigos orientados a objetos.

1.1 Revisão Bibliográfica

Nos últimos anos, vários trabalhos vêm sendo desenvolvidos com intuito reduzir os gargalos nos sistemas de telecomunicações e, principalmente, no que tange a entrada e conexão de bilhões de dispositivos IoT. No artigo [25] foi utilizado um protocolo CoAP (Constrained Aplication Protocol) com objetivo de otimizar a transmissão de dados em um ambiente simulado de rede. Foram implementados um cliente e um servidor, com a linguagem de programação Java através da biblioteca ARMmbed. A simulação nesse artigo foi feita através de máquinas virtuais dentro do Mininet (emulador de rede). Os resultados foram obtidos através de *scripts* (uma série de instruções ordenadas) automati-

zados para comparar o protocolo CoAP com o HTTP. O protocolo CoAP teve um melhor desempenho que o protocolo HTTP (Hypertext Transfer Protocol) por contar com um cabeçalho mais enxuto. O CoAp utiliza o protocolo UDP (User Datagram Protocol) na camada de transporte, não necessitando de retransmissões, porém sem confiabilidade do destino receber os dados corretos e em ordem.

Madureira et. al [26] descreveram o protocolo IoTP (Protocolo da Internet das Coisas) com possibilidade de agregação de dados adaptados às tecnologias de comunicação Ethernet, IEEE 802.11 e outras utilizadas pelos dispositivos IoT. O protocolo proposto armazena os pacotes recebidos com um mesmo ID de serviço no *switch* IoTP até que o número de pacotes do mesmo serviço alcance o valor configurado de pacotes agregados e, nesse momento, o pacote agregado e gerado é enviado ao *gateway* IoTP. Existe uma outra forma de agregação nesse trabalho, associado ao campo Sync Flag. Quando esse campo está habilitado no pacote recebido pelo *switch* IoTP, o mesmo agrega os pacotes armazenados até o momento com o mesmo ID de serviço, gera o pacote agregado e envia o mesmo para o *gateway* IoTP. Os resultados do trabalho foram obtidos por emulação, pelo Mininet, e resultaram em 78% de melhora em eficiência de rede (utilização da rede) e reduziu o número de pacotes enviados pela rede.

Os dois trabalhos apresentam formas de reduzir o tráfego de dados da rede utilizando propostas de protocolos desenvolvidos especificamente para redes IoT. Em [25], utilizou-se a redução do cabeçalho de cada pacote enviado, e em [26] foi utilizada a agregação de pacotes de mesmo serviço para se obter a redução de dados.

O artigo de Marcelino et. al [27] busca, por outro lado, otimizar o consumo de baterias em *hardware* de IoT industriais. Para esse objetivo foram propostas duas abordagens: alterar o tempo de acesso à memória do microprocessador para obter os dados a serem enviados e alterar a quantidade de informação enviada para a rede de transmissão sem fio, com tratamento dos dados antes de serem enviados. O acesso à memória não está ligado à esse trabalho, mas a diminuição na quantidade de envio de dados está. A proposta é realizar um pré-processamento nas amostras disponíveis, no caso 6144 elementos pré-processamento em um intervalo de 0,3 segundos. São utilizadas algumas ferramentas e teoremas de sinais para desenvolver um algoritmo capaz de reduzir essas 6144 amostras para 100 principais. Com isso melhora o envio de informações e reduz o consumo de energia, sendo usado o processamento de borda para diminuir a quantidade de dados enviados para o servidor [27]. Já, S. Singh em seu artigo [28], utiliza uma nova abordagem de processamento de borda (*edge computing*) [29], em que o processamento é feito no *Gateway*, chamando ele de *smart gateway*. A abordagem padrão em IoT é computação em nuvem (*cloud computing*) [30]. Esse padrão gera uma quantidade de tráfego maior, pois todos os dados a serem processados são enviados na íntegra. No processamento de borda presente no artigo [28], o *Gateway* realiza algum processamento, com intuito de reduzir o

tráfego na rede, pois a quantidade de dados depois de processada tende a ser menor. As Figuras 2 e 3 presentes no artigo [28] mostram os diagramas do processamento de borda referente ao IoT e o diagrama do processamento de borda apresentado no artigo em que o GateWay pode lidar tanto com dispositivos IoT, quanto com dispositivos não IoT.

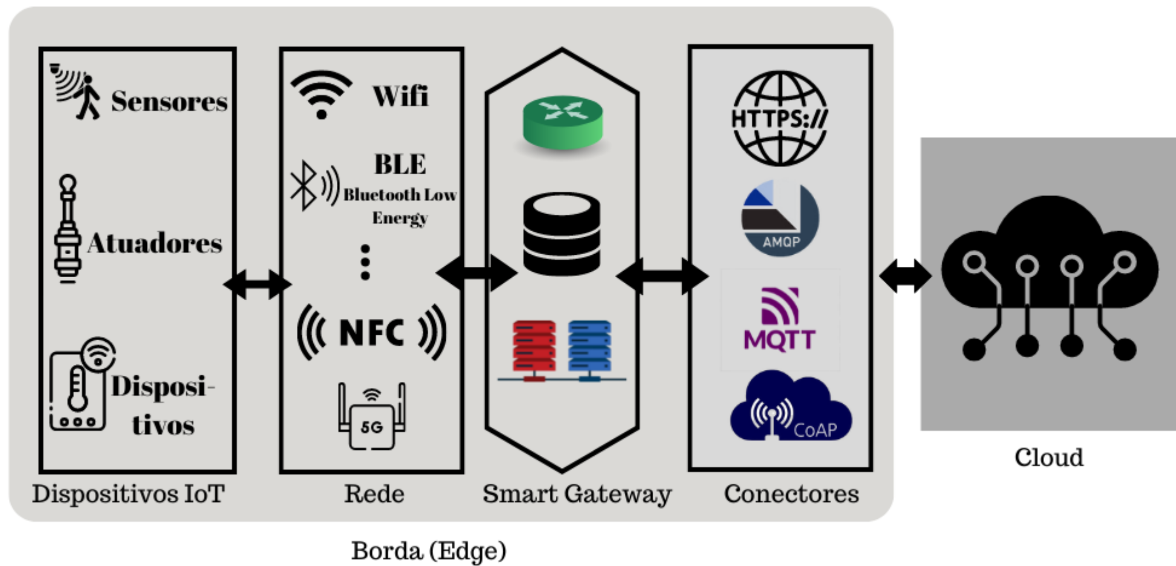

Figura 2 – Processamento de borda.

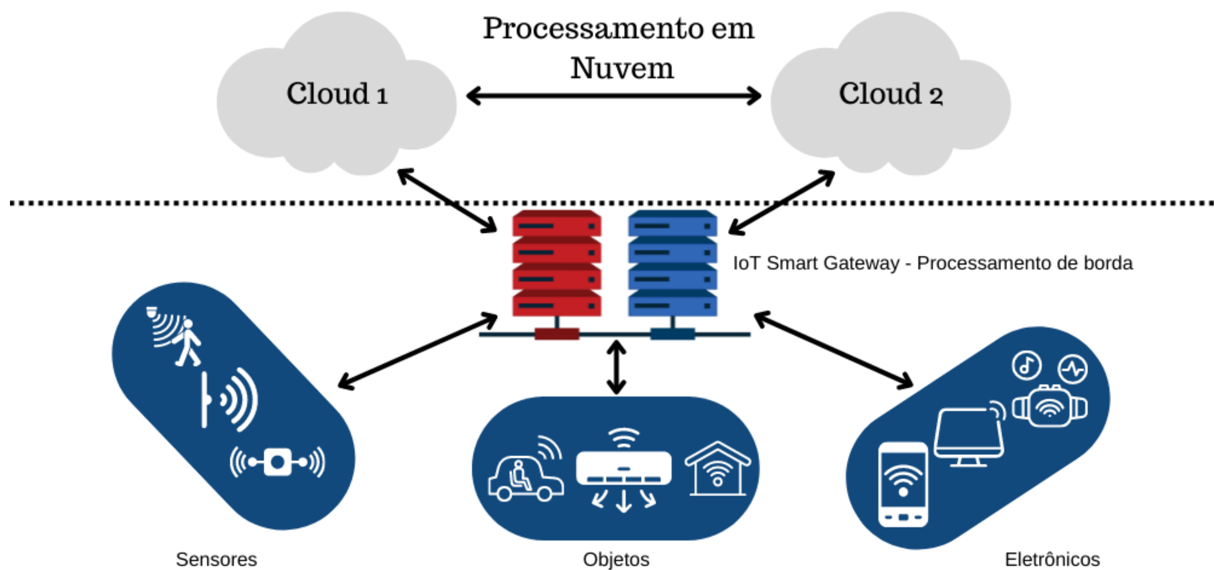

Figura 3 – Processamento de borda colaborativo.

O artigo de revisão dos autores Srinidhi, Kumar e Venugopal [31] tem um dos seus focos na otimização de rede, focado em maiores taxas de transmissão, recuperação de dados, eliminação de redundâncias de dados e melhora no tempo de resposta da aplicação e da rede. Especialmente em IoT, é citado o grande aumento no tráfego de dados e na

quantidade de dispositivos, que passa à grandeza de bilhões de dispositivos. O artigo menciona uma comparação entre as redes de celulares com as redes IoT. Dispositivos IoT geram menores quantidades de dados, porém as integrações feitas entre dispositivo IoT até suas aplicações atribuem um grande volume de tráfego por conta das mensagens de controle.

Com esse aumento de dados de controle, o artigo apresenta diversos tipos de algoritmos de endereçamento e de otimização de mensagens de controle, alguns deles são: PSO (*Particle Swarm Optimization* - otimização por enxame de partículas), algoritmos genéricos (GA), algoritmos heurísticos, algoritmos evolucionários (*Evolutionary Algorithms*), algoritmos baseados na lógica Fuzzy, algoritmos estocásticos e algoritmos miméticos (MA). Todos esses tipos de algoritmos foram estudados no artigo para resolução do problema de multi objetos e a busca pelo caminho mais curto (podendo ser o caminho com menor quantidade de saltos ou caminho com menor custo).

A prática mais utilizada em IoT é a computação em nuvem (cloud computing) [32], porém essa técnica resulta em um tráfego excessivo de informações entre os dispositivos e seus servidores, congestionando a rede. Existe uma técnica chamada computação em névoa (*fog computing*) apresentada no artigo [33], em que o processamento, armazenamento e os serviços de rede são alocados mais próximo aos usuários ou dispositivos, com isso tem-se uma plataforma virtualizada e distribuída. O artigo [34] propõe uma abordagem híbrida entre computação em nuvem e computação em névoa, para reduzir o tráfego da rede com a computação em névoa e levar o processamento do dispositivo com a computação em nuvem.

Assim, os dois primeiros trabalhos apresentados [25, 26] propõem novos protocolos específicos para IoT, o primeiro [25] diminui os dados de cabeçalho ao se comparar ao protocolo HTTP, o segundo [26] trabalha com agregação de pacotes. Esses necessitam de uma implementação tanto do dispositivo quanto do servidor. O terceiro [27] e o quarto [28] trabalhos utilizam o conceito de processamento de borda. O [27] realiza um pré-processamento no dispositivo antes de enviar os dados para o servidor. Nesse caso, teria um consumo maior de *hardware* do dispositivo e o mesmo teria que estar apto à esse processamento. No caso do artigo [28] é utilizado o processamento de borda porém o processamento é feito no *Gateway*, para reduzir o tráfego de dados da rede e retirar a carga de processamento ao dispositivo, que em IoT, tem energia limitada. No artigo de revisão [31] são apresentados vários algoritmos de melhor endereçamento (melhor caminho) e de otimização de mensagens de controle, para reduzir o tamanho do pacote em si e o percurso necessário para o pacote chegar ao destino. Por fim, nos artigos [33] e [34] são apresentados técnicas de computação em névoa para reduzir o tráfego de rede entre o dispositivo e o servidor.

1.2 Objetivo

O objetivo desse trabalho é aprimorar o envio de informações em uma rede IoT, trabalhando dinamicamente com os limites mínimos e máximos configuráveis em um novo modelo de serviço dinâmico. Com os limites configurados e o servidor ativo para interpretar os dados recebidos, se poderá atribuir uma nova configuração de intervalo de envio e de políticas de desastre e recuperação ao dispositivo individual.

1.3 Organização do trabalho

O trabalho está organizado em cinco itens principais: Capítulo 1 apresenta a introdução com o tema global, o problema a ser tratado e o objetivo do trabalho; Capítulo 2 explica as tecnologias e paradigmas utilizados para desenvolver o serviço dinâmico; Capítulo 3 é o desenvolvimento do serviço dinâmico com o código e a lógica utilizada; Capítulo 4 é um conjunto de simulações feitas sem e com a otimização do serviço; Capítulo 5 é um conjunto de experimentações realizadas utilizando o dispositivo Nvidia Jetson Nano; Capítulo 6 é a conclusão do trabalho, com a validação do objetivo e algumas ideias de continuação e validações do trabalho realizado. Nos apêndices estão os códigos na íntegra do aplicativo Android, do servidor dinâmico em Python e do painel de informações em Node-RED.

2 Ferramentas e tecnologias

Nesse Capítulo serão descritos os conceitos e tecnologias utilizada para desenvolver e simular o servidor inteligente, desde os serviços da Azure para conectar os dispositivos ao servidor, até as linguagens utilizadas para desenvolver o servidor, o dispositivo e o painel de monitoramento (do inglês, *dashboard*).

2.1 Gêmeo Digital

O gêmeo digital (*digital twin*) é uma representação digital abrangente de um dispositivo individual. Nessa representação estão contidas as propriedades, condições e comportamentos do dispositivo real por meio de modelos e dados [35].

Na literatura, pode-se encontrar outras definições para o gêmeo digital:

- Um Gêmeo Digital é uma simulação multi física, multiescala e probabilística integrada de um veículo ou sistema construído que usa os melhores modelos físicos disponíveis, atualizações de sensores para espelhar a vida de seu gêmeo real correspondente [36].
- Um gêmeo digital é um modelo computadorizado de um dispositivo ou sistema físico que representa todos os recursos funcionais e links com os elementos de trabalho [37].
- O gêmeo digital é na verdade um modelo virtual do modelo real ou sistema físico, que se adapta continuamente às mudanças operacionais com base nos dados e informações *online* coletados e pode prever o futuro da contraparte física correspondente [38].
- Um Gêmeo Digital é um conjunto de informações virtuais que descrevem completamente uma produção física potencial ou real desde o nível micro atômico até o nível macro geométrico [39].
- Um gêmeo digital é uma representação digital de um item físico ou montagem com simulações integradas e dados de serviço. A representação digital contém informações de várias fontes ao longo do ciclo de vida do produto. Essas informações são continuamente atualizadas e visualizadas de várias maneiras para prever condições atuais e futuras, tanto em ambientes de projeto quanto operacionais, para aprimorar a tomada de decisões [40].

2.2 Linguagem JavaScript - JSON

JSON (JavaScript Object Notation) é uma formatação de dados baseada nos tipos da linguagem JavaScript de programação [41]. O formato JSON pode representar quatro tipos primitivos: *strings* (sequencia de caracteres Unicode), números, booleano (verdadeiro ou falso) e *null* (vazio). [42] O JSON é constituído por um nome, ou chave, e seu valor. O exemplo abaixo demonstra algumas chaves e seus valores:

```
{
    "universidade": "UNIFEI",
    "cidade": "Itajubá",
    "estado": "MG"
}
```

Além dos quatro tipos primitivos, o JSON pode conter objetos e listas. Abaixo estão dois exemplos, o primeiro de objeto e o segundo de lista (*Array*) [43].

```
{
    "endereço":
    {
        "linha1": "Av. B P S 1303",
        "cidade": "Itajubá",
        "estado": "MG",
        "cep": "37500-903",
        "país": "Brasil"
    }
}

{
    "estado": "MG",
    "cidades": [ "Varginha" , "Itajubá" , "Pouso Alegre" ]
}
```

JSON desempenha um papel crucial em aplicações *cloud*. Softwares que executam funções ordenadas por máquinas remotas devem estabelecer um protocolo preciso para receber e responder a solicitações, normalmente chamadas de interface de programação de aplicativo (API - Interface de Programação de Aplicação). Dado que o JSON é uma linguagem que pode ser facilmente entendida por desenvolvedores e máquinas, tornou-se o formato mais popular para enviar e receber solicitações de API sobre o protocolo HTTP [44].

Como exemplo, um aplicativo contendo informações sobre o condições climáticas em todo o mundo. O aplicativo fornece uma API para permitir que outros sistemas acessem essas informações. Uma chamada hipotética para esta API pode ser uma solicitação contendo este arquivo JSON:

```
{
    "Country": "Brazil",
    "City": "Itajubá"
}
```

O sistema recebe um pedidop das condições climáticas atuais da cidade de Itajubá , Brazil. A API responderia com um resposta HTTP contendo o seguinte arquivo JSON:

```
{
    "timestamp": "14/01/2022 11:59:07",
    "temperature": 28,
    "Country": "Brazil",
    "City": "Itajubá",
    "description": "Sunny"
}
```

Indica que a temperatura é 28 graus e o dia está ensolarado. Este exemplo ilustra a simplicidade e legibilidade do JSON, que explica parcialmente sua rápida adoção.

2.3 Azure

Azure é uma plataforma de serviços de computação em nuvem da Microsoft, com uma ampla e crescente gama de serviços que geralmente constituem elementos básicos da computação em nuvem [45].

Computação em nuvem é uma alternativa moderna ao tradicional *data center*. Um vendedor de serviços em nuvem é responsável pela aquisição e manutenção de todo *hardware*. Geralmente, é disponibilizado um portal de fácil acesso em que o usuário consegue criar maquinas virtuais com características especificas de CPU, RAM, disco, sistema operacional, aplicações pré instaladas e localização geográfica do *data center*. O usuário consegue criar uma máquina virtual que atenda suas necessidades e pode acessá-la em alguns minutos [46].

Neste trabalho são utilizados os serviços em nuvem da Azure para criar, vincular e sincronizar a rede IoT. É criado um *link* de acesso para configurar os dispositivos IoT

para que sejam registrados na rede, com isso terão direitos de enviar e receber telemetrias e sincronizar o dispositivo gêmeo vinculado ao dispositivo real.

O Hub IoT é o serviço em nuvem da Azure especifico para redes IoT, em que disponibiliza o *link* de acesso e o conceito de dispositivo gêmeo a ser usado para configurar o dispositivo real com base na linguagem JSON.

2.3.1 Hub Iot

O serviço em nuvem Azure IoT Hub é utilizado para conectar os dispositivos IoT ao Microsoft Azure. Os Hubs IoT são capazes de processar bilhões de eventos por dia e suportar integrações com os outros serviços do Microsoft Azure. [47]

O Hub suporta protocolos de enfileiramento e transmissão de dados HTTPS (*Hyper Text Transfer Protocol Secure* - Protocolo de transferência de hipertexto seguro) [48] , AMQP (*Advanced Message Queuing Protocol* - Protocolo avançado de enfileiramento de mensagens) [49] , MQTT (*Message Queuing Telemetry* Transport - Transporte de filas de mensagem de telemetria) [50] , AMQP por *WebSocket* (Protocolo *WebSocket* habilita uma comunicação de mão dupla entre um dispositivo não seguro em um ambiente controlado e um *host* remoto) [51] e MQTT por *WebSocket*.

Neste trabalho, o Hub IoT é utilizado como ponto de configuração para a aplicação Android, ou seja, o envio e recebimento de dados terá o Hub IoT como ponto de acesso, através de um *link* disponibilizado pela plataforma Azure e com o protocolo AMQP.

2.3.2 Dispositivo Gêmeo

O Dispositivo Gêmeo (*Device Twin*) é uma implementação do conceito de Gêmeo Digital para dispositivos IoT conectados ao Hub IoT desenvolvido pela Microsoft na plataforma Azure. É um documento de manutenção JSON (Formato de data popular para envio de requisições e respostas em API - Interface de programação de aplicações) [44] que contêm metadados específicos, configurações e condições individuais dos dispositivos. [47]

As informações armazenadas no gêmeo são propriedades para serem enviadas ao dispositivo (*desired*) e propriedades a serem coletadas pelo dispositivo (*reported*). [47]

Os dispositivos gêmeos permitem interações de serviço por meio de APIs do Hub IoT. A partir dessas APIs, os metadados de um documento gêmeo podem ser consultados, o que permite cenários de relatórios em painéis ou monitoramento de trabalhos de longa duração. Com gêmeos, as consultas podem acontecer em milissegundos, com o entendimento de que o conjunto de resultados é o último estado do dispositivo relatado, não uma atualização em tempo real do dispositivo.[47]

O conteúdo do documento JSON do gêmeo é composto por 4 partes:

1. Informações de identificação: propriedades de identificação, como o ID do dispositivo e a impressão digital X509 que são informações de leitura somente e que são criadas pelo Hub IoT quando o dispositivo virtual foi inicialmente criado.
2. Tags: São metadados criados pelos serviços e soluções de *back-end* para categorizar e classificar os dispositivos. Exemplos: localização e periféricos (sensor de temperatura, sensor de umidade, sensor de GPS(Sistema de posicionamento global), sensor de poluição).
3. Propriedades desejadas: São propriedades criadas e atualizadas pelos serviços e soluções de *back-end*, usadas para requisitar uma mudança de configurações que são enviadas ao dispositivo. A atualização o documento gêmeo não atualiza imediatamente o dispositivo físico. Simplesmente inicia-se a solicitação de alteração a ser enviada ao dispositivo do Hub IoT.
4. Propriedades reportadas: contêm o último estado conhecido do dispositivo para as informações especificadas na seção "propriedades desejadas". Estes são atualizáveis do dispositivo, mas somente leitura e consultável nos serviços de *back-end*.

A Figura 4 é uma representação visual da estrutura do dispositivo gêmeo:

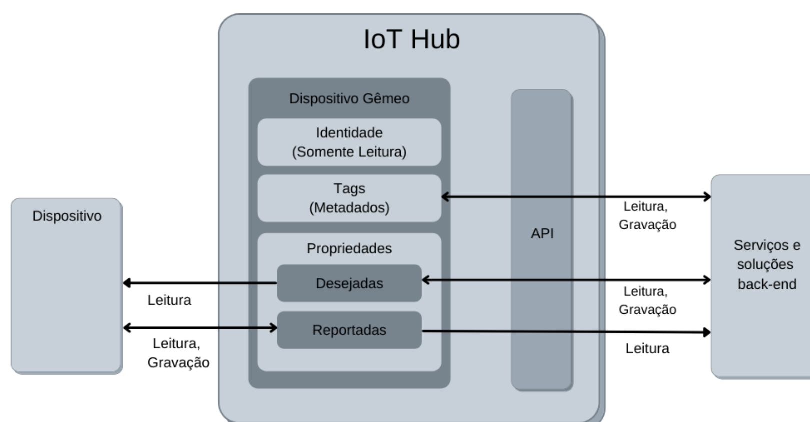

Figura 4 – Estrutura do dispositivo gêmeo.

Conforme mostrado na Figura 4, os serviços de *back-end* podem interagir com o dispositivo gêmeo por meio de API fornecida pelo Hub IoT. Por meio dessa API, os serviços podem ler e atualizar as *tags* do dispositivo, ler e atualizar as propriedades desejadas, além de ler e consultar as propriedades relatadas. Além disso, o dispositivo pode ler (e ser notificado de alterações) nas propriedades desejadas e ler e atualizar as propriedades reportadas. Abaixo está um documento JSON gêmeo com dispositivo de amostra:

```
{"deviceId": "device-2222",
"etag": "AAAAAAAAAAc=",
"status": "enabled",
"statusReason": "provisioned",
"statusUpdateTime": "0001-01-01T00:00:00",
"connectionState": "connected",
"lastActivityTime": "2015-02-30T16:24:48.789Z",
"cloudToDeviceMessageCount": 0,
"authenticationType": "sas",
"x509Thumbprint": {
"primaryThumbprint": null,
"secondaryThumbprint": null},
"version": 2,
"tags": {
"$etag": "123",
"deploymentLocation": {
"building": "43",
"floor": "1"}},
"properties": {
"desired": {
"telemetryConfig": {
"sendFrequency": "5m"},
"$metadata" : {...},
"$version": 1},
"reported": {
"telemetryConfig": {
"sendFrequency": "5m",
"status": "success"}
"batteryLevel": 55,
"$metadata" : {...},
"$version": 4}}}
```


Existem 3 restrições impostas a qualquer documento de dispositivo gêmeo:

- Todas as propriedades desejadas e reportadas tem que ser um JSON válido;
- O JSON de todas as propriedades não podem exceder o máximo de 5 níveis de indentação;
- O documento gêmeo não pode exceder um total de 8 kB de tamanho.

Essas restrições são impostas pela plataforma Azure, caso o documento não atenda às restrições impostas, a plataforma rejeita a nova configuração do dispositivo gêmeo. Isso não causará um erro, porém o ciclo de reconfiguração do dispositivo será quebrado.

2.4 Python

Python é uma dentre muitas linguagens de programação. Assim com as línguas humanas, existem muitas linguagens de computador, como JAVA, PHP, LISP, NODE, C entre outras. Cada linguagem tem suas especificidades para atender melhor uma necessidade, por exemplo escrever programas facilmente portáteis é um ponto para JAVA e acessar banco de dados e juntá-los em páginas da *Internet* é a especialidade do PHP. [52]

Python é uma linguagem de software livre e gratuita, que começou a ser utilizada em 1991, desde então está entre as dez linguagens de programação mais populares. Muito usada por organizações de alta rotatividade, por ter uma ótima reputação de produtividade. [52]

Python pode ser encontrada em muitos ambientes de computação. Alguns exemplos:

- Dispositivos móveis;
- Dispositivos embarcados;
- Ambientes de nuvem, servidores gerenciados por terceiros;
- Tela de linha de comando ou tela de terminal;
- Interfaces gráficas de usuário, incluindo ambientes em *cloud*;
- Aplicações em *cloud*, no lado do cliente e do lado do servidor.

As aplicações desenvolvidas na linguagem Python variam de *script* únicos até aplicações de milhões de linhas. A linguagem pode ser vista em sites, sistemas administrativos, manipulações de dados, entre outros.A relativa concisão presente na linguagem possibilita o desenvolvimento do programa menor do que seu equivalente em uma linguagem estática. [53]

Python tem um bom desempenho na maioria das aplicações, mas pode não ter a velocidade necessária em alguns casos mais criticos. Se a aplicação executa na maior parte do tempo comandos de cálculos (o termo técnico é limitado à CPU - CPU Bound), um programa escrito nas linguagens C, C++ ou Java poderão ter uma desempenho acima do programa equivalente escrito em Python [52].

O Python tem 4 tipos básicos de dados [53]:

- Booleano que pode ter valor de verdadeiro e falso;
- Inteiro que representa um número, como 45 ou 20000;
- Número flutuante que é um numérico com ponto decimal;
- Texto que é uma sequencia de caracteres.

Tudo na linguagem Python é tratado como objeto, desde os tipos básicos de dados até estruturas grandes de dados, funções e programas. Isso tráz consistência ao desenvolvimento [53]. Segue um exemplo de um código Python que recebe um nome, escreve no terminal "Olá ,"e o nome ditado e aguarda para fechar e sua execução.

Código Python:

```
name=input("Qual o seu nome? ")

print("Olá, ", name)

input("\nPressione qualquer tecla para fechar")
```

Ao executar o código acima, é mostrado as seguintes saídas na linha de comando:

```
Qual o seu nome? Miguel
Olá,  Miguel

Pressione qualquer tecla para fechar
```

Neste trabalho a linguagem Python será utilizada para desenvolver o serviçco dinâmico. Que será capaz de conectar ao Hub IoT da Azure, processar o recebimento de telemetrias dos dispositivos, atualizar o documento do dispositivo gêmeo e publicar para que o dispositivo real sofra a alteração necessária.

2.5 Android

O Android é uma plataforma completa para tecnologia móvel, com um pacote com programas para celulares, com um sistema operacional, *middleware*, aplicativos e interface do usuário [54]. O sistema operacional do Android é baseado em Linux [55], possui interface gráfica, GPS, diversos aplicativos e conta com um ambiente de programação completo, com um editor inteligente de código com explicação dos métodos em tempo de escrita, o editor visual de *layout*, o emulador de dispositivos Android (*smartphone*, *tablet* e tv) e um analisador de performance do aplicativo em tempo real, chamado Android

Studio, lançado em 2013. Nesse ambiente, o desenvolvedor pode optar por desenvolver com a linguagem Java ou Kotlin.

O Android Studio é um plataforma gratuita e compatível com Mac, Windows e Linux, que utiliza o Java SE Development Kit (JDK). Dentro da plataforma, o desenvolvedor consegue utilizar o SDK *Manager* para instalar múltiplos Android SDK, pois cada versão do sistema tem um SDK individual. Na plataforma Android Studio, o desenvolvedor consegue debugar um código com um dispositivo físico Android ou criar um dispositivo virtual Android, chamados AVDs (*Android Virtual Devices*). Esses dispositivos virtuais podem ser criados com resoluções especificas, como: tamanho da tela, resolução da tela, densidade de pixels, orientação da tela, memória RAM disponível e qual versão do sistema operacional Android que utilizará [56].

Alguns recursos presentes no Android [55]:

- *Framework* de aplicação: permite a incorporação, a reutilização e a substituição de recursos de componentes;
- Máquina Virtual Dalvik: máquina virtual otimizada para dispositivos móveis;
- Navegador Web Integrado: navegador baseado na engine de código aberto Webkit;
- Gráficos Otimizados: gráficos e tratamento de imagens por meio de uma biblioteca 2D e gráficos 3D baseados na especificação Open GL ES 1.0 (aceleração por hardware é opcional);
- SQlite: gerenciador de banco de dados para armazenamento de dados estruturados;
- Suporte Multimídia: suporta formatos de som (MP3, AAC, AMR), vídeo (MPEG4 e H.264) e formatos de imagens (JPG, PNG e GIF) nativamente;
- Câmera, GPS, bússola e acelerômetro.

As APIs presentes no Android podem ser modificadas por meio de programação do seu conteúdo ou utilizadas, caso já atenda a funcionalidade, de forma nativa e sem necessidade de customizações. Algumas delas são:

- android.util: contém várias classes utilitárias (classes de containers e utilitários XML);
- android.os: contém serviçoes referentes ao sistema operacional, passagem de parâmetros e comunicação entre processos;
- android.database: contém as APIs para comunicação com o banco de dados SQLite;

- android.content: APIs de acesso a dados no dispositivo, como as aplicações instaladas e seus recursos;
- android.view: pacote que contém as principais funções e componentes de interface gráfica;
- android.widget: contém *widgets* prontos (botões, listas, grades, etc) para serem utilizados nas aplicações;
- android.app: APIs de alto nível referentes ao modelo da aplicação;
- android.provider: API que contém os padrões de provedores de conteúdos;
- android.telephony: API para interagir com funcionalidades de telefonia e telecomunicação;
- android.webkit: APIs para ambientes *cloud*, bem como um navegador embutido;
- android.maps: possui bibliotecas necessárias para trabalhar com mapas.

O Android é composto por um grupo essencial de elementos para que o sistema possa instanciar e executar a aplicação, são eles [56]:

- Atividade (*Activity*): é utilizado para desenvolver uma tela da aplicação, composta por várias *Views*;
- Services: são códigos desenvolvidos para rodarem em *background*, não sendo interrompidos quando troca de atividade pelo usuário;
- *Broadcast Receivers*: é utilizado quando um evento externo é recebido através de uma intenção. Não possui interface, mas podem gerar uma notificação através do *Notification Manager*;
- *Content Providers*: utilizado para compartilhar dados entre aplicações no mesmo dispositivo.

A Figura 5 é um diagrama do ciclo de vida de uma atividade (*activity*) no Android e os métodos que são utilizados na transição de cada estágio desse ciclo.

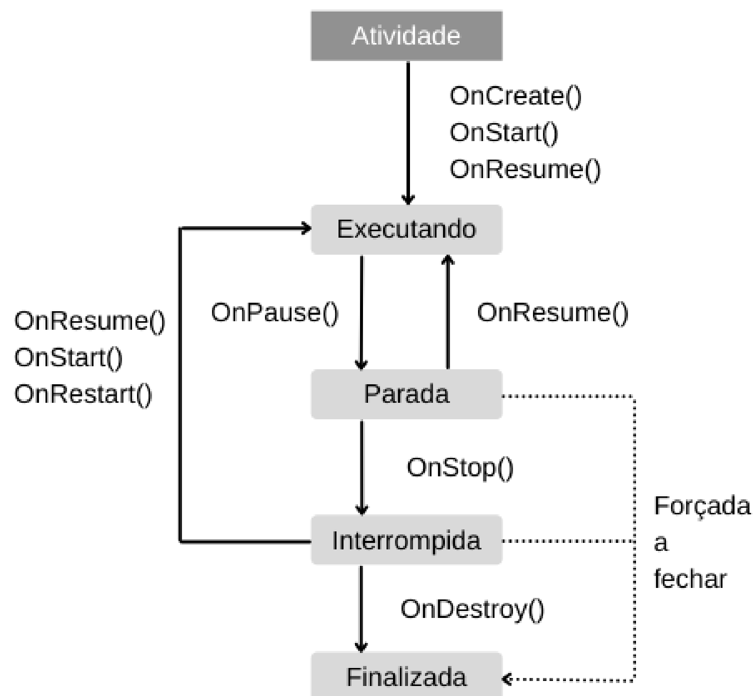

Figura 5 – Ciclo de vida de uma atividade.

2.6 Node-RED

Antes de explicar o Node-RED, é preciso entender o conceito de *Flow-Based Programming* (FBP), ou programação baseada em fluxo, entendido como uma linguagem de coordenação em vez de uma linguagem de programação. O FBP é constituído de componentes, que são blocos básicos utilizados pelos desenvolvedores para implementar uma aplicação. Desenvolver um novo componente só é necessário caso não exista um componente pronto que atenda aquela solicitação [57]. Normalmente as plataformas de desenvolvimento FBP possuem os seguintes itens [57]:

- Um conjunto de funções pré codificadas e pré testadas disponíveis em objetos, não em código fonte, com portas de entrada e saída;
- Um programa para orientar os módulos (componentes) que compõe aquela aplicação e implementa a interface de programação que os componentes usam para solicitar serviços;
- Um registro que determina as conexões entre os componentes para construir a aplicação e uma forma de transformar esses dados na estrutura de dados em que o driver interpreta. (Aplicações baseadas em FBP usam projetos visuais e são mais facilmente compreendidos dessa maneira)

- Os processos para converter, compilar e empacotar os módulos, as redes parciais e a rede global;
- Documentação
- Manual

Node-RED é uma das plataformas fundamentada em programação baseada em fluxo, desenvolvida pela IBM e pertencente ao OpenJS Foundation. No caso do Node-RED, os componentes do FBP são chamados de *node*. A implementação é definida no *node*, com isso o dado é entregue ao *node*, o mesmo realiza o processamento definido e passa o dado processado para o próximo *node*. A rede desempenha a função de entregar os dados para que fluam entre os nós [57].

A programação baseada em fluxo tem uma facilidade grande no desenvolvimento do modelo visual, proporcionando o acesso e o entendimento de vários níveis de desenvolvedores. Qualquer um é capaz de entender o fluxo se um problema for sub-dividido em várias etapas. O Node-RED não é somente uma plataforma de programação, mas também uma plataforma de execução baseada no Node.js [58].

O editor de fluxo presente no Node-RED é acessado via navegador, desenvolvido em Node.js, e é o núcleo da plataforma. No editor que é escolhido o node a ser usado, suas configurações, suas conexões e como irá reagir aos pacotes de dados. Após esses passos, o desenvolvedor pode implantar sua aplicação através do próprio editor.

O conjunto dos *nodes*, menu da Figura 6, que podem ser utilizados é chamado de paleta e essa pode ser expandida instalando novos nodes criados pela desenvolvedora ou pela comunidade de desenvolvedores. Com isso os desenvolvedores podem compartilhar nodes desenvolvidos e até mesmo fluxos completos, que é gerado pelo editor e baixado como um arquivo JSON.

A Figura 6 é um exemplo de uma implementação em Node-RED de um *webservice* que recebe dados de consulta, pesquisa em duas tabelas de um banco de dados que está presente no mesmo servidor e retorna o resultado das consultas no banco de dados via resposta HTTP.

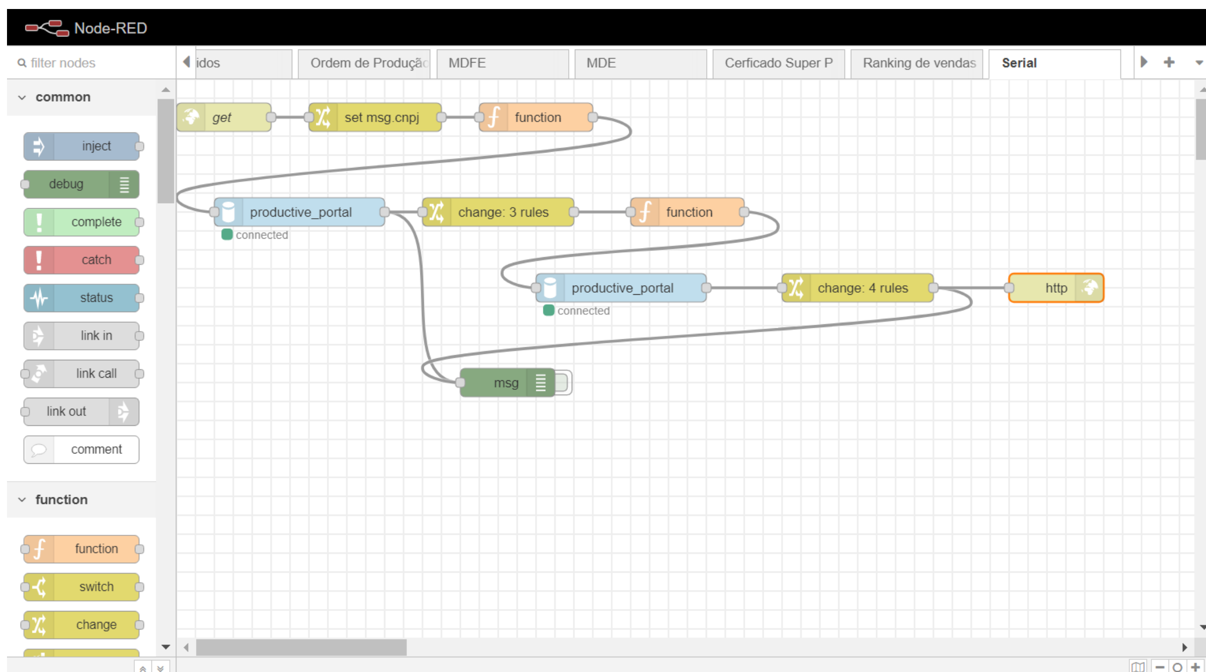

Figura 6 – Exemplo de código em fluxo desenvolvido em Node-RED.

2.7 Wireshark

Wireshark é um analisador de pacotes de rede. O analisador irá capturar os pacotes de rede e mostrar seus dados o mais detalhado possível [59]. O analisador Wireshark, originalmente chamado de Ethereal, se tornou uma ferramenta popular por ser uma solução de *software* de código aberto, o que o torna gratuito para todos os profissionais técnicos. Além disso, a ferramenta é capaz de se comunicar através de mais de 1000 protocolos de redes diferentes, possui uma equipe de desenvolvimento de mais de 800 desenvolvedores em todo mundo que mantém um suporte continuo e implementa novas funcionalidades [60]. Alguns usos mais comuns dos analisadores de rede: [61]

- Administradores de rede podem diagnosticar problemas da rede;
- Arquitetos de segurança conseguem auditar a segurança aplicada à um pacote da rede;
- Desenvolvedores de protocolos diagnosticam e aprendem com problemas relacionador à protocolos;
- *White-hat hackers*, ou hackers do bem, podem encontrar vulnerabilidades em aplicações e aprimorá-las antes dessas vulnerabilidades sejam utilizadas para roubo de dados.

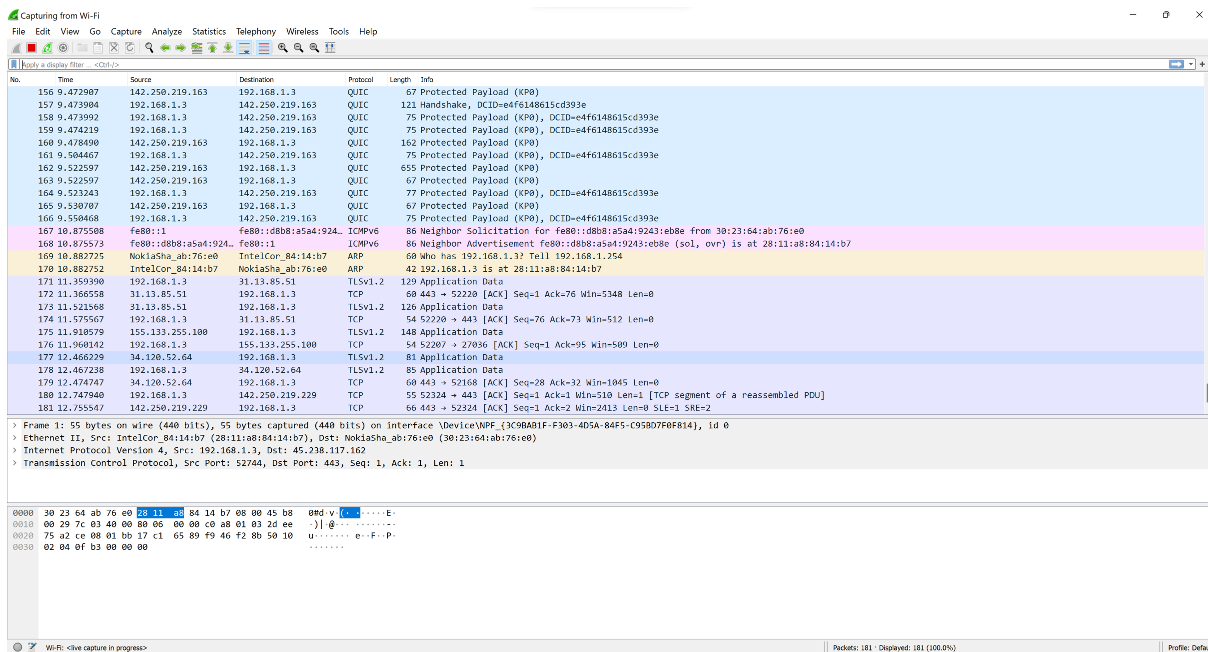

Figura 7 – Wireshark

Nesse trabalho o Wireshark foi utilizado para confirmar a quantidade de pacotes mostrados pelo código do dispositivo (Android). Outra informação utilizada no trabalho foi o tamanho dos pacotes, com intuito de demonstrar a diferença no volume de dados. Na figura 7 é um exemplo da tela do WireShark ao verificar pacotes na rede.

Para testar o comportamento do analisador de pacotes Wireshark foi feito um ping (um comando que envia um pacote simples para um nó remoto da rede [62]) com o analisador ativado. O comando ping enviou 50 pacotes e o analisador conseguiu verificar os 50 pacotes enviados. O comando ping e a saída do Wireshark estão presentes no Anexo D.

3 Serviço Dinâmico

O serviço dinâmico tem como objetivo configurar os dispositivos dinamicamente em produção, ou seja, quando o dispositivo enviar uma telemetria ao serviço, o mesmo pode reconfigurar a forma como o dispositivo se comporta (envio de mensagens), dependendo das informações recebidas pelo serviço. A Figura 8 mostra o fluxo macro do serviço.

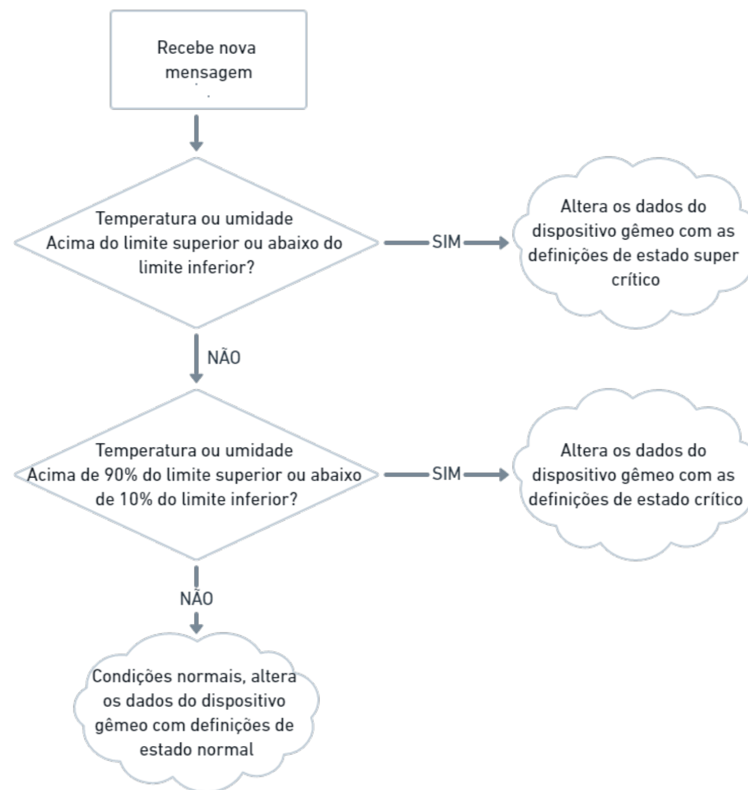

Figura 8 – Fluxograma do serviço - Recebendo uma nova mensagem.

O intuito dessa nova configuração é aprimorar a quantidade de informações, com o acréscimo da quantidade de envios caso o estado do dispositivo seja crítico, ou diminuir o envio de mensagens caso o estado do dispositivo esteja normal, priorizando baixo custo de dados e baixo consumo de energia do dispositivo.

Os algoritmos do serviço e do dispositivo teste utilizam o SDK disponibilizado pela Microsoft para utilizar a plataforma Azure. O projeto é feito com base em escalabilidade, para poder adicionar variáveis (além da temperatura e umidade) tanto no dispositivo quanto no serviço.

3.1 Dispositivo

O dispositivo foi simulado através de um aplicativo Android baseado no SDK e no *template* da Azure.Ele se conecta ao Hub IoT da Azure via protocolo MQTT (*Message Queuing Telemetry Transport*). O SDK utiliza um *String* de conexão com informações do nome da Hub IoT (*HostName*), a identificação do dispositivo (*Device Id*), e uma chave de acesso criada pela plataforma da Azure (*SharedAccessKey*).

```
String DeviceConnectionString =
"HostName=RedeHibridaIoT.azure-devices.net;
DeviceId=Teste1;
SharedAccessKey=QWGWWSZJ+gStxB7TLBqVyWyrv02yYadCRsMYizewDjE=";
```

Após fazer a conexão, é necessário incluir alguns métodos ao sistema, como: um retorno, caso a conexão tenha modificação, um retorno para as mensagens enviadas ao serviço, inscrever aos métodos que o serviço pode solicitar ao dispositivo, iniciar a sincronização do dispositivo gêmeo (*Device Twin*) e aplicar as políticas de desastre inicial.

Nesse trabalho, o algoritmo presente no serviço irá modificar dinamicamente, através do dispositivo gêmeo (*Device Twin*) o intervalo de envio de mensagens e os parâmetros da política de desastre. Essa modificação é feita quando o serviço recebe uma nova telemetria do dispositivo. O serviço valida a telemetria e verifica se as configurações atuais do dispositivo gêmeo podem ser otimizadas.

Quando o dispositivo gêmeo é atualizado, o dispositivo recebe um retorno de método do Azure Hub IoT, pois o mesmo foi inicializado no comando *startDeviceTwin* do objeto *client*. Esse retorno é o JSON do dispositivo gêmeo, composto pelo novo intervalo de mensagens e os novos parâmetros para a política de desastre.

O método que implementa a política de desastre é "*ExponentialBackoffWithJitter*", que recebe 5 parâmetros de configuração:

- *retryCount*: é o número máximo de tentativas alocadas na política;
- *minbackoff*: é o mínimo intervalo em millisegundos entre cada tentativa;
- *maxBackoff*: é o máximo intervalo em millisegundos entre cada tentativa;
- *deltaBackoff*: é o delta máximo em millisegundos entre tentativas;
- *firstFastRetry*: é um booleano que indica se a primeira tentativa deve ser imediata.

O código, em Python, abaixo retira os parâmetros do JSON e salva em suas determinadas variáveis:

```
JSONObject json = new JSONObject(propertyValue.toString());
sendMessagesInterval = json.getInt("interval");
retryCount = json.getInt("retrycount");
minBackOff = json.getInt("minbackoff");
maxBackOff = json.getInt("maxbackoff");
deltaBackOff = json.getInt("deltabackoff");
firstFastRetry = json.getBoolean("firstfastretry");
```

Com os parâmetros definidos, pode-se atualizar o intervalo de envio de mensagens e republicar a política de desastre:

```
RetryPolicy retryPolicy = new ExponentialBackoffWithJitter(
                        retryCount,
                        minBackOff,
                        maxBackOff,
                        deltaBackOff,
                        firstFastRetry);
client.setRetryPolicy(retryPolicy);
```

A política de desastre, ou retryPolicy, é um algoritmo presente no SDK (*Software Development Kit* - Kit de desenvolvimento de software) disponibilizado pela Microsoft para uso dos seus serviçoes Azure. Esse algoritmo implementa um recuo exponencial com estratégia de repetição quando uma telemetria que o dispositivo tenta enviar ao serviço não conclua com êxito.

No aplicativo Android utilizado nesse trabalho, três informações sobre o dispositivo foram escolhidas e enviadas ao serviço: temperatura, umidade e nível de bateria (poderiam ser outras informações). A temperatura e a umidade são números randômicos gerados no momento do envio, com foco em trabalhar vários cenários para o serviço reagir. A bateria é a informação que o sistema Android passa ao aplicativo. No caso deste trabalho, o serviço não utiliza a bateria para tomada de decisão, mas o código do aplicativo já está preparado para tal característica. Segue os códigos para as 3 informações:

```
temperature = 20.0 + Math.random() * 100;
humidity = 30.0 + Math.random() * 200;
BatteryManager bm = (BatteryManager)
getApplicationContext().getSystemService(BATTERY_SERVICE);
int batLevel =
bm.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPACITY);
```

Após ter as informações que serão enviadas, é necessário estruturar as informações em uma *String*, é criado a mensagem com a informação a ser enviada, a identificação da mensagem e um método de retorno.

```
String msgStr = "\"temperature\": " + (int) temperature +
    ", \"humidity\": " + (int) humidity +
    ", \"battery\": " + batLevel;
sendMessage = new Message(msgStr);
sendMessage.setMessageId
    (java.util.UUID.randomUUID().toString());
EventCallback eventCallback = new EventCallback();
client.sendEventAsync(sendMessage, eventCallback, msgSentCount);
```

O método de retorno (*eventCallback*) deriva do método de retorno de evento do Iot Hub (*IoTHubEventCallback*). Esse método retorna informações sobre a mensagem enviada, por exemplo, se o evento de envio da mensagem teve êxito ou falha. No dispositivo simulado é possível verificar se o envio teve êxito ou falha através das informações atualizadas presente na tela.

No *layout* do aplicativo Android que simula o dispositivo IoT é possível verificar algumas informações: quantidade de mensagens enviadas, quantidade de sucessos, quantidade de falhas, o intervalo atual de envio de mensagens em millisegundos, quantidade de mensagens recebidas do serviço, última leitura de temperatura em graus Celsius, última leitura da umidade em porcentagem, os dados da política de desastre atual, última mensagem enviada e última mensagem recebida.

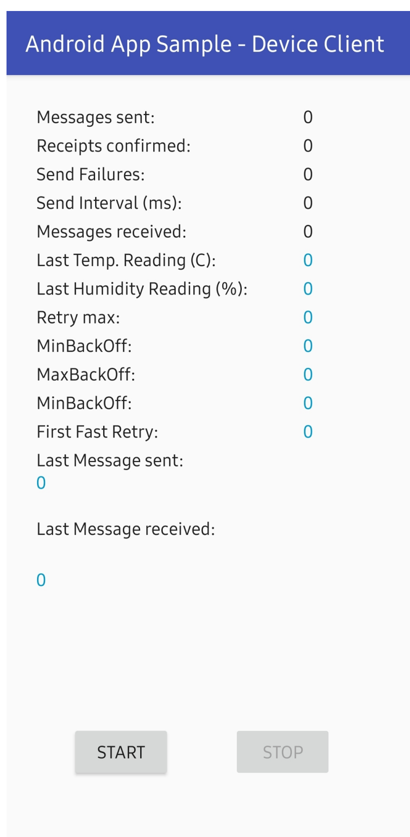

Figura 9 – Aplicativo Android.

A Figura 9 é composta por: Messages sent(número de mensagens enviadas), Receipts confirmed (número de mensagens confirmadas pelo serviço), Send Failures (número de falhas no envio), Send Interval (quantos millisegundos até o envio da próxima mensagem), Messages received (número de mensagens recebidas do serviço), Last Temp (última temperatura definida para ser enviada), Las humidity (última umidade definida a ser enviada), Retry max (quantidade de tentativas de re-envio apartir de uma falha), MinBackOff (intervalo mínimo para uma nova tentativa de envio), MaxBackOff (intervalo máximo para uma nova tentativa de envio), FirstFastRetry (tentativa imediata após uma falha), Last Message Sent (última mensagem enviada), Last Message Received (última mensagem recebida).

Ao clicar no botão "START", a aplicação faz a conexão com o Hub IoT e inicia a conexão com o Hub IoT e o envio das mensagens. Nesse momento os dados do intervalo de envio e os parâmetros da política de desastre são iniciadas com valores pré definidos no código. De início o intervalo de envio é configurado em 5000 millisegundos e os dados da política de desastre são: retryCount com 50000 tentativas, minBackOff com 100 milisegundos, maxBackOff com 100000 milisegundos, deltaBackOff com 100 mili segundos e firstFastRetry sendo falso. A figura 10 mostra o aplicativo Android.

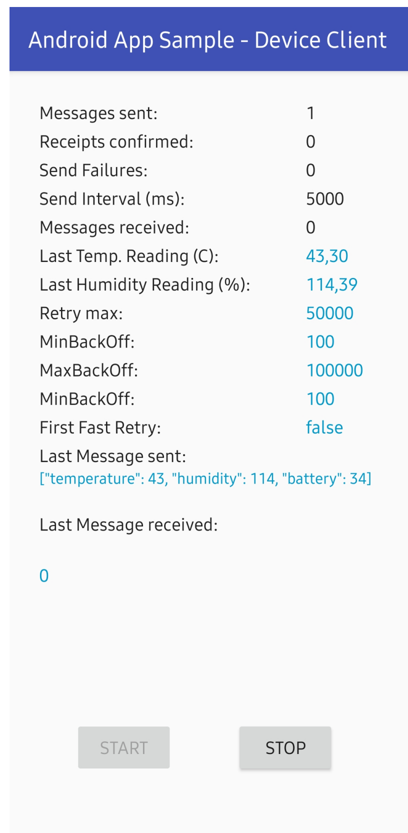

Figura 10 – Aplicativo Android ao iniciar.

A Figura 11 mostra a mudança das configurações de acordo com a telemetria enviada. Do lado esquerdo a mensagem que foi enviada continha uma temperatura em 49°C. Como no serviço a umidade máxima está definida em 70%, o serviço considerou a telemetria como super crítica, pois está acima dos parâmetros máximos. A partir disso o serviço alterou o dispositivo gêmeo com novos dados: o intervalo de envio passo de 5176ms para 4659ms e o MinBackOff da política de desastre passou de 63ms para 56ms. Os dados que não foram alterados é pelo fato do serviço ter limites mínimos e máximos, ou seja, a quantidade de tentativas de envio (retry max ou retryCount) já possuía o valor de 1000 e o máximo configurado no serviço em 1000, por isso a não alteração.

Figura 11 – Aplicativo Android - Comparação dos parâmetros.

3.2 Serviço dinâmico

O serviço desse trabalho foi desenvolvido em Python com a IDE (Ambiente de Desenvolvimento Integrado) Visual Studio Code desenvolvido pela Microsoft. Resumidamente, o serviço dinâmico não trata o que esta na mensagem, não armazena e não direciona para a aplicação. Esse serviço foi desenvolvido para otimizar as configurações dos dispositivos ligados ao Hub IoT da Azure, ou seja, ele recebe as mensagens que o Hub recebeu e modifica as configurações do dispositivo gêmeo referente àquele dispositivo.

O primeiro passo é conectar o serviço ao Hub IoT, para isso é necessário uma *String* de conexão e abrir a conexão via AMQP (Protocolo avançado de enfileiramento de mensagens).

```
CONNECTION_STR = "Endpoint=sb://ihsuprodcqres016dednamespace.service
bus.windows.net/;SharedAccessKeyName=iothubowner;
SharedAccessKey=pTOwcDX3j+NkkjyjsWsTNp3sG9iNYTh25Yjte2oxxHE=;
EntityPath=iothub-ehub-redehibrid-11558182-e1b8403e82"

client = EventHubConsumerClient.from_connection_string(
    conn_str=CONNECTION_STR,
```

```
        consumer_group="$default",
        transport_type=TransportType.AmqpOverWebsocket,
    )
```

Em seguida a conexão ao Hub IoT o algoritmo precisa registrar um método para receber os eventos e outro para erros.

```
    try:
        with client:
            client.receive_batch(
                on_event_batch=on_event_batch,
                on_error=on_error
            )
```

Com base no SDK, é no método de receber eventos (on_event_batch) aonde tem-se acesso a telemetria recebida do dispositivo e invoca o método de atualizar o dispositivo gêmeo. Os dados recebidos do Hub IoT são: identificação do dispositivo, temperatura, umidade e bateria. No trabalho não é utilizada a informação de bateria para tomada de decisão. A telemetria é recebida como uma *String*, um ajuste é feito para transformar a informação em JSON e assim obter cada dado individualmente para ser trabalhado.

```
# Define callbacks to process events
def on_event_batch(partition_context, events):
    for event in events:
        s = event.body_as_str()
        #add {} to read as json
        r = "{" + s
        r = r + "}"

        d = json.loads(r)

        temperature = d['temperature']
        humidity = d['humidity']
        battery = d['battery']

        device = event.properties[b'$.cdid']
        device =  str(device, 'UTF-8')

    partition_context.update_checkpoint()
    iothub_device_twin(device, temperature, humidity, battery)
```

Ao receber uma telemetria, o serviço exibe a mensagem no prompt de comando, conforme exemplo:

```
Nova mensagem:
Dispositivo:  {b'$.cdid': b'Teste1', b'temperatureAlert': b'true'}
Telemetria:  "temperature": 100, "humidity": 101, "battery": 67

Temperatura: 100
Umidade: 101
Bateria: 67
```

Depois da mensagem recebida e os dados extraídos da String da telemetria, é usado a identificação do dispositivo (ID) para buscar o arquivo JSON do dispositivo gêmeo referente a esse dispositivo real. Para acionar esse dispositivo gêmeo é preciso registrar um gerenciador IoT, com a *String* de conexão direta ao Hub IoT. O JSON do dispositivo gêmeo tem várias informações, as informações que são necessárias nesse momento estão contida no telemetryConfig dentro de desired, dentro de properties. Abaixo código Python do serviço dessa parte e um exemplo dessa parte especifica do JSON do dispositivo gêmeo.

```
def iothub_device_twin(device_id, temperature, humidity, battery):
    IOTHUB_CONNECTION_STRING = "HostName=RedeHibridaIoT.azure-devices
    .net;
    SharedAccessKeyName=iothubowner;
    SharedAccessKey=pTOwcDX3j+NkkjyjsWsTNp3sG9iNYTh25Yjte2oxxHE="

    iothub_registry_manager = IoTHubRegistryManager(IOTHUB_CONNECTION
    _STRING)

    twin = iothub_registry_manager.get_twin(device_id)
    twin_pro = twin.__getattribute__('properties')
    twin_desired = twin_pro.__getattribute__('desired')
    twin_config = twin_desired['telemetryConfig']
    interval = twin_config['interval']
    retrycount = twin_config['retrycount']
    minbackoff = twin_config['minbackoff']
    maxbackoff = twin_config['maxbackoff']
    deltabackoff = twin_config['deltabackoff']
    firstfastretry = twin_config['firstfastretry']

    {
```

```
"deviceId": "Teste1",
...
"properties": {
    "desired": {
        "telemetryConfig": {
            "interval": 4193.346392018642,
            "retrycount": 1000,
            "minbackoff": 51.04652363064001,
            "maxbackoff": 100000,
            "deltabackoff": 250,
            "firstfastretry": true
        },
        ...
```

Antes de trabalhar a lógica de como e quanto modificar os parâmetros, é necessário ajustar os dados recebidos, definir os limites mínimos e máximos da temperatura e umidade e estabelecer o incremento e decremento dependendo do estado do dispositivo. O estado do dispositivo poderá ser 3: normal, crítico e super crítico.

```
#Ajustando temperatura de graus para kelvin
temperature+=273

#Configuração do range de temperatura e de umidade
temperature_min = 10 +273
temperature_max = 80 +273
humidity_min=20
humidity_max=70

#Configuração do incremendo e decremendo dependendo do estado do
device
increment_normal = 0.5 #50%
increment_critic = 0.1 #10%
increment_super_critic = 0.2 #20%

status_normal = "normal"
status_critic = "critic"
status_super_critic = "super critic"
```

O estado super crítico é quando a temperatura ou a umidade estão menores que o

limite mínimo ou maiores que o limite máximo. Já o estado crítico é quando a temperatura ou umidade estão entre o limite mínimo e 10% acima do mínimo ou 10% abaixo do limite máximo.

```
status = status_normal
if(temperature<(temperature_min) or temperature>(temperature_max)):
        status = status_super_critic
        print("Temperatura super crítica")
    else:
        if(humidity<(humidity_min) or humidity>(humidity_max)):
            print("Umidade super crítica")
            status = status_super_critic
        else:
            if(temperature<(temperature_min*1.1) or
            temperature>(temperature_max*0.9)):
                status = status_critic
                print("Temperatura crítica")
            else:
                if(humidity<(humidity_min*1.1) or
                humidity>(humidity_max*0.9)):
                    print("Umidade crítica")
                    status = status_critic
```

Com o estado atual do dispositivo definido, as mudanças do intervalo de envio e dos parâmetros da política de desastre serão alterados. As alterações seguem o incremento e a lógica a seguir:

- Intervalo de envio: no estado super crítico e crítico o intervalo de envio é diminuído na porcentagem definida. No caso do teste desse trabalho, o estado super crítico diminui 20% e o crítico em 10%. Já no estado normal o intervalo de envio será aumentado em 50%. O intervalo tem limites mínimos e máximos que, caso ultrapassados, o valor do intervalo é definido como o valor do limite;
- O valor de *retrycount* (tentativas de re-envio) não segue o padrão de incremento dinâmico. É configurado no código um valor para o *retrycount* para cada estado do dispositivo (normal, crítico, super crítico);
- O dado de *minbackoff* segue o mesmo padrão do intervalo de envio. Nos estados criticos o valor é diminuído e no estado normal é aumentado. Além disso há uma verificação se esse dado chegou à um mínimo definido, caso sim, o valor será substituído pelo mínimo definido;

- O dado de *maxbackoff* segue um padrão inverso ao do *minbackoff*. Nos estados crítico o valor é aumentado na porcentagem definida do estado e diminuído caso esteja em um estado normal. Esse dado também tem a verificação, porém de valor máximo, ou seja, caso o novo valor calculado pelo incremento do estado for maior ao valor estipulado, o valor de *maxbackoff* será o valor estipulado;

- O valor de *deltabackoff* não segue o padrão de incremento dinâmico. É configurado no código um valor para o *deltabackoff* para cada estado do dispositivo (normal, crítico, super crítico);

- O dado de *firstfastretry* como é um dado de sim ou não (*true* ou *false*) tem 2 modos configurados: *true* para os estados críticos e *false* para o estado normal.

```
interval_min = 10
interval_max = 300000 #5 minutos

retrycount_normal = 10
retrycount_critic = 100
retrycount_super_critic = 1000

minbackoff_min = 10
minbackoff_max = 1000

maxbackoff_min = 1000
maxbackoff_max = 100000

deltabackoff_normal = 1000
deltabackoff_critic = 500
deltabackoff_super_critic = 250

firstfastretry_normal = False
firstfastretry_critic = True

if(status == "super critic"):
        interval*=(1-increment_super_critic)
        retrycount = retrycount_super_critic
        minbackoff*=(1-increment_super_critic)
        if(minbackoff<minbackoff_min):
            minbackoff=minbackoff_min
        maxbackoff*=(1+increment_super_critic)
```

```
            if(maxbackoff>maxbackoff_max):
                maxbackoff=maxbackoff_max
            deltabackoff=deltabackoff_super_critic
            firstfastretry=firstfastretry_critic
        else:
            if(status == "critic"):
                interval*=(1-increment_critic)
                retrycount = retrycount_critic
                minbackoff*=(1-increment_critic)
                if(minbackoff<minbackoff_min):
                    minbackoff=minbackoff_min
                maxbackoff*=(1+increment_critic)
                if(maxbackoff>maxbackoff_max):
                    maxbackoff=maxbackoff_max
                deltabackoff=deltabackoff_critic
                firstfastretry=firstfastretry_critic
            else:
                if(status == "normal"):
                    print("Normal")
                    interval*=(1+increment_normal)
                    retrycount = retrycount_super_critic
                    minbackoff*=(1+increment_normal)
                    if(minbackoff<minbackoff_min):
                        minbackoff=minbackoff_min
                    maxbackoff*=(1+increment_critic)
                    if(maxbackoff>maxbackoff_max):
                        maxbackoff=maxbackoff_max
                    deltabackoff=deltabackoff_normal
                    firstfastretry=firstfastretry_normal
```

Nesse momento os dados baixados do dispositivo gêmeo já foram atualizados de acordo com o estado atual do dispositivo, então é preciso publicá-los no no HuB IoT para que isso repercuta sobre o dispositivo em questão. Para atualizar o dispositivo gêmeo no Hub IoT é preciso criar a propriedade desejada e publica-la no serviço.

```
twin_properties = TwinProperties (desired= {
    'telemetryConfig' : {
        'interval' : interval,
        'retrycount' : retrycount,
        'minbackoff' : minbackoff,
```

```
            'maxbackoff': maxbackoff,
            'deltabackoff': deltabackoff,
            'firstfastretry': firstfastretry }})
    twin_patch = Twin(tags=new_tags, properties= twin_properties)
    twin = iothub_registry_manager.update_twin(device_id, twin_patch,
        twin.etag)
```

Nesse ponto, o dispositivo gêmeo já está com os novos valores e, assim que o dispositivo em questão estiver disponível, terá acesso aos novos dados do dispositivo gêmeo para se reconfigurar.

Abaixo o console do serviço e o que esse código imprime em tela. Existem 2 exemplos: um com temperatura super crítica e outro com a umidade super crítica. Nos dois casos, o intervalo de envio é reduzido, a quantidade de tentativas de re-envio é atribuído ao máximo (no caso 1000), o tempo mínimo de espera é reduzido, o tempo máximo de espera fica configurado ao limite máximo configurado, o delta referente a espera é colocado com 250 e a tentativa rápida é dado como verdadeiro.

```
Serviço iniciado.

Nova mensagem:
Dispositivo:  {b'$.cdid': b'Teste1', b'temperatureAlert': b'true'}
Telemetria:  "temperature": 100, "humidity": 101, "battery": 67

Temperatura: 100
Umidade: 101
Bateria: 67
Iniciando atualização do dispositivo gêmeo: Teste1

Configuração antes:
interval: 5752.189838160002
retrycount: 1000
minbackoff: 70.02266616000001
maxbackoff: 100000
deltabackoff: 250
firstfastretry: True

Iniciando as comparações:
Temperatura super crítica
```

```
Atualizou o dispositivo: Teste1
Configuração depois:
interval: 5176.970854344002
retrycount: 1000
minbackoff: 63.020399544000014
maxbackoff: 100000
deltabackoff: 250
firstfastretry: True


Nova mensagem:
Dispositivo:  {b'$.cdid': b'Teste1', b'temperatureAlert': b'true'}
Telemetria:  "temperature": 49, "humidity": 227, "battery": 67

Temperatura: 49
Umidade: 227
Bateria: 67
Iniciando atualização do dispositivo gêmeo: Teste1

Configuração antes:
interval: 5176.970854344002
retrycount: 1000
minbackoff: 63.020399544000014
maxbackoff: 100000
deltabackoff: 250
firstfastretry: True

Iniciando as comparações:
Umidade super crítica

Atualizou o dispositivo: Teste1
Configuração depois:
interval: 4659.273768909602
retrycount: 1000
minbackoff: 56.71835958960001
maxbackoff: 100000
deltabackoff: 250
firstfastretry: True
```

3.3 Painel de informações

Para uma visão mais gerencial do serviço, foi desenvolvido um painel em Node-RED com algumas informações: estado atual do serviço, última mensagem recebida, dispositivo referente à última mensagem recebida e a quantidade de mensagens recebidas pelo serviço dinâmico.

Esse painel recebe as informações do serviço Python via HTTP *post*, portanto o serviço e o painel podem estar no mesmo serviço ou em serviçoes diferentes.

Para desenvolver o painel foram utilizados *nodes* que já fazem parte da instalação padrão do Node-RED e *nodes* de painel presentes no palete *node-red-dashboard*, desenvolvido com a finalidade de criar painéis de visualização.

O painel é composto por quatro informações, três textuais (estado atual do serviço, última mensagem recebida, dispositivo referente à última mensagem recebida) e uma estilo medidor (quantidade de mensagens recebidas pelo serviço dinâmico). Todas essas informações utilizaram os *nodes* do palete *node-red-dashboard* e para que elas sejam exibidas de forma coerente, os nodes precisam receber as informações via msg.payload.

A Figura 12 mostra o fluxograma do Node-RED do painel:

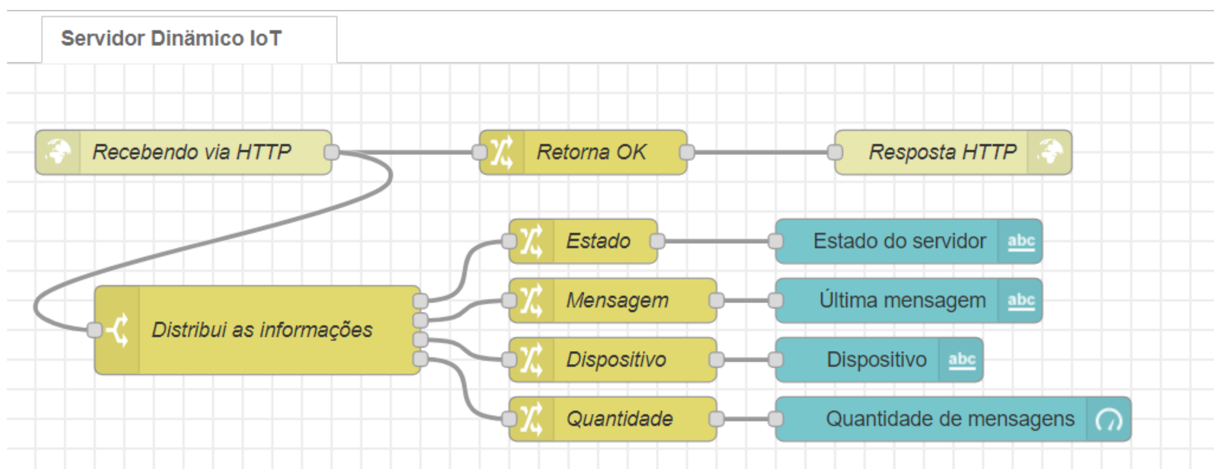

Figura 12 – Fluxograma do painel de visualização.

Ao receber uma requisição, o Node-RED irá distribuir as informações contidas na mensagem nos *nodes* de *dashboard* equivalentes.

Ao iniciar o serviço, o mesmo envia ao Node-RED que está em operação porém sem mensagens.

Figura 13 – Painel quando serviço é iniciado.

Nesse momento da Figura 13 o serviço está ativo, conectado ao Hub IoT da Azure e pronto para receber as mensagens e modificar os dispositivos gêmeos, caso necessário.

Quando o Node-RED recebe dados do serviço com todas as informações, ele popula os dados em seus devidos locais mostrado na figura 14:

Estado do servidor **ligado**

Última mensagem **"temperature": 21, "humidity": 43, "battery": 75**

Dispositivo **Teste1**

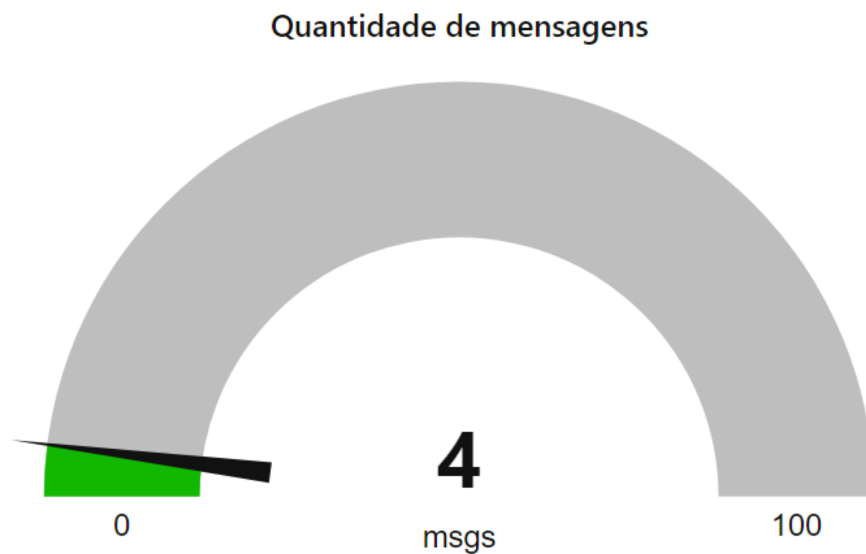

Figura 14 – Painel quando o serviço recebe mensagem.

Como esperado, quando o dispositivo IoT envia 4 mensagens ao serviço, o *dashboard* é atualizado, ao colocar o valor de 4 na quantidade de mensagens processadas e o nome do dispositivo e os dados da ultima mensagem recebida, visto na Figura 14.

Com a implementação feita, no Capítulo 4 são demonstrado simulações com os 3 estados do dispositivo para comparar os dados sem e com o serviço de otimização.

4 Simulação

Nesse Capítulo serão apresentadas simulações com um dispositivo IoT nas situações normal, crítica e super crítica com os resultados sem o serviço dinâmico e com o serviço dinâmico.

4.1 Configurações

Temperatura mínima	10°C
Temperatura máxima	80°C
Intervalo mínimo	1000ms
Intervalo máximo	300.000ms
Intervalo inicial	5.000ms
Decremento Crítico	10%
Decremento Super Crítico	20%
Incremento Normal	50%

Nas simulações apresentadas abaixo foram utilizados o intervalo mínimo de envio de telemetria como 2000ms e intervalo máximo de envio como 300000ms. Esses dados podem ser modificados nos campos *interval-min* e *interval-max* no serviço de acordo com os requisitos de cada aplicação.

A quantidade de tentativas de reenvio, caso a telemetria não seja enviada com sucesso, definida como *retrycount* nas políticas de desastre do Hub IoT, será atribuída de tal forma: estado normal 10, estado crítico 100 e estado super critico 1000. Desta forma, quanto maior a criticidade do estado do dispositivo a partir da sua temperatura, no caso dessa simulação, maior a quantidade de tentativas de reenvio.

Os dados de espera (*backoff*) mínimo e máximo são definidos da mesma forma que o intervalo de envio, com uma informação mínima e outra máxima. De acordo com o estado do dispositivo, o serviço ajusta através do incremento configurado no serviço.

Nessa simulação, os incrementos foram definidos com as porcentagens a seguir: estado normal com 50%, estado crítica com 10% e super critico com 20%. Dessa forma, quando o intervalo de envio esta em 5000ms e na última mensagem recebida de telemetria o serviço identificou que este dispositivo esta em seu estado normal, o intervalo de envio sofrerá um acréscimo de 50%, ou seja, passará de 5000ms para 7500ms.

Outra informação presente na política de desastre e que deve ser configurada pelo serviço é a primeira tentativa rápida de reenvio (*firstfastretry*). Nessa simulação será definida como positivo caso o estado do dispositivo for crítico ou super crítico e negativo caso o estado esteja normal.

Para essas simulações foram definidas a temperatura mínima como 10°C e a temperatura máxima 80°C. Por tanto foram feitas simulações com o dispositivo em estado normal, com 50°C, simulações em estado crítico, com 10°C e 75°C e em estado super crítico, com 0°C e com 95°C.

Os dados de umidade não serão modificados na simulação atual, com valor de 50% e o valor mínimo em 20% e máximo em 70%, para que o serviço identifique a umidade como normal.

Todas as simulações foram feitas com duração de dez minutos, com configuração inicial de intervalo de envio de 5000ms.

4.2 Dispositivo normal

As Figuras 15 e 16 são fotos da tela do dispositivo Android após os dez minutos de teste, uma com o serviço de otimização desligado e a outra com o serviço ligado, respectivamente. Para esse teste foi utilizada a temperatura em cinquenta graus, para que o serviço identificasse o dispositivo em estado normal.

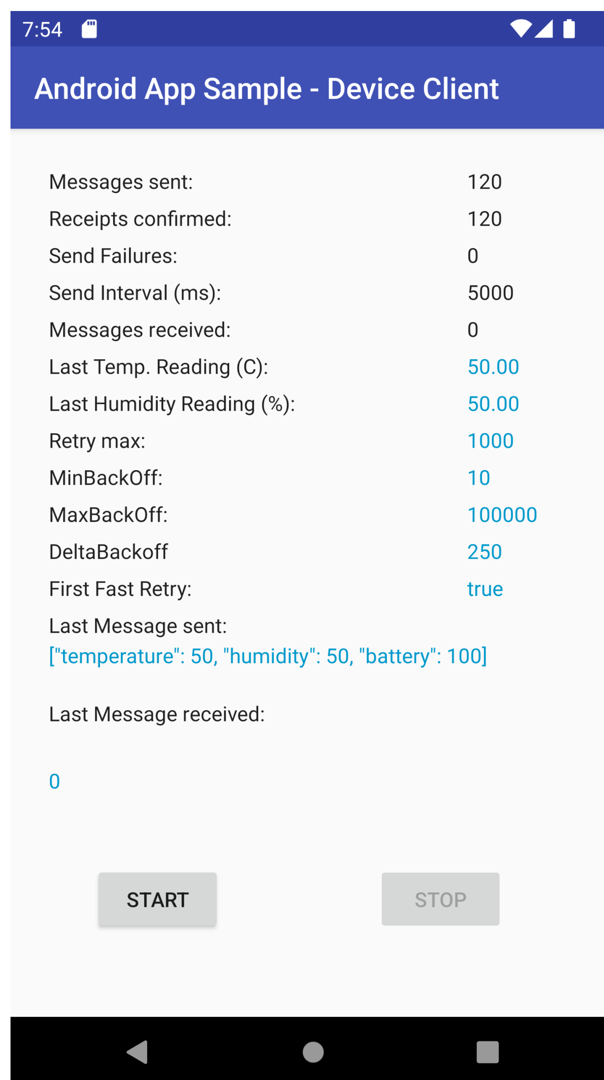

Figura 15 – Dispositivo Normal sem otimização.

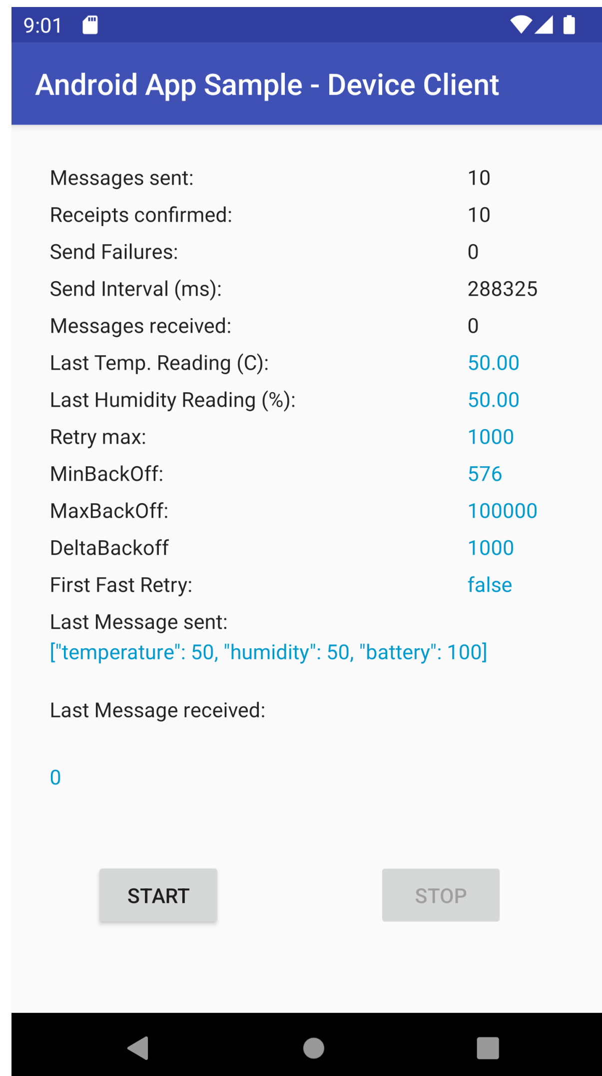

Figura 16 – Dispositivo Normal com otimização.

A partir das figuras é possível verificar que sem o serviço de otimização, a configuração de envio de 5000ms se manteve após os dez minutos de simulação e com o serviço, o intervalo foi modificado para valores maiores com intuito de economizar bateria do dispositivo IoT e minimizar os custos de envio de telemetria.

4.3 Dispositivo crítico

Com as mesmas premissas da simulação do dispositivo normal, nessa fase da simulação a temperatura foi testada no intervalo crítico durante dez minutos. As Figuras 17 e 18 mostram os resultados após os dez minutos com a temperatura em 75°C, assim o dispositivo está em estado crítico.

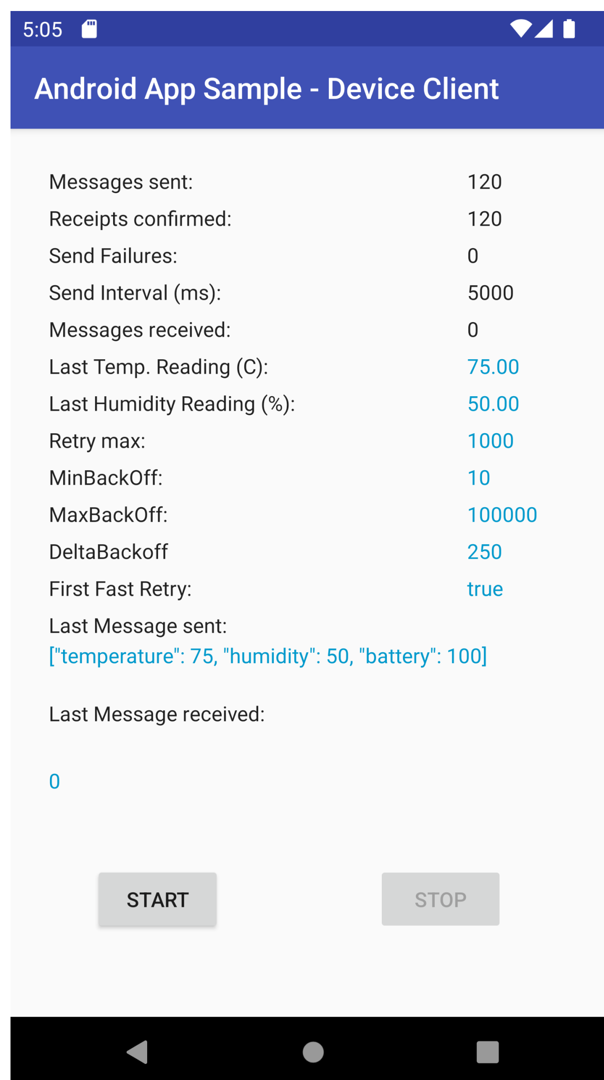

Figura 17 – Dispositivo Crítico com temperatura alta sem otimização.

Sem a otimização do serviço, mesmo o dispositivo em estado crítico, a quantidade de telemetria enviada não foi alterada. Isso muda quando o serviço de otimização entra em ação.

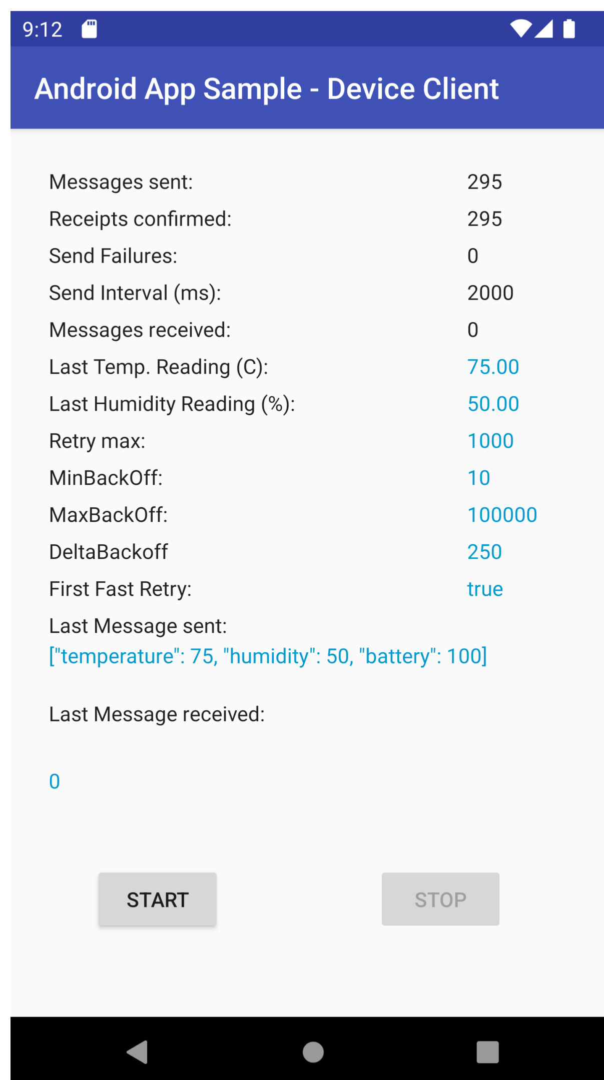

Figura 18 – Dispositivo Crítico com temperatura alta com otimização.

A partir da Figura 18 é possível verificar que ao ativar o serviço de otimização, como o estado do dispositivo é crítico, o serviço diminuiu o intervalo de envio de telemetria. Resulta, nesse caso, um aumento na quantidade de mensagens enviadas do dispositivo ao serviço. Isso ocorre com intuito de diminuir o tempo de resposta ao dispositivo crítico, seja ligar um ar condicionado se a temperatura estiver alta, ou ligar um aquecedor caso a temperatura estiver baixa.

O mesmo teste com o estado do dispositivo em crítico foi realizado com a temperatura baixa perante o normal. O resultado final está presente nas Figuras 19 e 20.

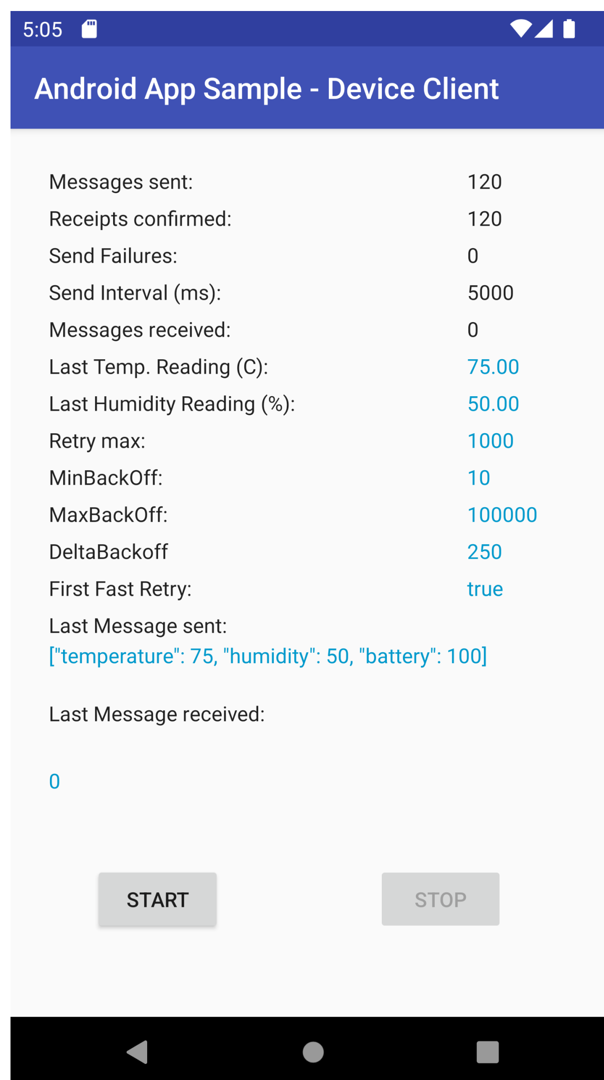

Figura 19 – Dispositivo Crítico com temperatura baixa sem otimização.

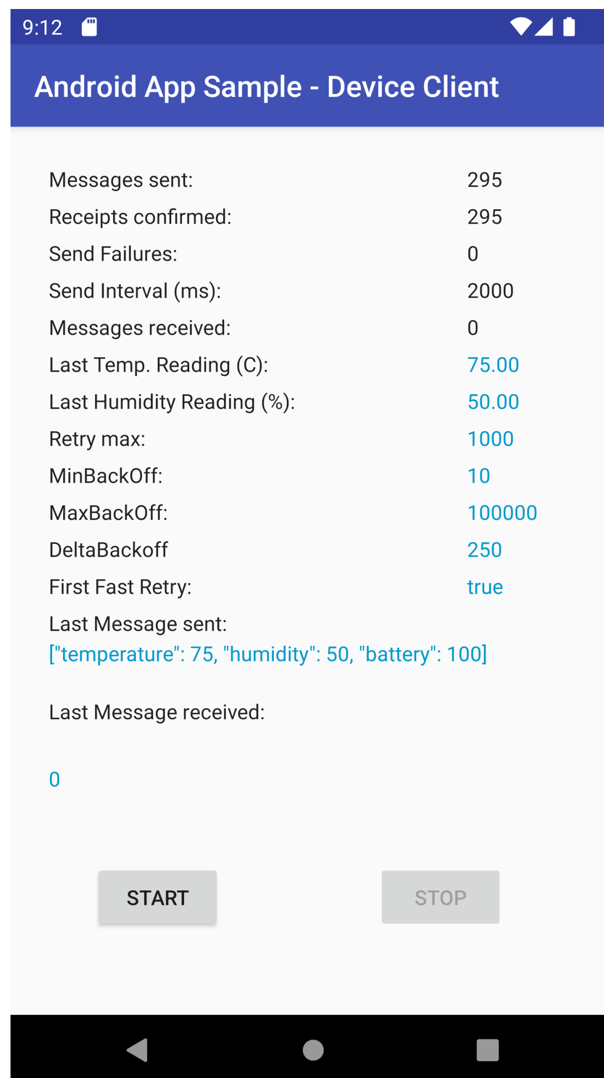

Figura 20 – Dispositivo Crítico com temperatura baixa com otimização.

O resultado foi o mesmo que em alta temperatura, o serviço diminuiu o intervalo de envio de mensagens, consequentemente a quantidade de mensagens no mesmo espaço de tempo foi maior.

4.4 Dispositivo Super Crítico

O estado de dispositivo super crítico é atingido quando a temperatura ou a umidade estão acima do máximo ou abaixo do mínimo configurados no serviço. A temperatura mínima configurada foi de 10°C e a máxima de 80°C. Para a simulação em dispositivo super crítico foi utilizada primeiramente a temperatura de 95°C para alta temperatura e zero graus Celsius para baixa temperatura.

As Figuras 21 e 22 são referentes ao dispositivo em estado super crítico com temperatura alta. A figura 21 sem otimização do serviço e a 22 com otimização do serviço.

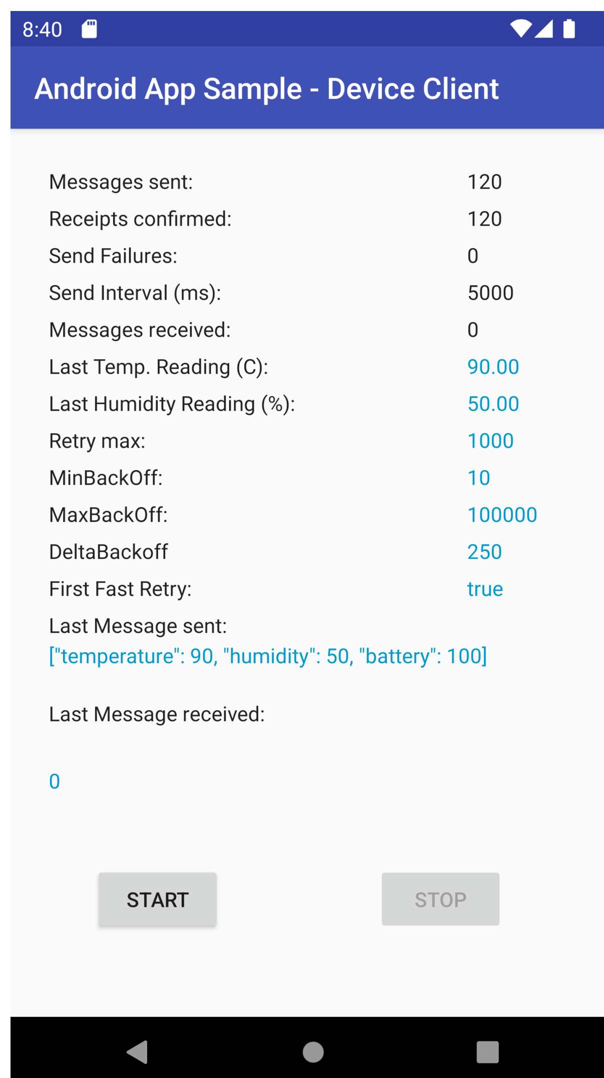

Figura 21 – Dispositivo Super Crítico com temperatura alta sem otimização.

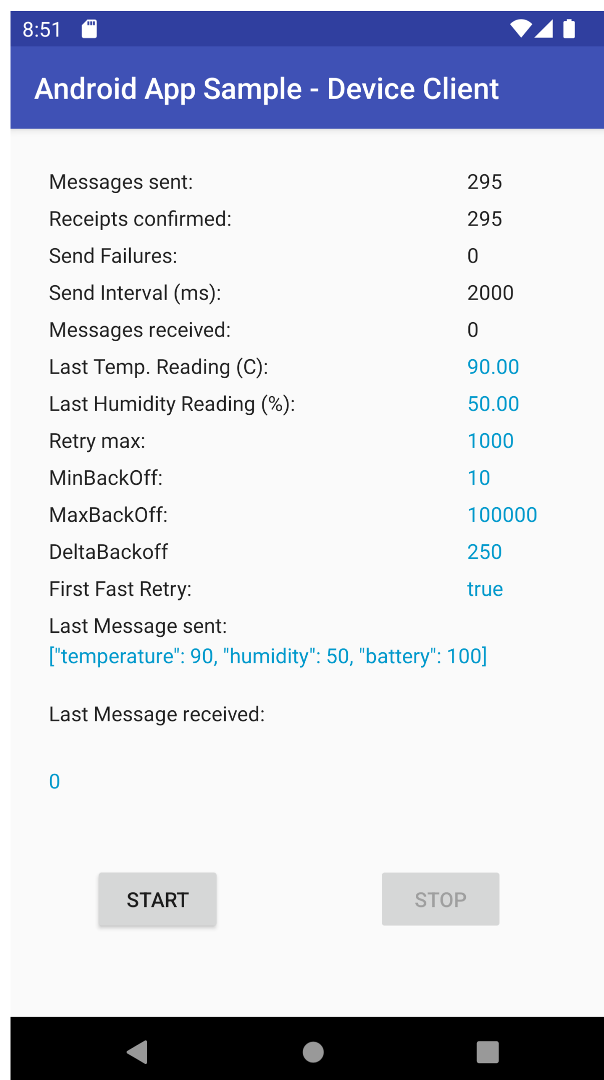

Figura 22 – Dispositivo Super Crítico com temperatura alta com otimização.

As Figuras 23 e 24 seguem referentes ao dispositivo em estado super crítico, porém com temperaturas abaixo da mínima configurada.

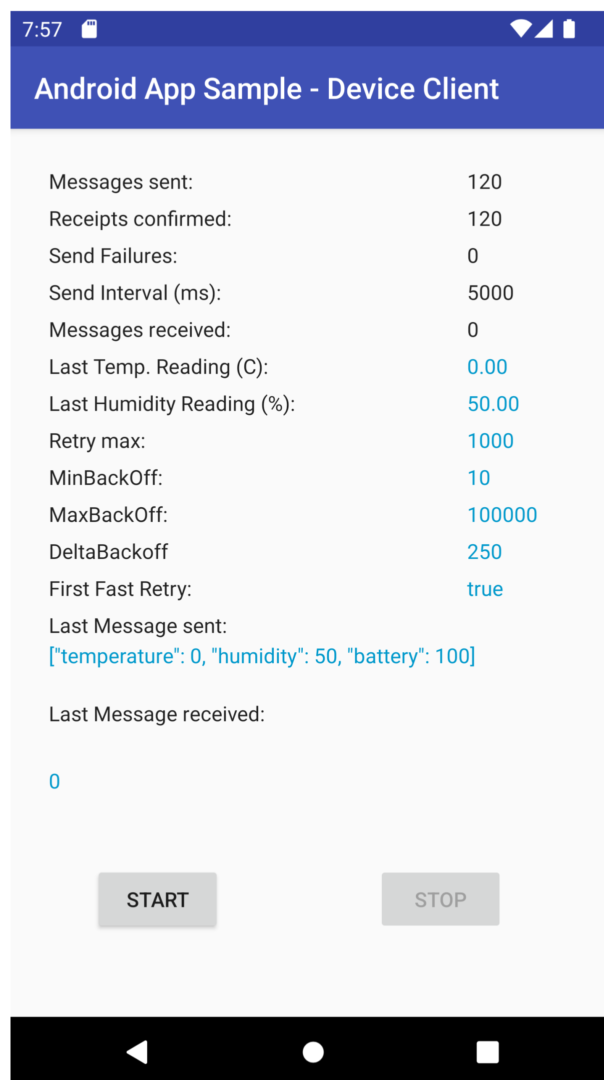

Figura 23 – Dispositivo Super Crítico com temperatura baixa sem otimização.

Na Figura 23, a quantidade de envio é padrão a todas as simulações feitas sem otimização, já que o intervalo de envio de telemetria fica constante em 5000ms, mesmo com o dispositivo em estado super crítico.

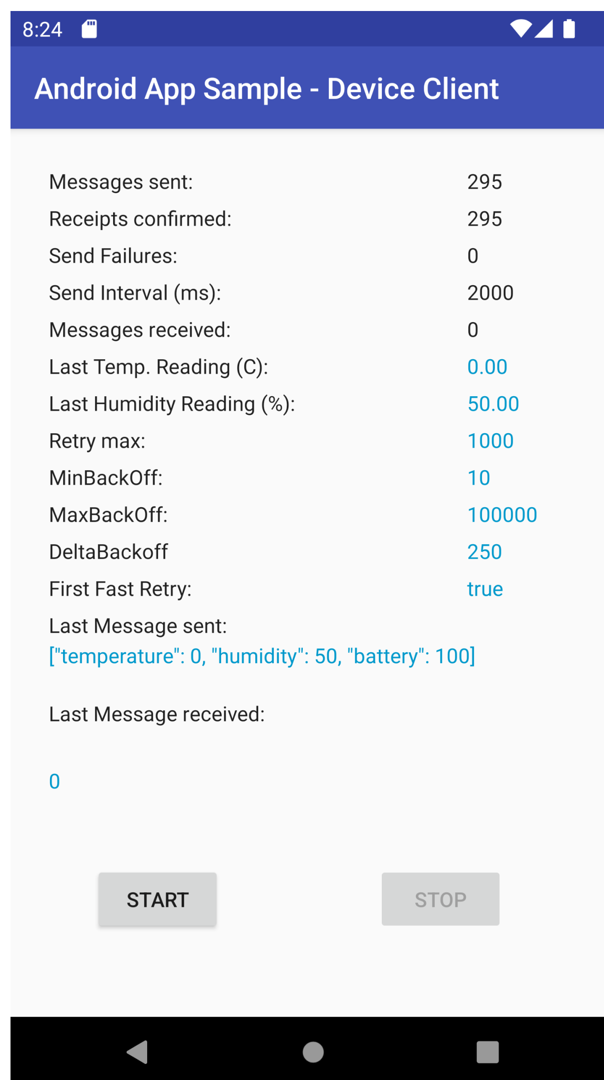

Figura 24 – Dispositivo Super Crítico com temperatura baixa com otimização.

Como pode-ser notar, quando o serviço de otimização está ativo e o dispositivo está em estado super crítico, o intervalo de envio diminui e consequentemente a quantidade de telemetria recebida no serviço aumenta.

4.5 Resultados

O resultado da otimização do serviço é diferente quando o dispositivo está em estado normal para quando o dispositivo está em estado crítico ou super crítico. A intenção do aprimoramento quando o dispositivo estiver normal é diminuir a quantidade de envio, para consumir menos bateria do dispositivo IoT e com menor gasto de envio de telemetria. A Figura 25 mostra exatamente essa diminuição na quantidade de envio de telemetria quando o dispositivo está em estado normal e o serviço de otimização está ativo.

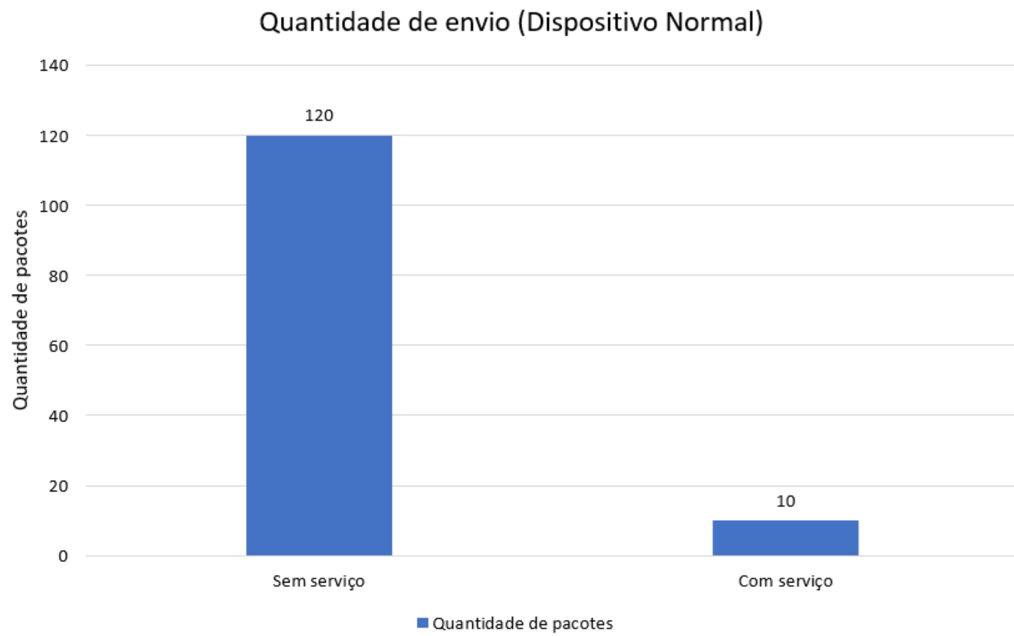

Figura 25 – Gráfico comparando otimização no dispositivo normal.

Além da quantidade de pacotes enviados, é possível verificar a mudança do intervalo de envio com o tempo em que o dispositivo está em estado normal, Figura 26, e a diferença no volume de dados trafegados pela rede na Figura 27.

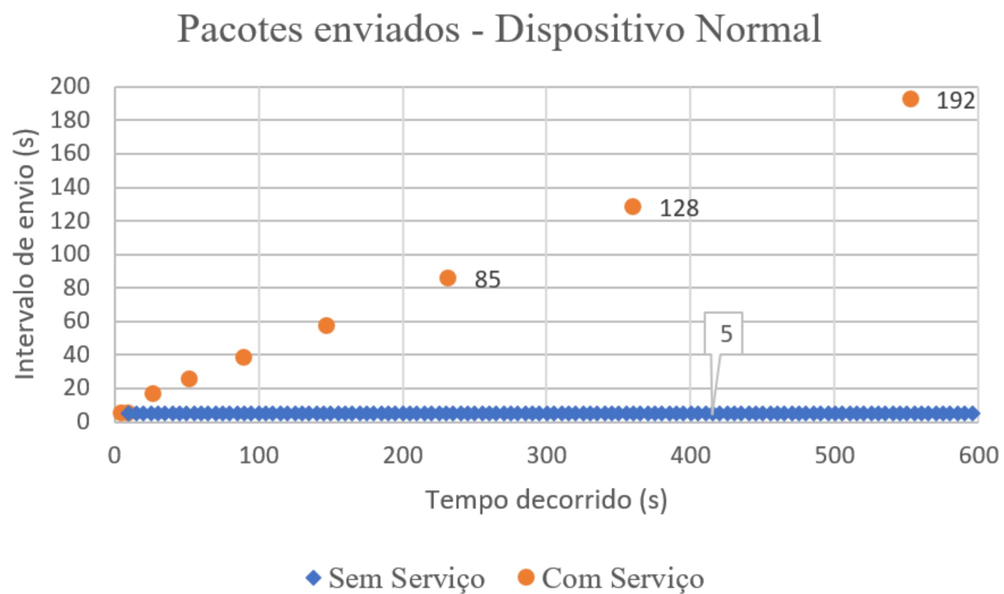

Figura 26 – Pacotes enviados - Dispositivo Normal.

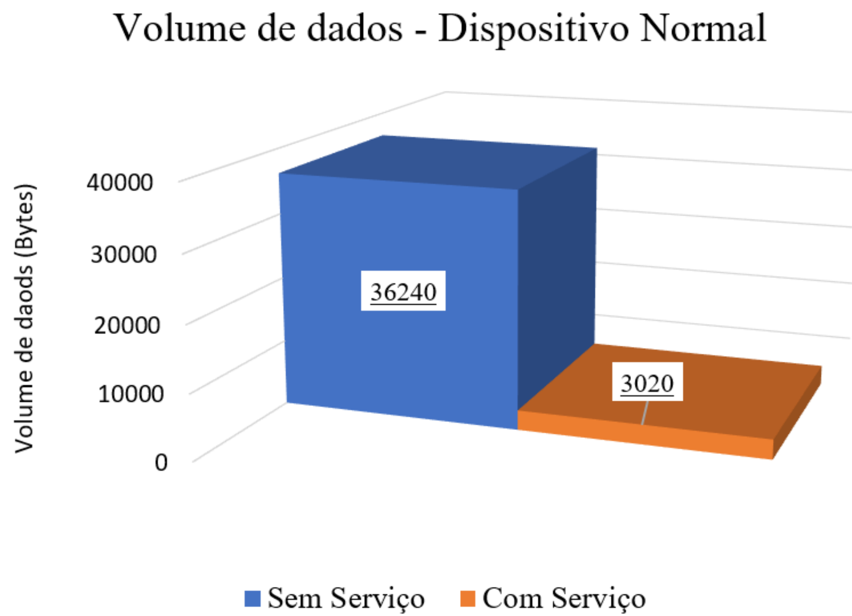

Figura 27 – Volume de dados - Dispositivo Normal.

As Figuras 26 e 27 mostram um aumento no intervalo de envio, causando diminuição da quantidade de pacotes e volume de dados trafegados pela rede, como era esperado com a otimização feita pelo serviço.

Em contra ponto, é possível verificar um aumento da quantidade de envio de telemetria quando o dispositivo está em estado crítico, seja em alta ou em baixa temperatura. Isso ocorre pois o serviço de otimização diminui o intervalo de envio de telemetria, já que o dispositivo não está em estado normal. Nas figuras 28 e 29 estão o resultado da simulação em que o dispositivo está em estado crítico, na figura 28 em baixa temperatura e na figura 29 em alta temperatura.

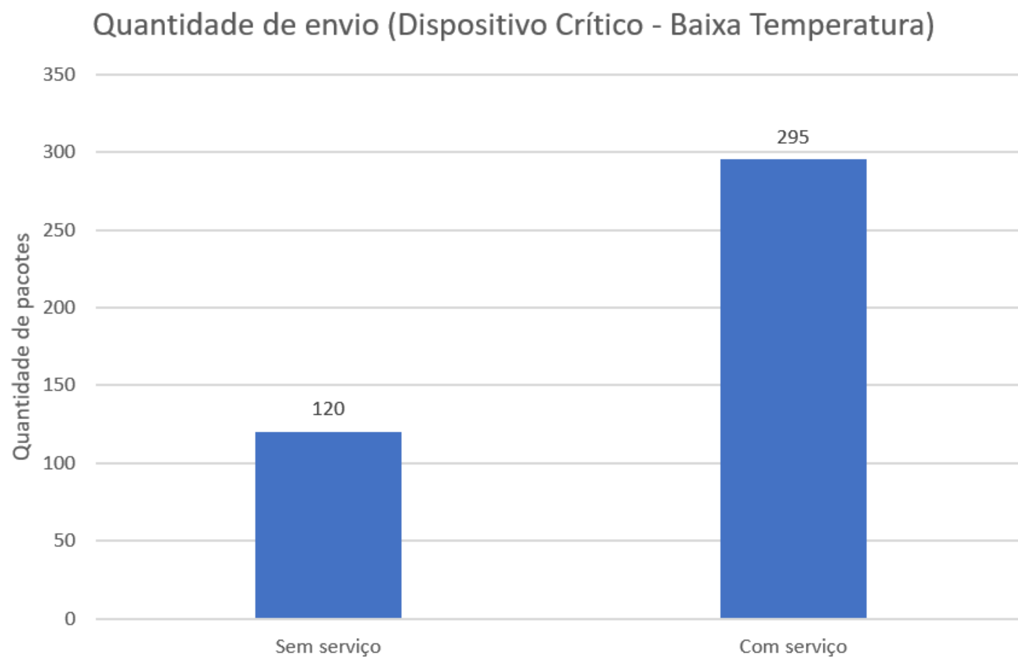

Figura 28 – Gráfico comparando otimização no dispositivo crítico em baixa temperatura.

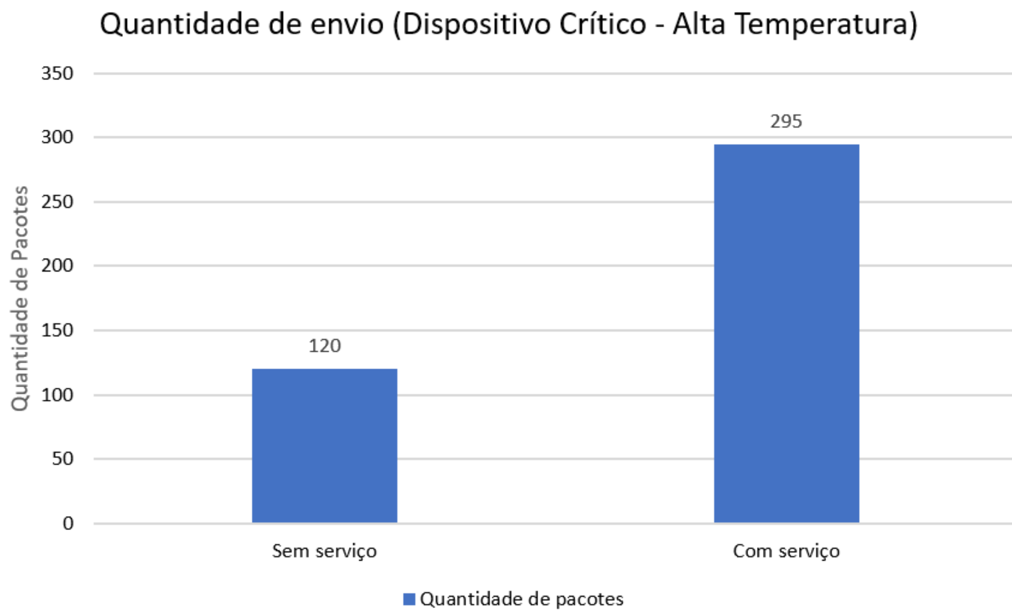

Figura 29 – Gráfico comparando otimização no dispositivo crítico em alta temperatura.

Para melhor visualização, a Figura 30 mostra a diferença de enviar 20 pacotes em estado crítico com e sem o serviço de otimização. É possível verificar que sem a otimização, demora 100 segundos para haver os 20 pacotes e com a otimização esse tempo cai para 46 segundos. Nos estados críticos, o serviço de otimização prioriza o maior volume de pacotes enviados, com intuito de aumentar a quantidade de informações dos dispositivo a fim de tomadas de decisões mais assertivas e de rápida resposta. Esse maior volume

de informações pode ser verificado na Figura 31, que compara a diferença do volume de dados enviados

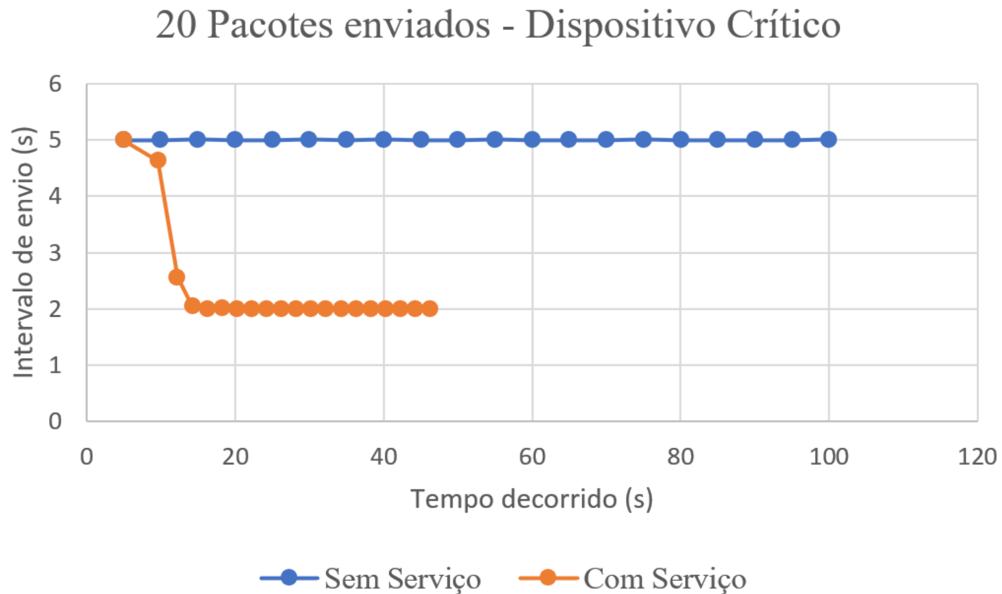

Figura 30 – Gráfico comparativo o envio de 20 pacotes com e sem otimização em estado crítico.

Na Figura 30, foi comparado o envio dos 20 primeiros pacotes. Sem o serviço, o intervalo de envio se manteve nos 5000ms (5 segundos). Já com o serviço ativo, é possível notar que o intervalo começa em 5000ms e tende a cair até o valor de 2000ms, que é o valor mínimo configurado. Esse resultado era esperado, já que o dispositivo esta em estado crítico. Como o intervalo de envio está menor, a quantidade de pacotes enviados é maior e consequentemente o volume de dados trafegados pela rede é maior, como visto na Figura 31.

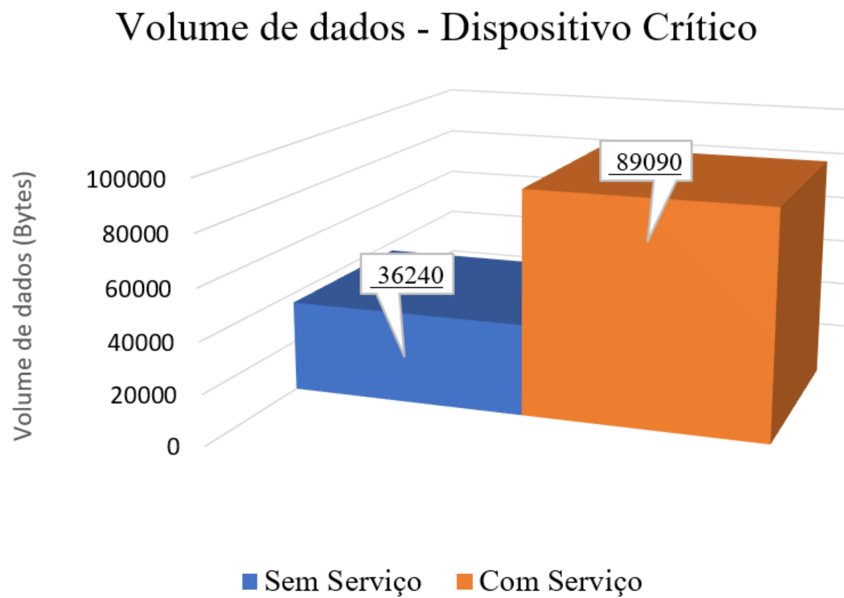

Figura 31 – Gráfico Volume de dados - dispositivo critico.

Quando o dispositivo está em estado super crítico, ou seja, a temperatura fica maior que a máxima permitida ou menor que a mínima, o comportamento da otimização é o mesmo do estado crítico, diminuir o intervalo de envio das telemetrias. A diferença desses dois estados está na velocidade em que o intervalo é alterado. No caso dessa simulação, o estado crítico altera o intervalo em 10% por mensagem recebida nesse estado e no super crítico a alteração é de 20% por vez. Essa porcentagem de alteração foi estipulada para essa simulação e pode ser alterada no serviço.

Abaixo estão as Figuras 32 e 33 correspondentes a quantidade de envio no estado super critico, com temperatura baixa e alta, respectivamente.

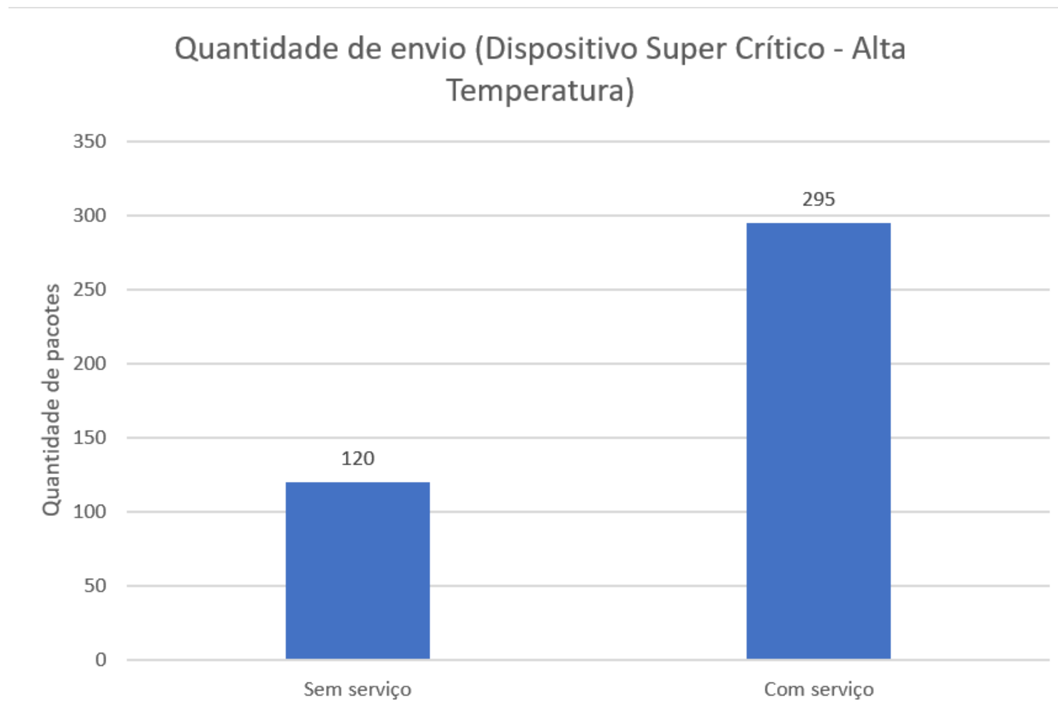

Figura 32 – Gráfico comparando otimização no dispositivo super crítico em alta temperatura.

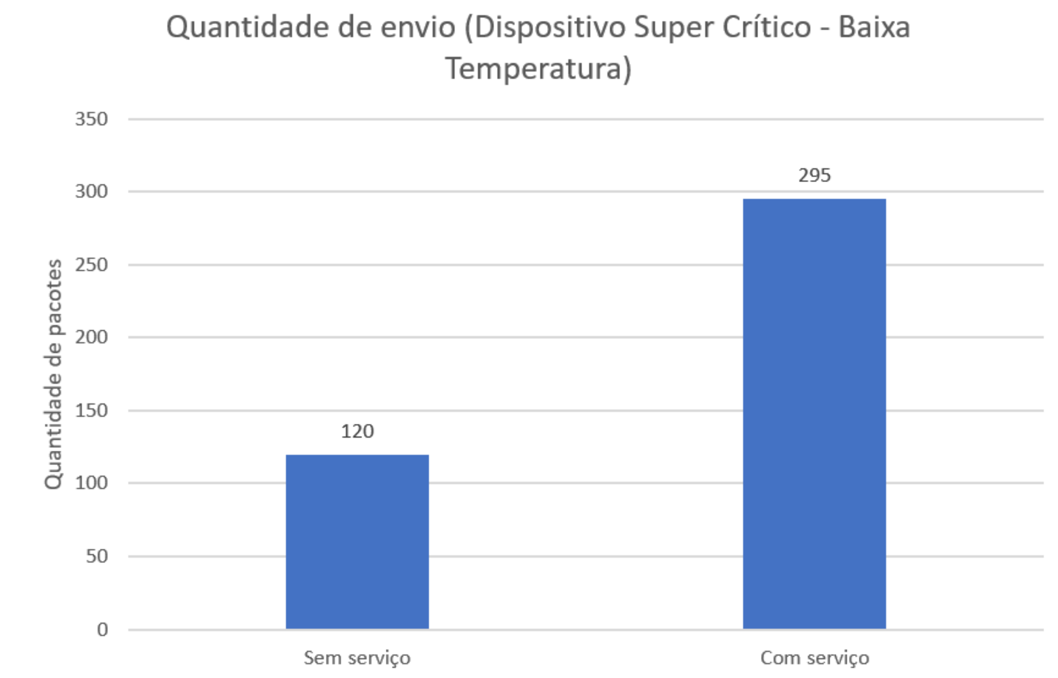

Figura 33 – Gráfico comparando otimização no dispositivo super crítico em baixa temperatura.

Os gráficos do dispositivo em estado crítico e super crítico são parecidos, a diferença é que a alteração do intervalo de envio no estado super crítico é mais agressiva, porém nos dois estados o serviço de otimização atende o requisito de intervalo mínimo, ou seja,

nos dois casos o intervalo final será o intervalo mínimo e no caso de estado super crítico esse intervalo mínimo será atingido antes, como na Figura 34.

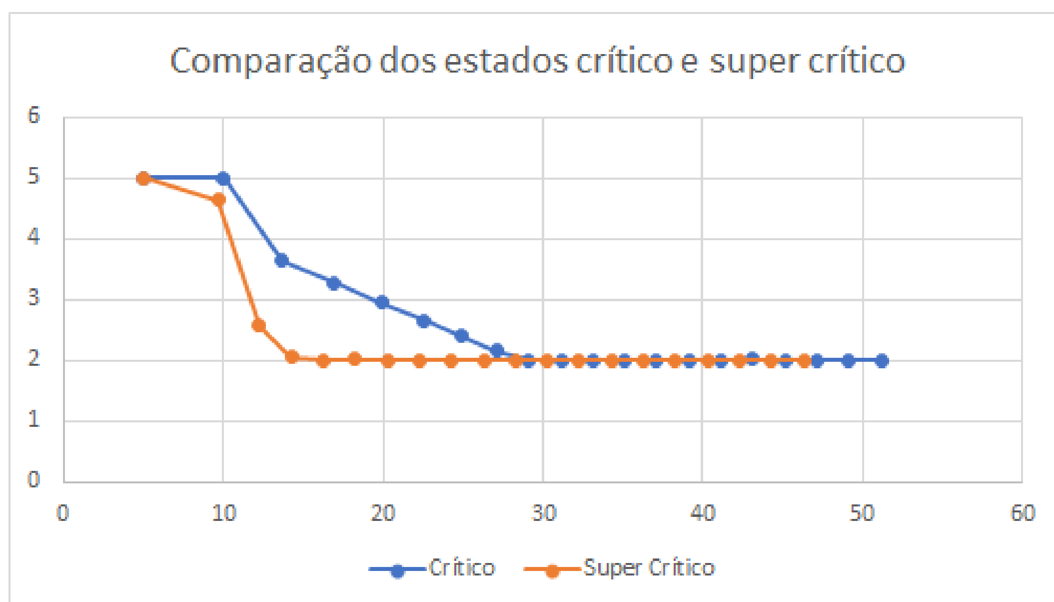

Figura 34 – Gráfico comparando o estado crítico ao super crítico.

Na Figura 34, pode-se notar que no estado Super Crítico a mudança do intervalo de envio é mais abrupta e acontece antes do estado Crítico, chegando ao intervalo mínimo pré-configurado de 2000ms (2 segundos) antes do estado crítico. Esse é o resultado esperado do serviço de otimização.

5 Experimentação

Nesse Capítulo serão apresentados experimentos utilizando o hardware NVIDIA Jetson Nano nas situações de estado normal, crítico e a transição entre o estado crítico e normal, quando o dispositivo está esfriando.

5.1 NVIDIA® Jetson Nano™

Anunciado no início de 2019, o módulo NVIDIA® Jetson Nano™, presente na Figura 35 seguida dos seus dados técnicos, é um computador desenvolvido para aplicações de processamento de borda (*edge computing*). Oferece visão computacional em tempo real e inferências em uma ampla variedade de modelos avançados de redes neurais profundas. Tais recursos permitem aplicações em robôs autônomos, dispositivos IoT com análise de ponta inteligente e sistemas avançados de IA (Inteligência Artificial) [63].

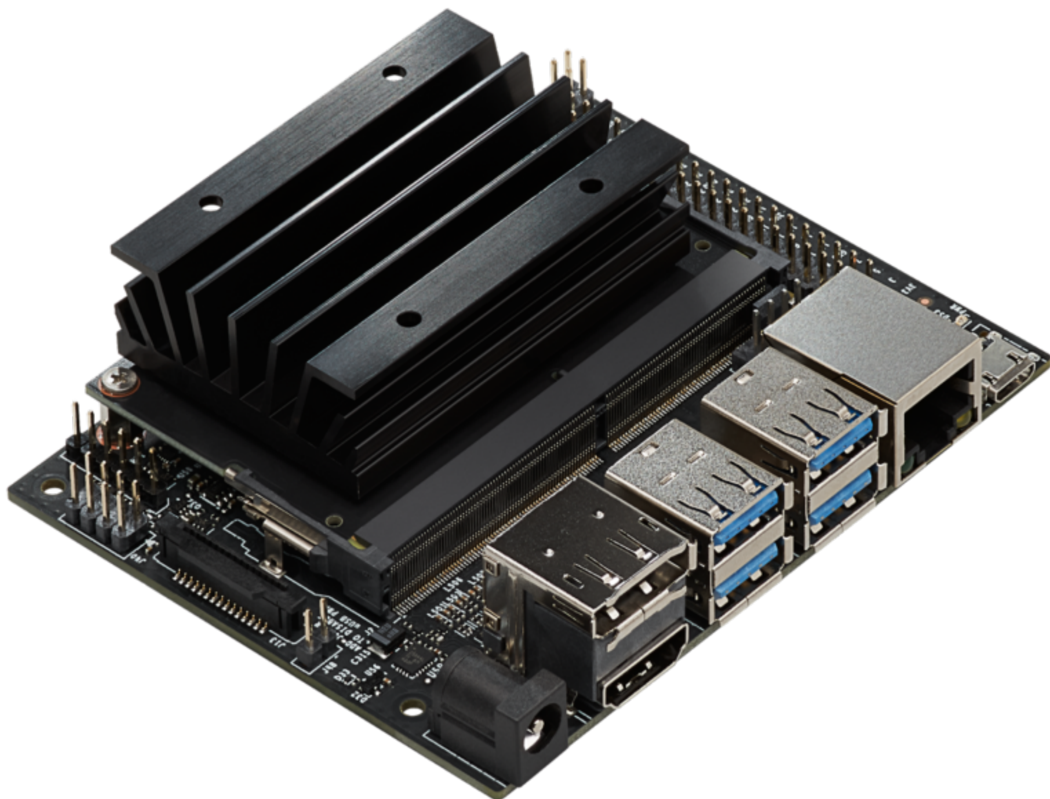

Figura 35 – Kit de desenvolvimento do NVIDIA Jerson Nano.

CPU	64-bit Quad-core ARM A57 @ 1.43GHz
GPU	128-core NVIDIA Maxwell @ 921MHz
Memória	4GB 64-bit LPDDR4 @ 1600MHz	25.6 GB/s
Video Encoder	4Kp30	(4x) 1080p30	(2x) 1080p60
Video Decoder	4Kp60	(2x) 4Kp30	(8x) 1080p30	(4x) 1080p60
USB	4x USB 3.0 A	USB 2.0 Micro B
Imagem	HDMI	DisplayPort
Rede	Gigabit Ethernet (RJ45)
Armazenamento	MicroSD card (16GB UHS-1)
Outras portas	(3x) I2C	(2x) SPI	UART	I2S	GPIOs

A última imagem do JetPack SDK é a versão 5.0.2, disponível no site do fabricante, compreende o Jetson Linux 35.1, Linux Kernel 5.10 baseado na distribuição Ubuntu 20.04, *drivers* da NVIDIA. Ao gravar um cartão com a imagem, é possível utilizar o módulo com todas as suas ferramentas instaladas [64].

5.2 Metodologia

Nessa experimentação, os dados de intervalo de envio mínimo, máximo e inicial seguiram os mesmo padrão da simulação, com os valores de 2000ms, 300000ms e 5000ms e o tempo dos experimentos foi definido como 10 minutos, mesmo período da simulação. Já a configuração de temperatura máxima é definida pelo guia térmico definido pela fabricante, no caso, 97°C.

Para confrontar os dados exibidos pelo código desenvolvido e o real cenário, foi utilizado um notebook com sistema operacional Windows como ponte de acesso entre o dispositivo Jetson Nano e a conexão com a internet, apresentado na Figura 36. Nessa disposição, pode-se extrair os pacotes enviados para o Hub IoT da Azure através do analisador de rede WireShark.

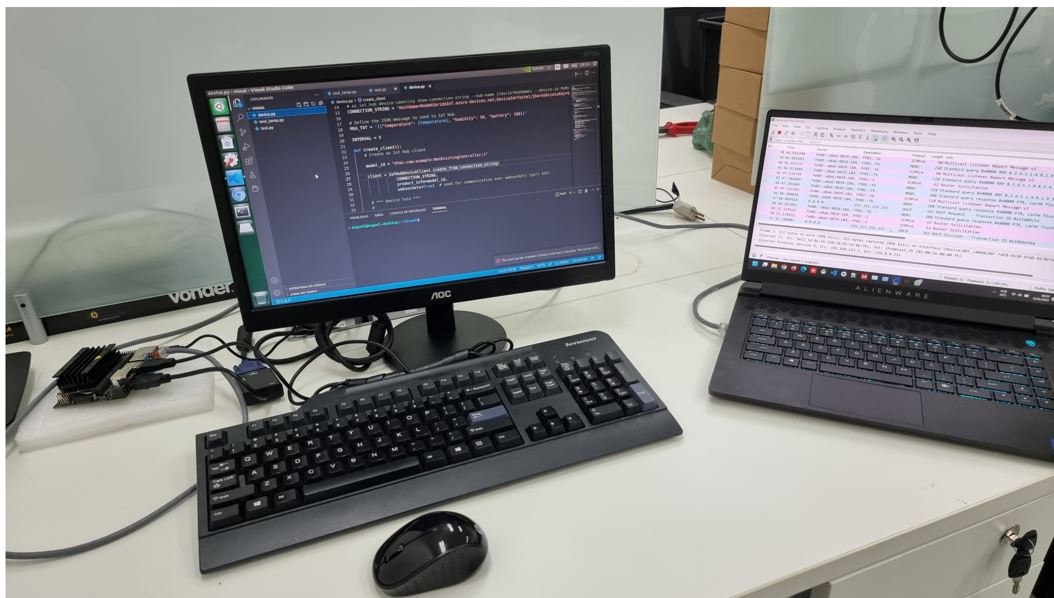

Figura 36 – Disposição dos testes.

O código desenvolvido para ser executado no Jetson Nano utiliza linguagem Python, segue o SDK da Microsoft para conexão com a Azure. Todo código está presente no Apêndice D.

Na imagem de sistema disponibilizado pela Nvidia está presente alguns exemplos de teste de carga, um deles é um exemplo utilizando a API proprietária da Nvidia chamada CUDA (Compute Unified Device Architecture) com foco em carga de trabalho para GPU (Unidade de processamento gráfico). Essa aplicação disponibilizada pela Nvidia foi responsável por enviar cargas de trabalho para a GPU, buscando aumentar a temperatura do sistema nos testes de dispositivo crítico.

5.3 Experimentos

O primeiro experimento foi no estado normal sem serviço ativo. A temperatura se manteve entre 52°C e 54°C e sem o serviço, o intervalo de envio não é alterado, mantendo o valor de 5 segundos. A Figura 37 mostra a temperatura e o intervalo de envio durante esse experimento.

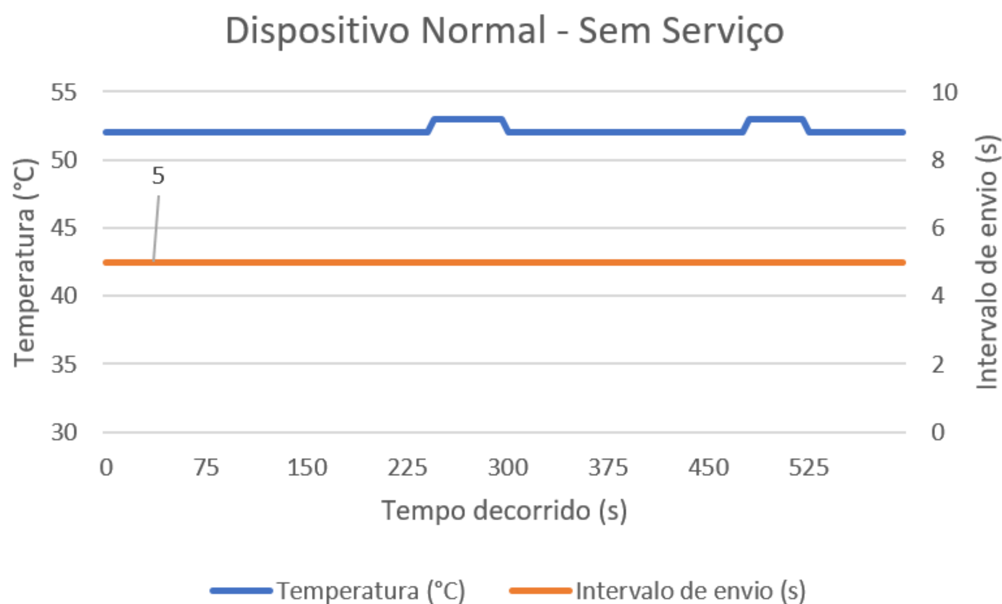

Figura 37 – Experimento - Dispositivo Normal - Sem Serviço.

Seguindo a mesma linha, o dispositivo estando em estado normal, porém com serviço ativo. A temperatura se manteve entre 53°C e 54°C e, nesse caso, com o serviço ativo, o intervalo de envio é modificado. Nesse experimento, o dispositivo está em estado normal e o serviço ativo, portanto o intervalo de envio sofrerá acréscimo até o limite, que é o intervalo de envio máximo. Na Figura 38 é possível verificar esse aumento do intervalo de envio ao decorrer do experimento, com a temperatura se mantendo em estado normal.

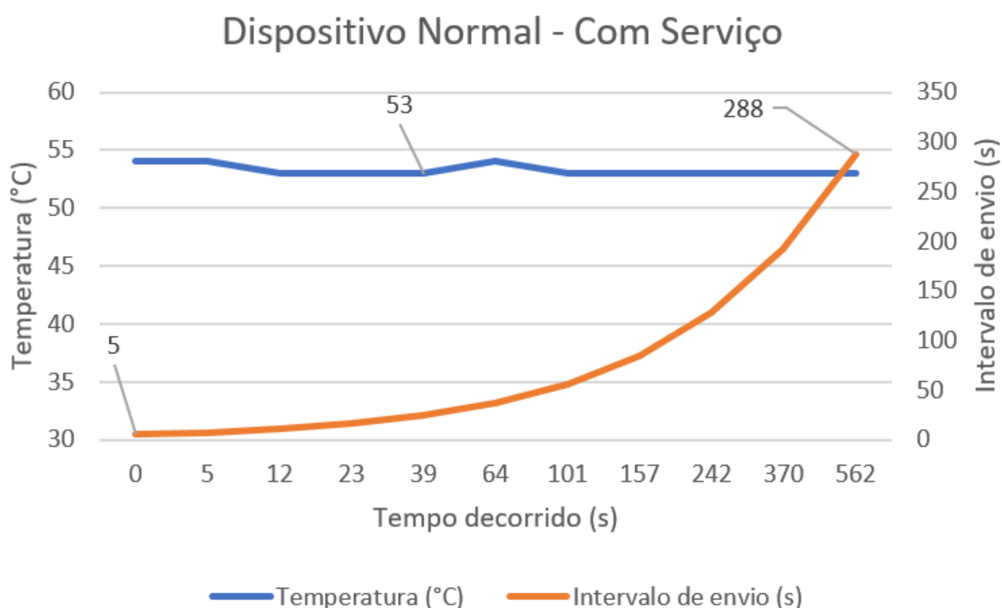

Figura 38 – Experimento - Dispositivo Normal - Com Serviço.

Na Figura 39, pode-se comparar o intervalo de envio e quantidade de pacotes

quando o dispositivo está normal e o serviço está desabilitado e habilitado.

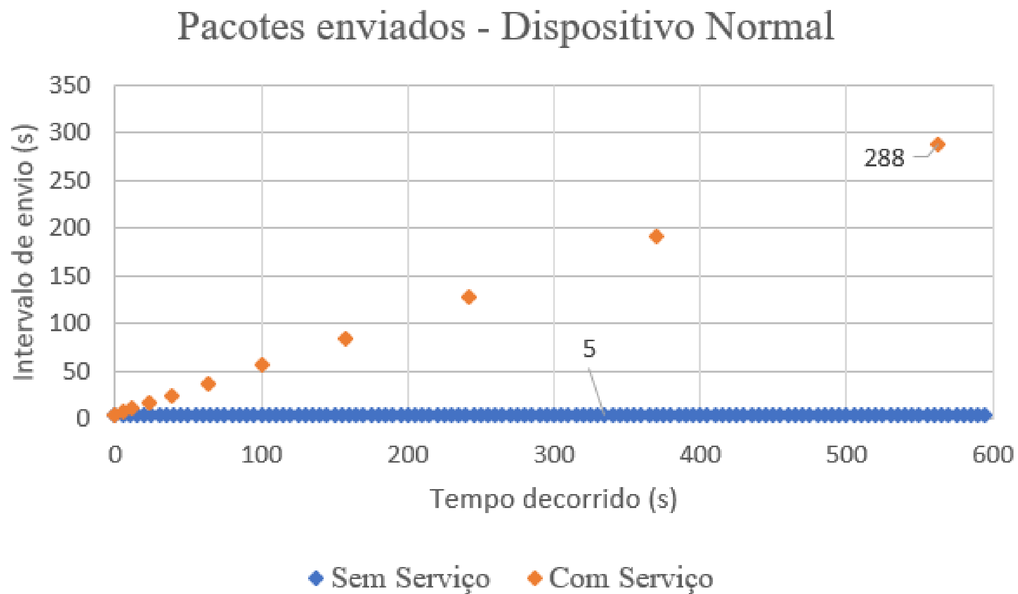

Figura 39 – Experimento - Dispositivo Normal - Comparação.

Já com o dispositivo em estado crítico, no teste sem o serviço ativo, a temperatura oscilou entre 75°C e 84°C e no teste com serviço ativo, 65°C e 79°C. Com a configuração atual da experimentação, o dispositivo é considerado crítico acima de 60°C e super crítico acima de 97°C. Na Figura 40 o dispositivo está crítico sem o serviço ativo e na figura 40 está crítico com o serviço ativo.

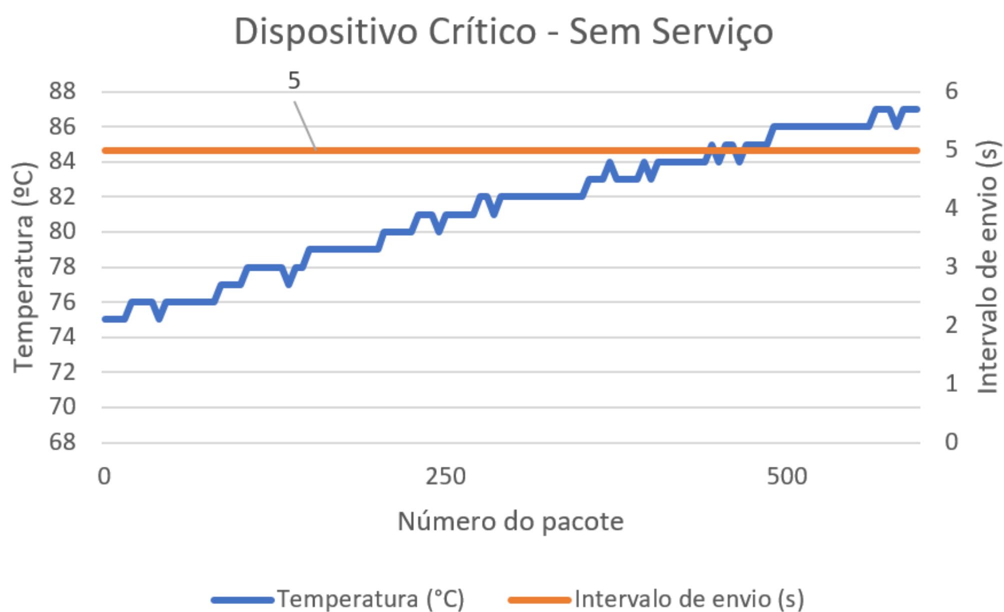

Figura 40 – Experimento - Dispositivo crítico - Sem Serviço.

O intervalo de envio se manteve em 5 segundos mesmo o dispositivo estando com temperatura alta. Na Figura 41 o serviço foi ativado e, por isso, o intervalo de envio foi decrementado, buscando o envio de mais pacotes ao servidor. O intervalo não decrementou menos de 2 segundos pois o intervalo mínimo foi configurado como 2 segundos.

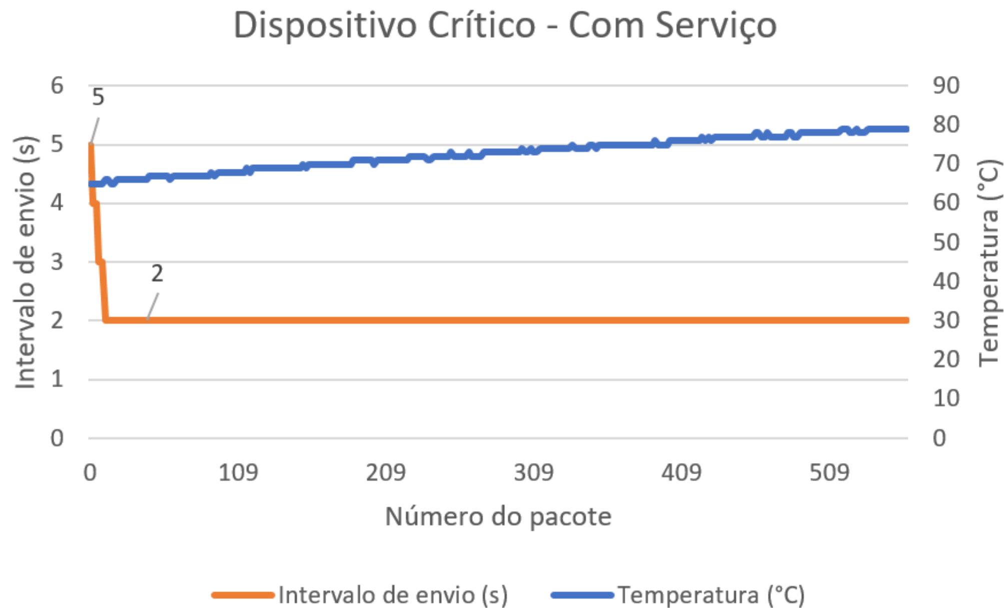

Figura 41 – Experimento - Dispositivo Crítico - Com Serviço.

Buscando mostrar a transição de estado do dispositivo, o próximo experimento foi feito começando no estado crítico, temperatura alta, e esfriando o dispositivo, até que o mesmo mudasse do estado crítico para o normal. O teste começou em estado critico com 5 segundos de intervalo de envio, o serviço decrementou esse intervalo até o limite mínimo de 2 segundos. No momento que o dispositivo sai do estado crítico e fica em estado normal (ponto de transição na Figura 42), o serviço começa a incrementar o intervalo de envio do mesmo. A Figura 42 mostra a temperatura e o intervalo de envio nesse teste.

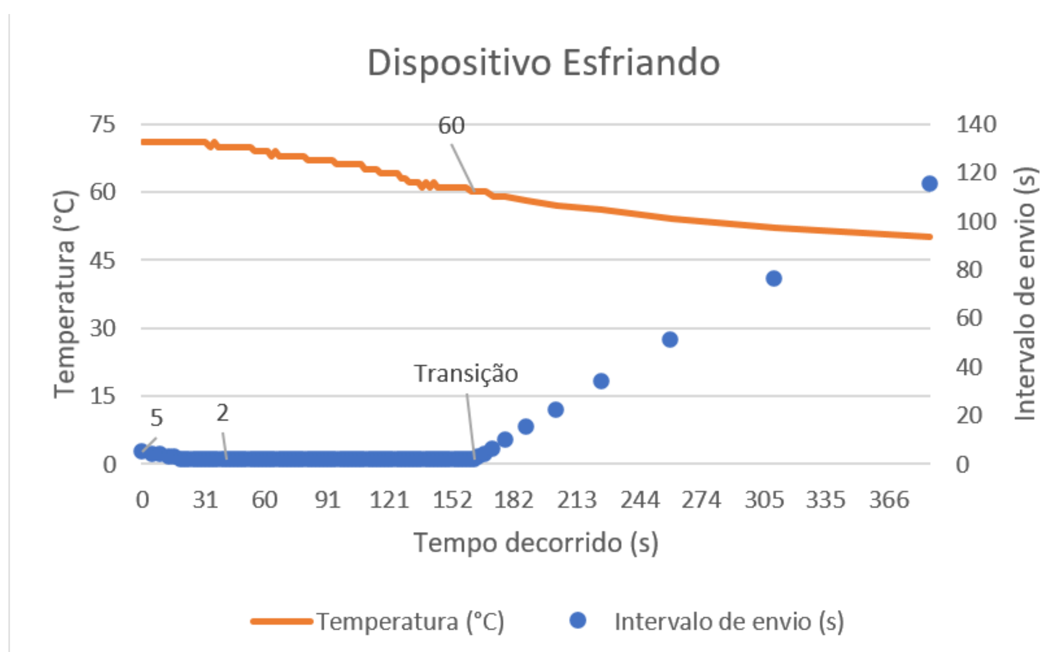

Figura 42 – Experimento - Dispositivo Esfriando - Com Serviço.

6 Conclusão

A partir dos resultados obtidos, é possível afirmar que o algoritmo desenvolvido em Python para o servidor foi capaz de identificar o estado atual do dispositivo, ao receber a telemetria do mesmo, e atribuir novas configurações de envio a partir do estado através dos novos parâmetros atribuídos ao dispositivo gêmeo referente aquele dispositivo.

Quando o estado do dispositivo está normal, o servidor prioriza a diminuição no envio de telemetria, consumindo menos energia do mesmo e reduzindo os custos de rede. Porém, quando o dispositivo está em estado crítico ou super crítico, a priorização é no volume de amostras, para reduzir o tempo de reação para tomada de decisão.

Essa mudança de configuração do dispositivo é realizado através do dispositivo gêmeo dos servidores da Azure. Ao mudar o dispositivo gêmeo, a nova configuração é enviada ao dispositivo. O dispositivo pode ter o algoritmo de alteração desses dados ou não. Se caso não tiver, a mudança não ocorrerá, porém o dispositivo pode trabalhar na mesma rede IoT sem problemas, só não terá a otimização.

Portanto, neste trabalho, o servidor dinâmico altera as configurações do dispositivo, intervalo de envio e políticas de desastre. Essa alteração ocorre de acordo com o estado do dispositivo, ou seja, aumenta o intervalo caso esteja tudo normal e diminui o intervalo caso o dispositivo esteja em estado crítico ou super crítico. Dessa forma, as configurações são alteradas enquanto o dispositivo está ativo, sem necessidade de pausa para atualização. A mudança das configurações são automáticas e ao mesmo tempo do dispositivo ativo.

6.1 Trabalhos futuros

Este trabalho foi desenvolvido utilizando dois tipos de parâmetros (temperatura e umidade) para testar e otimizar as configurações dos dispositivos. Como trabalho futuro pode-se adicionar novos parâmetros como: nível de bateria, sensor de luminosidade, sensor de proximidade, acelerômetro, dentre outros.

Outra linha de trabalho futuro é na validação do desempenho do algoritmo, da linguagem escolhida e do *hardware* que o servidor roda. Essa validação pode seguir duas linhas: quantidade de dispositivos na rede IoT que o servidor consegue processar e quantidade de mensagens (pacotes) recebidos pelo Hub IoT e tratadas pelo servidor.

Um linha distinta é trabalhar o servidor de otimização em um projeto real, com sensores, atuadores e rede em produção, sair do âmbito de simulação. Nesse caso, poderá verificar as mudanças tanto no acréscimo do intervalo de envio, caso o dispositivo esteja em estado normal, quanto o decréscimo, caso o dispositivo esteja em um dos estados

críticos.

Apêndices

APÊNDICE A – Código em Python do servidor inteligente

```
# --------------------------------------------------
# Copyright (c) Microsoft Corporation. All rights reserved.
# Licensed under the MIT License. See License.txt in the project root
for license information.
# --------------------------------------------------

"""
This sample demonstrates how to use the Microsoft Azure Event Hubs Client
for Python sync API to read messages sent from a device. Please see
the documentation for @azure/event-hubs package for more details at
https://pypi.org/project/azure-eventhub/ For an example that uses
checkpointing, follow up this sample with the sample in the azure-
eventhub-checkpointstoreblob package on GitHub at the following link:
https://github.com/Azure/azure-sdk-for-python/blob/master/sdk/eventhub/
azure-eventhub-checkpointstoreblob/samples/receive_events_using_
checkpoint_store.py
"""


from azure.eventhub import TransportType
from azure.eventhub import EventHubConsumerClient
from azure.iot.hub.protocol.models import twin_properties
import requests
from time import sleep
from azure.iot.hub import IoTHubRegistryManager
from azure.iot.hub.models import Twin, TwinProperties
import json

IOTHUB_CONNECTION_STRING = "HostName=RedeHibridaIoT.azure-devices.net;
SharedAccessKeyName=iothubowner;SharedAccessKey=pTOwcDX3j+NkkjyjsWsTNp
3sG9iNYTh25Yjte2oxxHE="

# Event Hub-compatible endpoint
```

```
# az iot hub show --query properties.eventHubEndpoints.events.endpoint
#--name {your IoT Hub name}
#EVENTHUB_COMPATIBLE_ENDPOINT = "{your Event Hubs compatible endpoint}"
EVENTHUB_COMPATIBLE_ENDPOINT = "iothub-ehub-redehibrid-11558182-
e1b8403e82"

# Event Hub-compatible name
# az iot hub show --query properties.eventHubEndpoints.events.path
#--name {your IoT Hub name}
EVENTHUB_COMPATIBLE_PATH = "iothub-ehub-redehibrid-11558182-e1b8403e82"

# Primary key for the "service" policy to read messages
# az iot hub policy show --name service --query primaryKey --hub-name
#{your IoT Hub name}
#IOTHUB_SAS_KEY = "{your service primary key}"

# If you have access to the Event Hub-compatible connection string
#from the Azure portal, then
# you can skip the Azure CLI commands above, and assign the connection
#string directly here.
CONNECTION_STR = "Endpoint=sb://ihsuprodcqres016dednamespace.servicebus.
windows.net/;SharedAccessKeyName=iothubowner;SharedAccessKey=pTOwcDX3j+
NkkjyjsWsTNp3sG9iNYTh25Yjte2oxxHE=;EntityPath=iothub-ehub-redehibrid-
11558182-e1b8403e82"

quantidade = 0
url = 'http://127.0.0.1:1880/server'

interval_min = 10
interval_max = 300000 #5 minutos

retrycount_normal = 10
retrycount_critic = 100
retrycount_super_critic = 1000

minbackoff_min = 10
minbackoff_max = 1000

maxbackoff_min = 1000
```

```
maxbackoff_max = 100000

deltabackoff_normal = 1000
deltabackoff_critic = 500
deltabackoff_super_critic = 250

firstfastretry_normal = False
firstfastretry_critic = True

def iothub_device_twin(device_id, temperature, humidity, battery):
    try:

        #Ajustando temperatura de graus para kelvin
        temperature+=273

        #Configuração do range de temperatura e de umidade
        temperature_min = 10 +273
        temperature_max = 80 +273
        humidity_min=20
        humidity_max=70

        #Configuração do incremendo e decremendo dependendo do
        #estado do device
        increment_normal = 0.5 #50%
        increment_critic = 0.1 #10%
        increment_super_critic = 0.2 #20%

        print("Iniciando atualização do dispositivo gêmeo: "+device_id)

        iothub_registry_manager = IoTHubRegistryManager
        (IOTHUB_CONNECTION_STRING)

        new_tags = {}

        twin = iothub_registry_manager.get_twin(device_id)
        twin_pro = twin.__getattribute__('properties')
        twin_desired = twin_pro.__getattribute__('desired')
        twin_config = twin_desired['telemetryConfig']
        interval = twin_config['interval']
```

```
retrycount = twin_config['retrycount']
minbackoff = twin_config['minbackoff']
maxbackoff = twin_config['maxbackoff']
deltabackoff = twin_config['deltabackoff']
firstfastretry = twin_config['firstfastretry']

print()
print('Configuração antes: ')
print('interval: '+str(interval))
print('retrycount: '+str(retrycount))
print('minbackoff: '+str(minbackoff))
print('maxbackoff: '+str(maxbackoff))
print('deltabackoff: '+str(deltabackoff))
print('firstfastretry: '+str(firstfastretry))
print()

print("Iniciando as comparações:")

status_normal = "normal"
status_critic = "critic"
status_super_critic = "super critic"
status = status_normal


if(temperature<(temperature_min) or
temperature>(temperature_max)):
    status = status_super_critic
    print("Temperatura super crítica")
else:
    if(humidity<(humidity_min) or humidity>(humidity_max)):
        print("Umidade super crítica")
        status = status_super_critic
    else:
        if(temperature<(temperature_min*1.1) or
        temperature>(temperature_max*0.9)):
            status = status_critic
            print("Temperatura crítica")
        else:
            if(humidity<(humidity_min*1.1) or
```

```
            humidity>(humidity_max*0.9)):
                print("Umidade crítica")
                status = status_critic

    if(status == "super critic"):
        interval*=(1-increment_super_critic)
        retrycount = retrycount_super_critic
        minbackoff*=(1-increment_super_critic)
        if(minbackoff<minbackoff_min):
            minbackoff=minbackoff_min
        maxbackoff*=(1+increment_super_critic)
        if(maxbackoff>maxbackoff_max):
            maxbackoff=maxbackoff_max
        deltabackoff=deltabackoff_super_critic
        firstfastretry=firstfastretry_critic
    else:
        if(status == "critic"):
            interval*=(1-increment_critic)
            retrycount = retrycount_critic
            minbackoff*=(1-increment_critic)
            if(minbackoff<minbackoff_min):
                minbackoff=minbackoff_min
            maxbackoff*=(1+increment_critic)
            if(maxbackoff>maxbackoff_max):
                maxbackoff=maxbackoff_max
            deltabackoff=deltabackoff_critic
            firstfastretry=firstfastretry_critic
        else:
            if(status == "normal"):
                print("Normal")
                interval*=(1+increment_normal)
                retrycount = retrycount_super_critic
                minbackoff*=(1+increment_normal)
                if(minbackoff<minbackoff_min):
                    minbackoff=minbackoff_min
                maxbackoff*=(1+increment_critic)
                if(maxbackoff>maxbackoff_max):
                    maxbackoff=maxbackoff_max
                deltabackoff=deltabackoff_normal
```

```
                firstfastretry=firstfastretry_normal

        twin_properties = TwinProperties(desired={  'telemetryConfig' : {
        'interval' : interval, 'retrycount' : retrycount,
        'minbackoff' : minbackoff, 'maxbackoff': maxbackoff,
        'deltabackoff': deltabackoff, 'firstfastretry': firstfastretry}})
        twin_patch = Twin(tags=new_tags, properties= twin_properties)
        twin = iothub_registry_manager.update_twin(device_id, twin_patch,
        twin.etag)

        print()
        print("Atualizou o dispositivo: "+device_id)
        print('Configuração depois: ')
        print('interval: '+str(interval))
        print('retrycount: '+str(retrycount))
        print('minbackoff: '+str(minbackoff))
        print('maxbackoff: '+str(maxbackoff))
        print('deltabackoff: '+str(deltabackoff))
        print('firstfastretry: '+str(firstfastretry))
        print()

    except Exception as ex:
        print("Unexpected error {0}".format(ex))
        return
    except KeyboardInterrupt:
        print("IoT Hub Device Twin service sample stopped")

# Define callbacks to process events
def on_event_batch(partition_context, events):
    for event in events:
        #print("Received event from partition: {}.".format(partition_
        #context.partition_id))
        print()
        print("Nova mensagem:")
        print("Dispositivo: ", event.properties)
        print("Telemetria: ", event.body_as_str())

        #print("System properties (set by IoT Hub): ",
        event.system_properties)
```

```
        print()

        s = event.body_as_str()
        #add {} to read as json
        r = "{" + s
        r = r + "}"

        d = json.loads(r)

        temperature = d['temperature']
        humidity = d['humidity']
        battery = d['battery']

        print("Temperatura: " +str(temperature))
        print("Umidade: " +str(humidity))
        print("Bateria: " +str(battery))

        myobj = {'msg': s }

        requests.post(url, data = myobj)

        device = event.properties[b'$.cdid']

        myobj = {'device': device}

        requests.post(url, data = myobj)

        global quantidade
        quantidade+=1

        myobj = {'qtd': quantidade}
        requests.post(url, data = myobj)
        device =  str(device, 'UTF-8')



    partition_context.update_checkpoint()
    iothub_device_twin(device, temperature, humidity, battery)
```

```
def on_error(partition_context, error):
    # Put your code here. partition_context can be None in the on_error
    #callback.
    if partition_context:
        print("An exception:{} occurred during receiving from Partition:
        {}.".format(partition_context.partition_id,
            error
        ))
    else:
        print("An exception:{}occurred during the load balance process."
        .format(error))



def main():
    client = EventHubConsumerClient.from_connection_string(
        conn_str=CONNECTION_STR,
        consumer_group="$default",
        transport_type=TransportType.AmqpOverWebsocket,
    )

    myobj = {'status': 'on'}
    x = requests.post(url, data = myobj)

    print("Servidor iniciado.")

    try:
        with client:
            client.receive_batch(
                on_event_batch=on_event_batch,
                on_error=on_error
            )
    except KeyboardInterrupt:
        myobj = {'status': 'off'}

        x = requests.post(url, data = myobj)

        print(x.text)
        print("Servidor desligando.")
```

```
if __name__ == '__main__':
    main()
```

APÊNDICE B – Código em Java (Android) que simula o dispositivo IoT

```
package com.microsoft.azure.iot.sdk.samples.androidsample;

import android.annotation.SuppressLint;
import android.content.Context;
import android.os.BatteryManager;
import android.os.Bundle;
import android.os.Handler;
import android.support.v7.app.AlertDialog;
import android.support.v7.app.AppCompatActivity;
import android.text.Layout;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;

import com.microsoft.azure.sdk.iot.device.DeviceClient;
import com.microsoft.azure.sdk.iot.device.DeviceTwin.Device;
import com.microsoft.azure.sdk.iot.device.DeviceTwin.DeviceMethodData;
import com.microsoft.azure.sdk.iot.device.DeviceTwin.Property;
import com.microsoft.azure.sdk.iot.device.IotHubClientProtocol;
import com.microsoft.azure.sdk.iot.device.IotHubConnectionStatusChange
Callback;
import com.microsoft.azure.sdk.iot.device.IotHubConnectionStatusChange
Reason;
import com.microsoft.azure.sdk.iot.device.IotHubEventCallback;
import com.microsoft.azure.sdk.iot.device.IotHubMessageResult;
import com.microsoft.azure.sdk.iot.device.IotHubStatusCode;
import com.microsoft.azure.sdk.iot.device.Message;
import com.microsoft.azure.sdk.iot.device.transport.ExponentialBackoff
WithJitter;
import com.microsoft.azure.sdk.iot.device.transport.IotHubConnection
Status;
```

```
import com.microsoft.azure.sdk.iot.device.transport.RetryPolicy;

import org.json.JSONException;
import org.json.JSONObject;

import java.io.IOException;
import java.net.URISyntaxException;
import java.nio.charset.StandardCharsets;


public class MainActivity extends AppCompatActivity {

    private double temperature;
    private double humidity;
    private Message sendMessage;
    private String lastException;

    private DeviceClient client;

    IotHubClientProtocol protocol = IotHubClientProtocol.MQTT;

    Button btnStart;
    Button btnStop;

    TextView txtMsgsSentVal;
    TextView txtLastTempVal;
    TextView txtLastHumidityVal;
    TextView txtLastMsgSentVal;
    TextView txtLastMsgReceivedVal;
    TextView txtLastMsgInterval;
    TextView txtRetryCountVal;
    TextView txtMinBackOffVal;
    TextView txtMaxBackOffVal;
    TextView txtDeltaBackOffVal;
    TextView txtFirstFastRetryVal;

    private int msgSentCount = 0;
    private int receiptsConfirmedCount = 0;
    private int sendFailuresCount = 0;
```

```
    private int msgReceivedCount = 0;
    private int sendMessagesInterval = 5000;

    private int retryCount = 50000;
    private int minBackOff = 100;
    private int maxBackOff = 100000;
    private int deltaBackOff = 100;
    private boolean firstFastRetry = false;

    private final Handler handler = new Handler();
    private Thread sendThread;

    private static final int METHOD_SUCCESS = 200;
    public static final int METHOD_THROWS = 403;
    private static final int METHOD_NOT_DEFINED = 404;

    @SuppressLint("WrongConstant")
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main2);

        btnStart = findViewById(R.id.btnStart);
        btnStop = findViewById(R.id.btnStop);

        txtMsgsSentVal = findViewById(R.id.txtMsgsSentVal);
        txtLastTempVal = findViewById(R.id.txtLastTempVal);
        txtLastHumidityVal = findViewById(R.id.txtLastHumidityVal);
        txtLastMsgSentVal = findViewById(R.id.txtLastMsgSentVal);
        txtLastMsgReceivedVal = findViewById(R.id.txtLastMsgReceivedVal);
        txtLastMsgInterval = findViewById(R.id.txtSendIntervalVal);
        txtRetryCountVal = findViewById(R.id.txtRetryCountVal);
        txtMinBackOffVal = findViewById(R.id.txtMinBackOffVal);
        txtMaxBackOffVal = findViewById(R.id.txtMaxBackOffVal);
        txtDeltaBackOffVal = findViewById(R.id.txtDeltaBackOffVal);
        txtFirstFastRetryVal = findViewById(R.id.txtFirstFastRetryVal);

        txtLastMsgSentVal.setBreakStrategy(Layout.BREAK_STRATEGY_SIMPLE);
```

```
        btnStop.setEnabled(false);
    }

    private void stop()
    {
        new Thread(() -> {
            try
            {
                sendThread.interrupt();
                client.closeNow();
                System.out.println("Shutting down...");
            }
            catch (Exception e)
            {
                lastException = "Exception while closing IoTHub
                connection: " + e;
                handler.post(exceptionRunnable);
            }
        }).start();
    }

    public void btnStopOnClick(View v)
    {
        stop();

        btnStart.setEnabled(true);
        btnStop.setEnabled(false);
    }

    private void start()
    {
        sendThread = new Thread(() -> {
            try
            {
                initClient();
                for(;;)
                {
                    sendMessages();
```

```
                Thread.sleep(sendMessagesInterval);
            }
        }
        catch (InterruptedException e)
        {
            //return;
        }
        catch (Exception e)
        {
            lastException = "Exception while opening IoTHub
            connection: " + e;
            handler.post(exceptionRunnable);
        }
    });

    sendThread.start();
}

public void btnStartOnClick(View v)
{
    start();

    btnStart.setEnabled(false);
    btnStop.setEnabled(true);
}

final Runnable updateRunnable = new Runnable() {
    @SuppressLint({"DefaultLocale", "SetTextI18n"})
    public void run() {
        txtLastTempVal.setText(String.format("%.2f",temperature));
        txtLastHumidityVal.setText(String.format("%.2f",humidity));
        txtMsgsSentVal.setText(Integer.toString(msgSentCount));
        txtLastMsgSentVal.setText("[" + new String(sendMessage.
        getBytes(), Message.DEFAULT_IOTHUB_MESSAGE_CHARSET) + "]");
        txtLastMsgInterval.setText(String.valueOf
        (sendMessagesInterval));

        txtRetryCountVal.setText(String.valueOf(retryCount));
        txtMinBackOffVal.setText(String.valueOf(minBackOff));
```

```
            txtMaxBackOffVal.setText(String.valueOf(maxBackOff));
            txtDeltaBackOffVal.setText(String.valueOf(deltaBackOff));
            txtFirstFastRetryVal.setText(String.valueOf(firstFastRetry));

        }
    };

    final Runnable exceptionRunnable = new Runnable() {
        public void run() {
            AlertDialog.Builder builder = new AlertDialog.Builder
            (MainActivity.this);
            builder.setMessage(lastException);
            builder.show();
            System.out.println(lastException);
            btnStart.setEnabled(true);
            btnStop.setEnabled(false);
        }
    };

    final Runnable methodNotificationRunnable = () -> {
        Context context = getApplicationContext();
        CharSequence text = "Set Send Messages Interval to "
        + sendMessagesInterval + "ms";
        int duration = Toast.LENGTH_LONG;

        Toast toast = Toast.makeText(context, text, duration);
        toast.show();
    };

    @SuppressLint("DefaultLocale")
    private void sendMessages()
    {
        temperature = 20.0 + Math.random() * 100;
        humidity = 30.0 + Math.random() * 200;
        BatteryManager bm = (BatteryManager) getApplicationContext()
        .getSystemService(BATTERY_SERVICE);
        int batLevel = bm.getIntProperty(BatteryManager
        .BATTERY_PROPERTY_CAPACITY);
        Log.e("battery", String.valueOf(batLevel));
```

```
    String msgStr = "\"temperature\": " + (int) temperature +
            ", \"humidity\": " + (int) humidity +
            ", \"battery\": " + batLevel;
    try
    {
        sendMessage = new Message(msgStr);
        sendMessage.setProperty("temperatureAlert", temperature > 28
        ? "true" : "false");
        sendMessage.setMessageId(java.util.UUID.randomUUID()
        .toString());
        System.out.println("Message Sent: " + msgStr);
        EventCallback eventCallback = new EventCallback();
        client.sendEventAsync(sendMessage, eventCallback,
        msgSentCount);
        msgSentCount++;
        handler.post(updateRunnable);
    }
    catch (Exception e)
    {
        System.err.println("Exception while sending event: " + e);
    }
}

protected static class DeviceTwinStatusCallBack implements
IotHubEventCallback {
    @Override
    public void execute(IotHubStatusCode status, Object context) {
        System.out.println("IoT Hub responded to device twin
        operation with status " + status.name());
    }
}

private void initClient() throws URISyntaxException, IOException
{
    String connString = BuildConfig.DeviceConnectionString;
    client = new DeviceClient(connString, protocol);

    // Create a Device object to store the device twin properties
```

```
Device dataCollector = new Device() {
    // Print details when a property value changes
    @Override
    public void PropertyCall(String propertyKey, Object
    propertyValue, Object context) {
        System.out.println(propertyKey + " changed to "
        + propertyValue);
        Log.e("devicetwin", propertyKey+ "   "+propertyValue);
        try
        {
            JSONObject json = new JSONObject(propertyValue
            .toString());
            sendMessagesInterval = json.getInt("interval");
            retryCount = json.getInt("retrycount");
            minBackOff = json.getInt("minbackoff");
            maxBackOff = json.getInt("maxbackoff");
            deltaBackOff = json.getInt("deltabackoff");
            firstFastRetry = json.getBoolean("firstfastretry");

            RetryPolicy retryPolicy = new ExponentialBackoffWith
            Jitter(
                    retryCount,
                    minBackOff,
                    maxBackOff,
                    deltaBackOff,
                    firstFastRetry);
            client.setRetryPolicy(retryPolicy);

        } catch (JSONException e)
        {
            e.printStackTrace();
            Log.e("error", e.toString());
        }

    }
};

try
{
```

```
            client.registerConnectionStatusChangeCallback(new
            IotHubConnectionStatusChangeCallbackLogger(), new Object());
            client.open();
            MessageCallback callback = new MessageCallback();
            client.setMessageCallback(callback, null);
            client.subscribeToDeviceMethod(new SampleDeviceMethod
            Callback(), getApplicationContext(), new DeviceMethodStatus
            CallBack(), null);
            client.startDeviceTwin(new DeviceTwinStatusCallBack(), null,
            dataCollector, null);

            // Create a reported property and send it to your IoT hub.
            dataCollector.setReportedProp(new Property
            ("connectivityType", "cellular"));
            client.sendReportedProperties(dataCollector
            .getReportedProp());

            RetryPolicy retryPolicy = new ExponentialBackoffWithJitter
            (Integer.MAX_VALUE,100,1000,100,false);
            client.setRetryPolicy(retryPolicy);

        }
        catch (Exception e)
        {
            System.err.println("Exception while opening IoTHub
            connection: " + e);
            client.closeNow();
            System.out.println("Shutting down...");
        }
    }

    class EventCallback implements IotHubEventCallback
    {
        public void execute(IotHubStatusCode status, Object context)
        {
            int i = context instanceof Integer ? (Integer) context : 0;
            System.out.println("IoT Hub responded to message " + i
                    + " with status " + status.name());
```

```
            if((status == IotHubStatusCode.OK) ||
            (status == IotHubStatusCode.OK_EMPTY))
            {
                TextView txtReceiptsConfirmedVal = findViewById
                (R.id.txtReceiptsConfirmedVal);
                receiptsConfirmedCount++;
                txtReceiptsConfirmedVal.setText
                (String.valueOf(receiptsConfirmedCount));
            }
            else
            {
                TextView txtSendFailuresVal = findViewById
                (R.id.txtSendFailuresVal);
                sendFailuresCount++;
                txtSendFailuresVal.setText
                (String.valueOf(sendFailuresCount));
            }
        }
    }

    class MessageCallback implements com.microsoft.azure.sdk.iot.device
    .MessageCallback
    {
        public IotHubMessageResult execute(Message msg, Object context)
        {
            System.out.println("Received message with content: "
            + new String
            (msg.getBytes(),
            Message.DEFAULT_IOTHUB_MESSAGE_CHARSET));
            msgReceivedCount++;
            TextView txtMsgsReceivedVal = findViewById
            (R.id.txtMsgsReceivedVal);
            txtMsgsReceivedVal.setText(String.valueOf(msgReceivedCount));
            String msg2 = "[" + new String(msg.getBytes(),
            Message.DEFAULT_IOTHUB_MESSAGE_CHARSET) + "]";
            txtLastMsgReceivedVal.setText(msg2);
            return IotHubMessageResult.COMPLETE;
        }
    }
```

```
    protected static class IotHubConnectionStatusChangeCallbackLogger
    implements IotHubConnectionStatusChangeCallback
    {
        @Override
        public void execute(IotHubConnectionStatus status,
        IotHubConnectionStatusChangeReason statusChangeReason,
        Throwable throwable, Object callbackContext)
        {
            System.out.println();
            System.out.println("CONNECTION STATUS UPDATE: " + status);
            System.out.println("CONNECTION STATUS REASON: " + status
            ChangeReason);
            System.out.println("CONNECTION STATUS THROWABLE: " +
            (throwable == null ? "null" : throwable.getMessage()));
            System.out.println();

//            if (throwable != null)
//            {
//                throwable.printStackTrace();
//            }
//
//            if (status == IotHubConnectionStatus.DISCONNECTED)
//            {
//                connection was lost, and is not being re-established.
//                Look at provided exception for how to resolve this
//                issue.
//                Cannot send messages until this issue is resolved,
//                and you manually re-open the device client
//            }
//            else if (status == IotHubConnectionStatus.DISCONNECTED_
//            RETRYING)
//            {
//                connection was lost, but is being re-established.
//                Can still send messages, but they won't be sent until
//                the connection is re-established
//            }
//            else if (status == IotHubConnectionStatus.CONNECTED)
//            {
```

```
//                  Connection was successfully re-established.
//                Can send messages.
//            }
        }
    }

    private int method_setSendMessagesInterval(Object methodData)
    throws JSONException
    {
        String payload = new String((byte[])methodData,
        StandardCharsets.UTF_8).replace("\"", "");
        JSONObject obj = new JSONObject(payload);
        sendMessagesInterval = obj.getInt("sendInterval");
        handler.post(methodNotificationRunnable);
        return METHOD_SUCCESS;
    }

    private int method_default(Object data)
    {
        System.out.println("invoking default method for this device: "
        + data.toString());
        // Insert device specific code here
        return METHOD_NOT_DEFINED;
    }

    protected static class DeviceMethodStatusCallBack
    implements IotHubEventCallback
    {
        public void execute(IotHubStatusCode status, Object context)
        {
            System.out.println("IoT Hub responded to device method
            operation with status " + status.name());
        }
    }

    protected class SampleDeviceMethodCallback implements
    com.microsoft.azure.sdk.iot.device.DeviceTwin.DeviceMethodCallback
    {
        @Override
```

```
        public DeviceMethodData call(String methodName, Object
        methodData, Object context)
        {
            DeviceMethodData deviceMethodData ;
            try {
                if ("setSendMessagesInterval".equals(methodName)) {
                    int status = method_setSendMessagesInterval
                    (methodData);
                    deviceMethodData = new DeviceMethodData(status,
                    "executed " + methodName);
                } else {
                    int status = method_default(methodData);
                    deviceMethodData = new DeviceMethodData(status,
                    "executed " + methodName);
                }
            }
            catch (Exception e)
            {
                deviceMethodData = new DeviceMethodData(METHOD_THROWS,
                "Method Throws " + methodName);
            }
            return deviceMethodData;
        }
    }
}
```

APÊNDICE C – Código Node-RED do painel de visualização

```
[
    {
        "id": "dfa65983.f7f828",
        "type": "tab",
        "label": "Servidor Dinämico IoT",
        "disabled": false,
        "info": "",
        "env": []
    },
    {
        "id": "a37932fe.8cb1c",
        "type": "http in",
        "z": "dfa65983.f7f828",
        "name": "Recebendo via HTTP",
        "url": "/server",
        "method": "post",
        "upload": false,
        "swaggerDoc": "",
        "x": 120,
        "y": 60,
        "wires": [
            [
                "f2b9370b.5bb558",
                "21232dc5.cb5e82"
            ]
        ]
    },
    {
        "id": "ea5d1f71.57a78",
        "type": "change",
        "z": "dfa65983.f7f828",
        "name": "Estado",
        "rules": [
```

```
        {
            "t": "set",
            "p": "payload",
            "pt": "msg",
            "to": "payload.estado",
            "tot": "msg"
        }
    ],
    "action": "",
    "property": "",
    "from": "",
    "to": "",
    "reg": false,
    "x": 390,
    "y": 120,
    "wires": [
        [
            "5788c8ea.d5a698"
        ]
    ]
},
{
    "id": "364ccb0.53a4636",
    "type": "http response",
    "z": "dfa65983.f7f828",
    "name": "Resposta HTTP",
    "statusCode": "",
    "headers": {},
    "x": 640,
    "y": 60,
    "wires": []
},
{
    "id": "f2b9370b.5bb558",
    "type": "change",
    "z": "dfa65983.f7f828",
    "name": "Retorna OK",
    "rules": [
        {
```

```
                "t": "set",
                "p": "payload",
                "pt": "msg",
                "to": "OK",
                "tot": "str"
            }
        ],
        "action": "",
        "property": "",
        "from": "",
        "to": "",
        "reg": false,
        "x": 390,
        "y": 60,
        "wires": [
            [
                "364ccb0.53a4636"
            ]
        ]
    },
    {
        "id": "5788c8ea.d5a698",
        "type": "ui_text",
        "z": "dfa65983.f7f828",
        "group": "d84cefce.261db",
        "order": 1,
        "width": 0,
        "height": 0,
        "name": "",
        "label": "Estado do servidor",
        "format": "{{msg.payload}}",
        "layout": "row-spread",
        "className": "",
        "x": 610,
        "y": 120,
        "wires": []
    },
    {
        "id": "3eed7cd5.af44d4",
```

```
        "type": "ui_text",
        "z": "dfa65983.f7f828",
        "group": "d84cefce.261db",
        "order": 3,
        "width": 0,
        "height": 0,
        "name": "",
        "label": "Última mensagem",
        "format": "{{msg.payload}}",
        "layout": "row-spread",
        "x": 610,
        "y": 160,
        "wires": []
    },
    {
        "id": "5986fca2.3aab34",
        "type": "change",
        "z": "dfa65983.f7f828",
        "name": "Mensagem",
        "rules": [
            {
                "t": "set",
                "p": "payload",
                "pt": "msg",
                "to": "payload.mensagem",
                "tot": "msg"
            }
        ],
        "action": "",
        "property": "",
        "from": "",
        "to": "",
        "reg": false,
        "x": 410,
        "y": 160,
        "wires": [
            [
                "3eed7cd5.af44d4"
            ]
```

```
        ]
    },
    {
        "id": "21232dc5.cb5e82",
        "type": "switch",
        "z": "dfa65983.f7f828",
        "name": "Distribui as informações",
        "property": "payload",
        "propertyType": "msg",
        "rules": [
            {
                "t": "hask",
                "v": "estado",
                "vt": "str"
            },
            {
                "t": "hask",
                "v": "mensagem",
                "vt": "str"
            },
            {
                "t": "hask",
                "v": "dispositivo",
                "vt": "str"
            },
            {
                "t": "hask",
                "v": "quantidade",
                "vt": "str"
            }
        ],
        "checkall": "true",
        "repair": false,
        "outputs": 4,
        "x": 170,
        "y": 180,
        "wires": [
            [
                "ea5d1f71.57a78"
```

```
            ],
            [
                "5986fca2.3aab34"
            ],
            [
                "d0d522a3.a62a8"
            ],
            [
                "765f05db.2574dc"
            ]
        ]
    },
    {
        "id": "eff8c812.bbb9c8",
        "type": "ui_text",
        "z": "dfa65983.f7f828",
        "group": "d84cefce.261db",
        "order": 3,
        "width": 0,
        "height": 0,
        "name": "",
        "label": "Dispositivo",
        "format": "{{msg.payload}}",
        "layout": "row-spread",
        "className": "",
        "x": 590,
        "y": 200,
        "wires": []
    },
    {
        "id": "d0d522a3.a62a8",
        "type": "change",
        "z": "dfa65983.f7f828",
        "name": "Dispositivo",
        "rules": [
            {
                "t": "set",
                "p": "payload",
                "pt": "msg",
```

```
                "to": "payload.dispositivo",
                "tot": "msg"
            }
        ],
        "action": "",
        "property": "",
        "from": "",
        "to": "",
        "reg": false,
        "x": 410,
        "y": 200,
        "wires": [
            [
                "eff8c812.bbb9c8"
            ]
        ]
    },
    {
        "id": "765f05db.2574dc",
        "type": "change",
        "z": "dfa65983.f7f828",
        "name": "Quantidade",
        "rules": [
            {
                "t": "set",
                "p": "payload",
                "pt": "msg",
                "to": "payload.quantidade",
                "tot": "msg"
            }
        ],
        "action": "",
        "property": "",
        "from": "",
        "to": "",
        "reg": false,
        "x": 410,
        "y": 240,
        "wires": [
```

```
            [
                "20e8fb08.3854c4"
            ]
        ]
    },
    {
        "id": "20e8fb08.3854c4",
        "type": "ui_gauge",
        "z": "dfa65983.f7f828",
        "name": "",
        "group": "d84cefce.261db",
        "order": 4,
        "width": 0,
        "height": 0,
        "gtype": "gage",
        "title": "Quantidade de mensagens",
        "label": "msgs",
        "format": "{{value}}",
        "min": 0,
        "max": "100",
        "colors": [
            "#00b500",
            "#e6e600",
            "#ca3838"
        ],
        "seg1": "",
        "seg2": "",
        "x": 640,
        "y": 240,
        "wires": []
    },
    {
        "id": "d84cefce.261db",
        "type": "ui_group",
        "name": "Default",
        "tab": "980602f5.8e079",
        "order": 1,
        "disp": false,
        "width": "14",
```

```
        "collapse": false,
        "className": ""
    },
    {
        "id": "980602f5.8e079",
        "type": "ui_tab",
        "name": "Home",
        "icon": "dashboard",
        "disabled": false,
        "hidden": true
    }
]
```

APÊNDICE D – Código em Python da experimentação no dispositivo Nvidia Jetson

```
# Copyright (c) Microsoft. All rights reserved.
# Licensed under the MIT license. See LICENSE file in the project
# root for full license information.

import time
import os
import logging
from azure.iot.device import IoTHubDeviceClient, Message, MethodResponse
from pyparsing import empty

CONNECTION_STRING = 'HostName=RedeHibridaIoT.azure-devices.net;
DeviceId=Teste1;
SharedAccessKey=QWGWWSZJ+gStxB7TLBqVyWyrv02yYadCRsMYizewDjE='

# Define the JSON message to send to IoT Hub.
MSG_TXT = '{{"temperature": {temperature}, "humidity": 50, "battery": 100}}'

INTERVAL = 5

def create_client():
    # Create an IoT Hub client

    model_id = "dtmi:com:example:NonExistingController;1"

    client = IoTHubDeviceClient.create_from_connection_string(
                CONNECTION_STRING,
                product_info=model_id,
                websockets=True)
                # used for communication over websockets (port 443)

    # *** Device Twin ***
    #
    # define behavior for receiving a twin patch
```

```
    # NOTE: this could be a function or a coroutine

    def twin_patch_handler(twin):
        global INTERVAL
        INTERVAL = twin['telemetryConfig']['interval'] // 1000

    try:

        # Attach the Device Twin Desired properties change request handler
        client.on_twin_desired_properties_patch_received = twin_patch_handler

        client.connect()

    except:
        # Clean up in the event of failure
        client.shutdown()
        raise

    return client


def run_telemetry_sample(client):
    # This sample will send temperature telemetry every second
    print("IoT Hub device sending periodic messages")

    client.connect()

    while True:
        # *** Sending a message ***
        #
        # Build the message with simulated telemetry values.
        temperature = os.popen('cat /sys/devices/virtual/thermal/
        thermal_zone0/temp').read()[0:2]
        msg_txt_formatted = MSG_TXT.format(temperature=temperature)
        message = Message(msg_txt_formatted)

        message.content_encoding = "utf-8"
        message.content_type = "application/json"
```

```
        # Send the message.
        print(f"Sending message: {message}")
        client.send_message(message)

        print(f"Next message in: {INTERVAL} segundos")
        time.sleep(INTERVAL)


def main():
    print ("IoT Hub Quickstart #2 - Simulated device")
    print ("Press Ctrl-C to exit")

    # Instantiate the client. Use the same instance of the client
    for the duration of
    # your application
    client = create_client()

    # Send telemetry
    try:
        run_telemetry_sample(client)
    except KeyboardInterrupt:
        print("IoTHubClient sample stopped by user")
    finally:
        print("Shutting down IoTHubClient")
        client.shutdown()


if __name__ == '__main__':
    main()
```

APÊNDICE E – Teste Wireshark

O teste do analisador de pacotes de rede foi feito através de um PING no prompt de comando do Windows.

```
C:\Users\miesp>ping www.google.com.br -t

Disparando www.google.com.br [142.251.132.227] com 32 bytes de dados:
Resposta de 142.251.132.227: bytes=32 tempo=6ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=4ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=6ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=5ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=5ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=6ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=5ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=9ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=6ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=6ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=6ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=8ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=8ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=9ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=6ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=5ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=6ms TTL=61
```

```
Resposta de 142.251.132.227: bytes=32 tempo=6ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=5ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=5ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=12ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=6ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=10ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=5ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=5ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=6ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=8ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=5ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=8ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=8ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=7ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=6ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=5ms TTL=61
Resposta de 142.251.132.227: bytes=32 tempo=9ms TTL=61

Estatísticas do Ping para 142.251.132.227:
    Pacotes: Enviados = 50, Recebidos = 50, Perdidos = 0 (0% de
             perda),
Aproximar um número redondo de vezes em milissegundos:
    Mínimo = 4ms, Máximo = 12ms, Média = 6ms
```

Abaixo o resultado da saída de dados do Wireshark:

N.º	Tempo (s)	Fonte	Destino	Protocolo
1	6.385969	192.168.1.6	142.251.132.227	ICMP
2	7.392322	192.168.1.6	142.251.132.227	ICMP
3	8.403252	192.168.1.6	142.251.132.227	ICMP
4	9.424583	192.168.1.6	142.251.132.227	ICMP

Continua na próxima página

Tabela 1 – Continuando

N.º	Tempo (s)	Fonte	Destino	Protocolo
5	10.429522	192.168.1.6	142.251.132.227	ICMP
6	11.438736	192.168.1.6	142.251.132.227	ICMP
7	12.445332	192.168.1.6	142.251.132.227	ICMP
8	13.465277	192.168.1.6	142.251.132.227	ICMP
9	14.482171	192.168.1.6	142.251.132.227	ICMP
10	15.493144	192.168.1.6	142.251.132.227	ICMP
11	16.496368	192.168.1.6	142.251.132.227	ICMP
12	17.509300	192.168.1.6	142.251.132.227	ICMP
13	18.522094	192.168.1.6	142.251.132.227	ICMP
14	19.535204	192.168.1.6	142.251.132.227	ICMP
15	20.542755	192.168.1.6	142.251.132.227	ICMP
16	21.546061	192.168.1.6	142.251.132.227	ICMP
17	22.564785	192.168.1.6	142.251.132.227	ICMP
18	23.574981	192.168.1.6	142.251.132.227	ICMP
19	24.593866	192.168.1.6	142.251.132.227	ICMP
20	25.610403	192.168.1.6	142.251.132.227	ICMP
21	26.616755	192.168.1.6	142.251.132.227	ICMP
22	27.624306	192.168.1.6	142.251.132.227	ICMP
23	28.644881	192.168.1.6	142.251.132.227	ICMP
24	29.662340	192.168.1.6	142.251.132.227	ICMP
25	30.680271	192.168.1.6	142.251.132.227	ICMP
26	31.685617	192.168.1.6	142.251.132.227	ICMP
27	32.690821	192.168.1.6	142.251.132.227	ICMP
28	33.709192	192.168.1.6	142.251.132.227	ICMP
29	34.714590	192.168.1.6	142.251.132.227	ICMP
30	35.718767	192.168.1.6	142.251.132.227	ICMP
31	36.738713	192.168.1.6	142.251.132.227	ICMP
32	37.743789	192.168.1.6	142.251.132.227	ICMP
33	38.758001	192.168.1.6	142.251.132.227	ICMP
34	39.760982	192.168.1.6	142.251.132.227	ICMP
35	40.764874	192.168.1.6	142.251.132.227	ICMP
36	41.769349	192.168.1.6	142.251.132.227	ICMP
37	42.774265	192.168.1.6	142.251.132.227	ICMP
38	43.778623	192.168.1.6	142.251.132.227	ICMP
39	44.780917	192.168.1.6	142.251.132.227	ICMP
40	45.786098	192.168.1.6	142.251.132.227	ICMP

Continua na próxima página

Tabela 1 – Continuando

N.º	Tempo (s)	Fonte	Destino	Protocolo
41	46.803819	192.168.1.6	142.251.132.227	ICMP
42	47.819367	192.168.1.6	142.251.132.227	ICMP
43	48.835782	192.168.1.6	142.251.132.227	ICMP
44	49.840647	192.168.1.6	142.251.132.227	ICMP
45	50.857590	192.168.1.6	142.251.132.227	ICMP
46	51.873399	192.168.1.6	142.251.132.227	ICMP
47	52.893416	192.168.1.6	142.251.132.227	ICMP
48	53.897609	192.168.1.6	142.251.132.227	ICMP
49	54.903503	192.168.1.6	142.251.132.227	ICMP
50	57.099974	192.168.1.6	142.251.132.227	ICMP

Referências

1 XIA, F. et al. Internet of things. *International Journal of Communication Systems*, v. 25, n. 9, p. 1101, 2012. 1

2 TARKOMA, S.; KATASONOV, A. Internet of things strategic research agenda. *Finnish Strategic Centre for Science, Technology and Innovation*, 2011. 1

3 WHITMORE, A.; AGARWAL, A.; XU, L. D. The internet of things—a survey of topics and trends. *Information systems frontiers*, Springer, v. 17, n. 2, p. 261–274, 2015. 1

4 PATIL, S. M.; VIJAYALASHMI, M.; TAPASKAR, R. Iot based solar energy monitoring system. In: IEEE. *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*. [S.l.], 2017. p. 1574–1579. 1

5 AKKAYA, K. et al. Iot-based occupancy monitoring techniques for energy-efficient smart buildings. In: IEEE. *2015 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*. [S.l.], 2015. p. 58–63. 1

6 DOBRILOVIĆ, D. Networking technologies for smart cities: An overview. *Interdisciplinary Description of Complex Systems: INDECS*, Hrvatsko interdisciplinarno društvo, v. 16, n. 3-A, p. 408–416, 2018. 1, 2

7 ZANELLA, A. et al. Internet of things for smart cities. *IEEE Internet of Things journal*, IEEE, v. 1, n. 1, p. 22–32, 2014. 1, 2

8 SHYAM, G. K.; MANVI, S. S.; BHARTI, P. Smart waste management using internet-of-things (iot). In: IEEE. *2017 2nd international conference on computing and communications technologies (ICCCT)*. [S.l.], 2017. p. 199–203. 1

9 GARCIA, G. M. et al. Tecnologias de redes para internet das coisas-iot. 1

10 DURAES, W.; FERREIRA, F. H. I. B.; MANZAN, R. *Arquitetura de soluções IoT: Desenvolva com Internet das Coisas para o mundo real*. [S.l.]: Casa do Código, 2022. 1

11 ROSA, L. d. S. P. et al. Aplicações do 5g em internet das coisas (iot). *INATEL, MINAS GERAIS, JUN*, 2017. 2

12 MARTINS, R. de S. Internet das coisas e blockchain: segurança e disponibilidade em redes de dispositivos conectados. 2

13 CHAUDHARI, B. S.; ZENNARO, M.; BORKAR, S. Lpwan technologies: Emerging application characteristics, requirements, and design considerations. *Future Internet*, Multidisciplinary Digital Publishing Institute, v. 12, n. 3, p. 46, 2020. 2

14 MATERNIA, M. et al. 5g ppp use cases and performance evaluation models. *5G PPP*, v. 1, 2016. 2

15 ELIJAH, O. et al. An overview of internet of things (iot) and data analytics in agriculture: Benefits and challenges. *IEEE Internet of Things Journal*, IEEE, v. 5, n. 5, p. 3758–3773, 2018. 2

16 BANAFA, A. Iot and blockchain convergence: benefits and challenges. *IEEE Internet of Things*, 2017. 2

17 TARDY, I. et al. Comparison of wireless techniques applied to environmental sensor monitoring. *SINTEF Rapport*, SINTEF, 2017. 2

18 RAZA, U.; KULKARNI, P.; SOORIYABANDARA, M. Low power wide area networks: An overview. *ieee communications surveys & tutorials*, IEEE, v. 19, n. 2, p. 855–873, 2017. 2, 3

19 ADELANTADO, F. et al. Understanding the limits of lorawan. *IEEE Communications magazine*, IEEE, v. 55, n. 9, p. 34–40, 2017. 2

20 RAZA, S. et al. Building the internet of things with bluetooth smart. *Ad Hoc Networks*, Elsevier, v. 57, p. 19–31, 2017. 2

21 RAJENDHAR, P.; KUMAR, P. P.; VENKATESH, R. Zigbee based wireless system for remote supervision and control of a substation. In: IEEE. *2017 International Conference on Innovative Research In Electrical Sciences (IICIRES)*. [S.l.], 2017. p. 1–4. 3

22 BARKER, P.; HAMMOUDEH, M. A survey on low power network protocols for the internet of things and wireless sensor networks. In: *Proceedings of the international conference on future networks and distributed systems*. [S.l.: s.n.], 2017. p. 1–8. 3

23 STILLER, B. et al. *Cisco Annual Internet Report (2018–2023) White Paper*. [S.l.], 2020. Disponível em: <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html>. 3

24 SHAH, T.; VENKATESAN, S. Authentication of iot device and iot server using secure vaults. In: *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. [S.l.: s.n.], 2018. p. 819–824. 3

25 GONÇALVES, M. T.; SANTOS, E. R. dos; SANTOS, M. A. B. dos. Otimização no envio de dados em dispositivos para internet das coisas (iot). *Jornada de Iniciação Científica e Extensão*, v. 15, n. 1, p. 144, 2020. 4, 5, 7

26 MADUREIRA, A. L.; ARAÚJO, F. R.; SAMPAIO, L. Um protocolo iot para redução de tráfego em redes de plano de dados programáveis. In: SBC. *Anais do XXXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. [S.l.], 2020. p. 826–839. 5, 7

27 MARCELINO, A. G. L. Otimização do consumo de bateria em hardware de internet das coisas industrial. *Universidade Federal de São Carlos*, Universidade Federal de São Carlos, 2021. 5, 7

28 SINGH, S. Optimize cloud computations using edge computing. In: IEEE. *2017 International Conference on Big Data, IoT and Data Science (BID)*. [S.l.], 2017. p. 49–53. 5, 6, 7

29 CAO, K. et al. An overview on edge computing research. *IEEE access*, IEEE, v. 8, p. 85714–85728, 2020. 5

30 AAZAM, M. et al. Cloud of things: Integrating internet of things and cloud computing and the issues involved. In: IEEE. *Proceedings of 2014 11th International Bhurban Conference on Applied Sciences & Technology (IBCAST) Islamabad, Pakistan, 14th-18th January, 2014*. [S.l.], 2014. p. 414–419. 5

31 SRINIDHI, N.; KUMAR, S. D.; VENUGOPAL, K. Network optimizations in the internet of things: A review. *Engineering Science and Technology, an International Journal*, Elsevier, v. 22, n. 1, p. 1–21, 2019. 6, 7

32 QIAN, L. et al. Cloud computing: An overview. In: SPRINGER. *IEEE international conference on cloud computing*. [S.l.], 2009. p. 626–631. 7

33 CHEN, S.; ZHANG, T.; SHI, W. Fog computing. *IEEE Internet Computing*, v. 21, n. 2, p. 4–6, 2017. 7

34 ALSHAREEF, H. et al. Towards an effective management of iot by integrating cloud and fog computing. In: IEEE. *2019 IEEE International Conference on Smart Internet of Things (SmartIoT)*. [S.l.], 2019. p. 197–204. 7

35 HAAG, S.; ANDERL, R. Digital twin–proof of concept. *Manufacturing Letters*, Elsevier, v. 15, p. 64–66, 2018. 9

36 GLAESSGEN, E.; STARGEL, D. The digital twin paradigm for future nasa and us air force vehicles. In: *53rd AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics and materials conference 20th AIAA/ASME/AHS adaptive structures conference 14th AIAA*. [S.l.: s.n.], 2012. p. 1818. 9

37 CHEN, Y. Integrated and intelligent manufacturing: perspectives and enablers. *Engineering*, Elsevier, v. 3, n. 5, p. 588–595, 2017. 9

38 LIU, Z.; MEYENDORF, N.; MRAD, N. The role of data fusion in predictive maintenance using digital twin. In: AIP PUBLISHING LLC. *AIP Conference Proceedings*. [S.l.], 2018. v. 1949, p. 020023. 9

39 ZHENG, Y.; YANG, S.; CHENG, H. An application framework of digital twin and its case study. *Journal of Ambient Intelligence and Humanized Computing*, Springer, v. 10, n. 3, p. 1141–1153, 2019. 9

40 ERKOYUNCU, J. A. et al. Digital twins: Understanding the added value of integrated models for through-life engineering services. *Procedia Manufacturing*, Elsevier, v. 16, p. 139–146, 2018. 9

41 BRAY, T. et al. The javascript object notation (json) data interchange format. *RFC 7159, DOI 10.17487/RFC7159, March 2014,< http://www. rfc-editor. org . . .* , RFC 7159, DOI 10.17487/RFC7159, March 2014,< http://www. rfc-editor. org . . . , 2014. 10

42 CROCKFORD, D. *The application/json media type for javascript object notation (json)*. [S.l.], 2006. 10

43 MARRS, T. *JSON at work: practical data integration for the web*. [S.l.]: "O'Reilly Media, Inc.", 2017. 10

44 PEZOA, F. et al. Foundations of json schema. In: *Proceedings of the 25th International Conference on World Wide Web*. [S.l.: s.n.], 2016. p. 263–273. 10, 12

45 COPELAND, M. et al. Microsoft azure. *New York, NY, USA:: Apress*, Springer, 2015. 11

46 COLLIER, M.; SHAHAN, R. *Microsoft Azure Essentials-Fundamentals of Azure*. [S.l.]: Microsoft Press, 2015. 11

47 STACKOWIAK, R. Azure iot hub. In: *Azure Internet of Things Revealed*. [S.l.]: Springer, 2019. p. 73–85. 12

48 HUSÁK, M. et al. Network-based https client identification using ssl/tls fingerprinting. In: IEEE. *2015 10th international conference on availability, reliability and security*. [S.l.], 2015. p. 389–396. 12

49 VINOSKI, S. Advanced message queuing protocol. *IEEE Internet Computing*, IEEE, v. 10, n. 6, p. 87–89, 2006. 12

50 SONI, D.; MAKWANA, A. A survey on mqtt: a protocol of internet of things (iot). In: *International Conference On Telecommunication, Power Analysis And Computing Techniques (ICTPACT-2017)*. [S.l.: s.n.], 2017. v. 20. 12

51 FETTE, I.; MELNIKOV, A. *The websocket protocol*. [S.l.]: RFC 6455, December, 2011. 12

52 PYTHON, I. Python. *Citeseer*, Citeseer, 2019. 15

53 LUTZ, M. *Programming python*. [S.l.]: "O'Reilly Media, Inc.", 2001. 15, 16

54 LECHETA, R. R. *Google Android-3ª Edição: Aprenda a criar aplicações para dispositivos móveis com o Android SDK*. [S.l.]: Novatec Editora, 2013. 16

55 PEREIRA, L. C. O.; SILVA, M. L. da. *Android para desenvolvedores*. [S.l.]: Brasport, 2009. 16, 17

56 DIMARZIO, J. *Beginning Android Programming with Android Studio*. [S.l.]: John Wiley & Sons, 2016. 17, 18

57 MORRISON, J. P. Flow-based programming. In: *Proc. 1st International Workshop on Software Engineering for Parallel and Distributed Systems*. [S.l.: s.n.], 1994. p. 25–29. 19, 20

58 HAGINO, T.; O'LEARY, N. *Practical Node-RED Programming: Learn powerful visual programming techniques and best practices for the web and IoT*. Packt Publishing, 2021. ISBN 9781800207660. Disponível em: <https://books.google.com.br/books?id=nlwlEAAAQBAJ>. 20

59 LAMPING, U.; WARNICKE, E. Wireshark user's guide. *Interface*, v. 4, n. 6, p. 1, 2004. 21

60 BAXTER, J. H. *Wireshark essentials*. [S.l.]: Packt Publishing Ltd, 2014. 21

61 NATH, A. *Packet Analysis with Wireshark*. [S.l.]: Packt Publishing Ltd, 2015. 21

62 SOTTILE, M. J.; MINNICH, R. G. Supermon: A high-speed cluster monitoring system. In: IEEE. *Proceedings. IEEE International Conference on Cluster Computing*. [S.l.], 2002. p. 39–46. 22

63 FRANKLIN, D. *Jetson Nano Brings AI Computing to Everyone*. 2019. <https://developer.nvidia.com/blog/jetson-nano-ai-computing/>. 60

64 NVIDIA. *JetPack SDK*. 2023. <https://developer.nvidia.com/embedded/jetpack>. 61