Avaliação da Integração do Protocolo MQTT em uma Plataforma de Cidades Inteligentes*

Bruno Carneiro da Cunha, Daniel Macêdo Batista

[1]Departamento de Ciência da Computação – Universidade de São Paulo (USP)

brunocarneirodacunha@usp.br, batista@ime.usp.br

Abstract. *The number of applications based on the Message Queuing Telemetry Transport (MQTT) protocol has increased greatly, making it the world's most popular pub/sub protocol. This paper presents the development and evaluation of an MQTT adaptor for the InterSCity platform, an integrated open-source software for the development of robust smart cities and Internet of Things applications. The adaptor allows sensors and actuators to publish data to and receive commands from InterSCity using the MQTT protocol. Performance evaluation has shown that using the MQTT adaptor could be more efficient than using the standard HTTP API. For example, with the use of MQTT, the observed throughput was up to 84.27 times higher than with the use of HTTP.*

Resumo. *O número de aplicativos baseados no protocolo Message Queuing Telemetry Transport (MQTT) tem aumentado bastante, fazendo com que esse seja o protocolo pub/sub mais popular do mundo. Este artigo apresenta o desenvolvimento e a avaliação de um adaptador MQTT para a plataforma de cidades inteligentes InterSCity, uma plataforma de código aberto para aplicações robustas de cidades inteligentes e Internet das Coisas. O adaptador permite que sensores e atuadores interajam com a plataforma usando o protocolo MQTT. A avaliação de desempenho do adaptador sugere que o uso do MQTT pode ser mais eficiente que o uso da API HTTP. Por exemplo, com o uso do MQTT, foi observada uma vazão até 84,27 vezes maior do que com o uso do HTTP.*

1. Introdução

Nos últimos anos, o número de aplicativos baseados em *Message Queuing Telemetry Transport* (MQTT) [6] aumentou bastante, fazendo com que esse seja o protocolo pub/sub[1] mais popular do mundo, com utilização em uma ampla variedade de domínios, como os de saúde, automação residencial, transportes inteligentes e distribuição de energia.

Uma das motivações para a utilização do MQTT, principalmente em cenários de cidades inteligentes e Internet das Coisas, é o fato do mesmo ser considerado um protocolo leve [8], no sentido de ser mais eficiente na transmissão de dados e de exigir poucos recursos locais dos nós envolvidos na comunicação. Essas duas características são muito

*Esta pesquisa está inserida no subprojeto **MISIoT: Mechanisms to Improve Security in IoT platforms**, do Projeto InterSCity (`https://interscity.org/`).

[1]*Publish/Subscribe*: modelo de troca de mensagens no qual um *publisher* confia a um *broker* o envio da mensagem aos *subscribers* de um tópico.

importantes em ambientes com dispositivos conectados de forma intermitente à rede e alimentados por baterias.

Plataformas de cidades inteligentes, como a InterSCity [4, 1], têm sido desenvolvidas nos últimos anos para fornecer uma infraestrutura que facilite o desenvolvimento de aplicações para melhorar a vida dos cidadãos. Essas plataformas devem ser capazes de permitir a comunicação de sensores e de atuadores dos mais diversos tipos. Uma forma de ampliar o suporte a diferentes dispositivos é com a utilização de protocolos padronizados já implementados nos mesmos. Dentre esses protocolos, destaca-se o MQTT.

Este artigo apresenta o desenvolvimento e a avaliação de desempenho de um adaptador MQTT para a plataforma InterSCity. Os resultados obtidos na avaliação de desempenho sugerem que o uso do MQTT pode ser mais eficiente que o uso da API HTTP padrão da plataforma. Por exemplo, com o uso do MQTT, foi observada uma vazão até 84,27 vezes maior do que com o uso do HTTP.

O restante deste artigo está organizado da seguinte forma: a Seção 2 descreve os trabalhos relacionados. A Seção 3 apresenta detalhes da implementação do adaptador MQTT. A Seção 4 descreve os experimentos que foram realizados para a análise de desempenho do adaptador para compará-lo com a API HTTP. A Seção 5 detalha o ambiente de experimentação e analisa os resultados obtidos. A Seção 6 apresenta as conclusões e os trabalhos futuros.

2. Trabalhos relacionados

Stephen Nicolas fez uma comparação entre os protocolos HTTPS e MQTT voltada principalmente para analisar o consumo de bateria em dispositivos Android [7]. Foi comparada a vazão entre os dois protocolos e chegou-se a um resultado que reforça aqueles apresentados por nós na subseção 4.1: o MQTT é mais eficiente do que protocolos de propósito geral, como o HTTPS.

Rafael Dias da Costa, em sua monografia de conclusão de curso [3], focou na melhoria de uma parte da plataforma InterSCity, e também desenvolveu um adaptador MQTT, embora esse só funcione para enviar comandos a dispositivos atuadores. O adaptador desenvolvido por nós avança o trabalho realizado em [3] por permitir a comunicação tanto com sensores quanto com atuadores.

João Cardoso et al. apresentaram diversas métricas para analisar o desempenho de plataformas similares à InterSCity [2]. De especial utilidade para nós foi a seção que descreve as métricas quantitativas usadas naquele estudo. As informações foram úteis ao planejarmos os nossos experimentos.

3. Integração do protocolo

A InterSCity é uma plataforma baseada no *framework* Ruby on Rails, e com arquitetura em microsserviços, ou seja, ela é formada por vários módulos que se comunicam para fornecer os serviços. Os cinco módulos são: *Actuator Controller*, *Data Collector*, *Resource Adaptor*, *Resource Cataloguer* e *Resource Discoverer*. Todos esses módulos comunicam-se entre si pelo *broker* AMQP `RabbitMQ`, conforme ilustrado na Figura 1, que apresenta a arquitetura da InterSCity e resume cada um dos módulos.

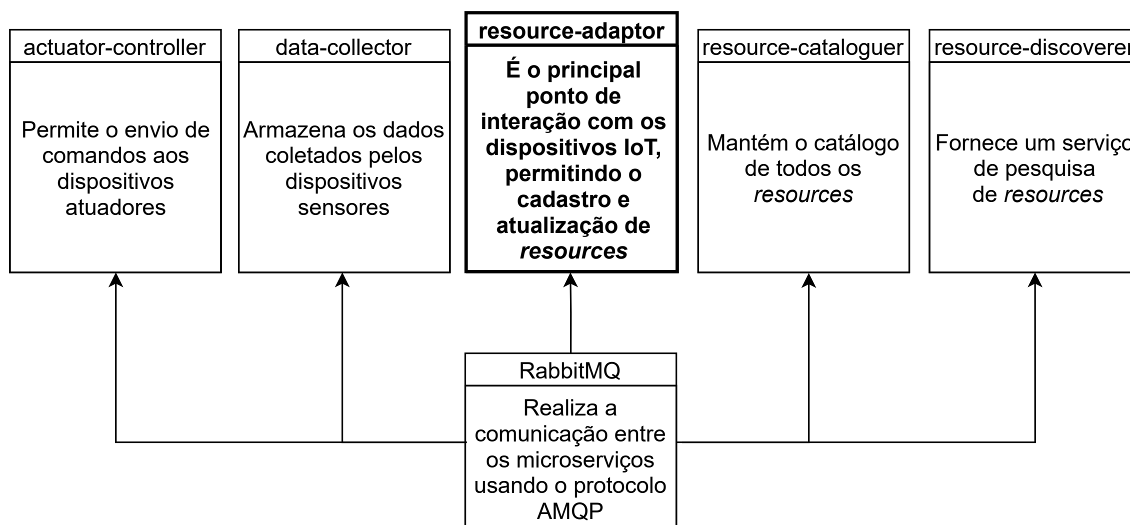


Figura 1. A arquitetura da InterSCity e descrição de seus módulos.

O único microsserviço modificado para a integração do MQTT foi o *Resource Adaptor*, que está em destaque na Figura 1. Foram adicionados dois novos *workers* a esse microsserviço: o primeiro *worker* foi chamado de **`MQTTSubscriber`**. Na inicialização, uma instância dessa classe se conecta ao servidor MQTT fornecido, e assina o tópico `resources/#`. Qualquer atualização em qualquer tópico sob `resources` é repassada à instância, e processada de maneira semelhante à API HTTP.

Para o envio de comandos via MQTT da plataforma aos atuadores, a classe `ActuatorCommandNotifier` foi modificada para instanciar uma nova classe, de nome **`MQTTPublisher`**. Este *worker* recebe todos os comandos enviados aos dispositivos atuadores na plataforma, e atualiza os respectivos tópicos `commands/{uuid}` no servidor MQTT.

Além das duas novas classes, foi necessário executar também um *broker* MQTT, para que esse cuidasse da troca de mensagens entre os clientes publicantes e os assinantes. Escolheu-se o **`Mosquitto`**, a implementação mais bem estabelecida e documentada de *broker* MQTT.

4. Descrição dos experimentos

Os experimentos descritos a seguir procuraram avaliar o desempenho da plataforma InterSCity com o adaptador MQTT, composto pelas novas classes `MQTTPublisher` e `MQTTSubscriber`. Os experimentos também avaliaram diversas métricas com o uso da API HTTP padrão da plataforma, a fim de comparar ambas as opções de transmissão de dados.

4.1. Vazão

No primeiro experimento foi avaliada a capacidade da plataforma de receber um grande fluxo de dados. Uma simulação com um número variável de *threads* clientes foi criada, replicando o comportamento de dispositivos sensores tentando enviar dados à plataforma simultaneamente. Ao término do experimento, a taxa média de mensagens por

segundo foi calculada, repetindo-se as medições 30 vezes para cada número de *threads* a fim de colher um resultado estatisticamente mais representativo. Cada mensagem enviada pelos dispositivos simulados continha um valor simulado de temperatura e um *timestamp*, formatados em JSON no padrão esperado pela plataforma. É importante frisar que **a vazão foi calculada no cliente**, ou seja, ela representa o número de mensagens por segundo que os clientes conseguiram enviar à plataforma com sucesso.

4.2. Latência

No segundo experimento foi avaliado o intervalo de tempo entre o envio de um comando a um dispositivo atuador na plataforma, e o seu recebimento por esse atuador. O experimento foi repetido 3.000 vezes para cada um dos dois protocolos (HTTP e MQTT). Cada comando enviado continha um valor para o atributo *illuminate* e um *timestamp*, formatados em JSON.

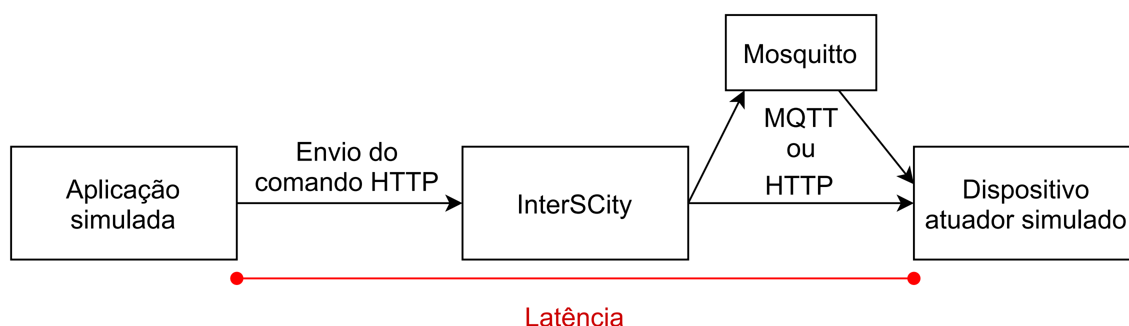


Figura 2. Medição do tempo para envio de comando a um atuador.

4.3. Consumo de CPU e memória

Para o último experimento foi repetido o experimento de vazão (Subseção 4.1) com 40 *threads*, mas com uma métrica diferente: foi medido o consumo de CPU e de memória por cada um dos microsserviços na plataforma, pelo servidor do `RabbitMQ` e pelo broker MQTT `Mosquitto`.

Os processos mencionados foram monitorados ao longo de um minuto e cada sensor simulado fez o envio de 1.000 valores. Antes do começo do envio dos valores, há um intervalo de aproximadamente dez segundos para o cadastro dos *resources* na plataforma.

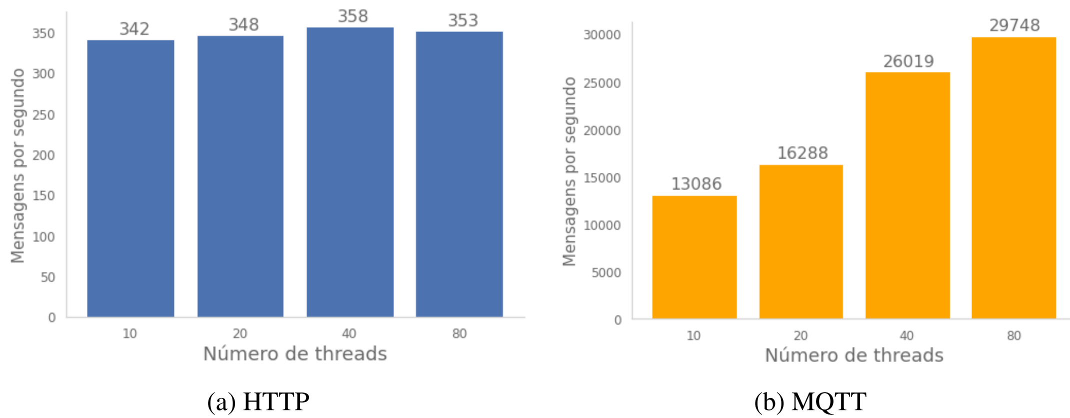
5. Análise dos resultados

Os experimentos foram executados em uma máquina *Intel i5-8400* 2,8GHz de 6 núcleos, com 16GB de memória DDR4, de 2666MHz. O sistema operacional foi o `elementaryOS 5.1 Hera`, baseado no `Ubuntu 18.04 LTS`. O servidor MQTT usado foi o `Mosquitto v.31`, e o servidor HTTP foi o `Puma 3.12.1`, usado pela InterSCity.

Para garantir que não houvessem outros processos fazendo uso intensivo dos recursos do ambiente a ponto de invalidar os resultados, os processos criados pelo sistema operacional foram listados e ordenados pelo maior consumo de CPU. Pelas informações obtidas nessa listagem, não foi observado nenhum processo competindo pelos recursos em um nível que invalidasse os resultados obtidos.

Os clientes se comunicaram com a plataforma através da interface de *loopback*, com largura de banda de 52,6 Gbits/s medida pela ferramenta `iperf`, de maneira que podemos excluir da interpretação qualquer variação significativa na qualidade do enlace.

5.1. Vazão



(a) HTTP (b) MQTT

Figura 3. Vazão média

Esse primeiro resultado mostrou uma acentuada diferença de desempenho entre os dois protocolos (Figura 3). Usando o adaptador MQTT desenvolvido, foi possível ter uma vazão no mínimo 38,26 vezes maior do que o HTTP, com 10 *threads*, e no máximo 84,27 vezes maior, com 80 *threads*.

Notou-se também que o aumento do número de *threads*, simulando um maior número de sensores enviando dados, não aumentou a vazão do HTTP, enquanto no MQTT o aumento de *threads* causou um incremento significativo na vazão. Isso deve-se principalmente ao fato do servidor HTTP `Puma` estar rodando na implementação de Ruby conhecida como `MRI`, escolhida pelos mantenedores da plataforma InterSCity, que por padrão não possui *multithreading* real [5].

5.2. Latência

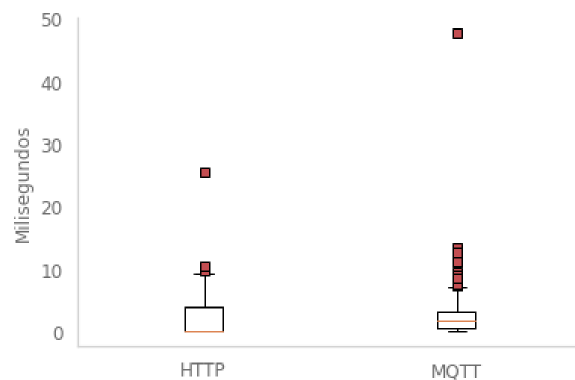


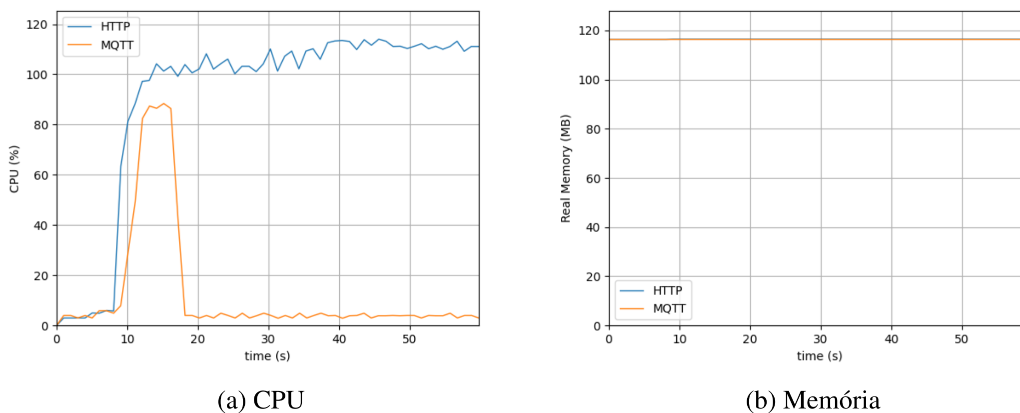
Figura 4. Latência

Pelos resultados obtidos no segundo experimento (Figura 4), não é possível dizer que um protocolo demonstrou uma clara vantagem sobre o outro, mas sim que os resultados foram bastante equivalentes. A média de latência usando o HTTP ficou em 1,671ms, enquanto a do MQTT ficou em 1,786ms.

5.3. Consumo de CPU e memória

Os microsserviços *Actuator Controller*, *Data Collector*, *Resource Cataloguer* e *Resource Discoverer* não apresentaram nenhuma diferença significativa de uso de CPU ou memória entre os dois protocolos, portanto foram omitidos desta seção.

5.3.1. Resource Adaptor

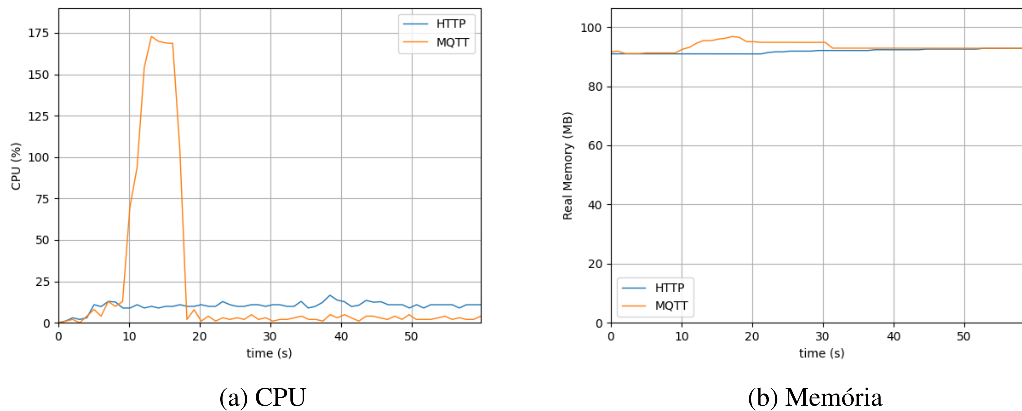


(a) CPU (b) Memória

Figura 5. Consumo de CPU e memória – Resource Adaptor

O uso de memória foi idêntico usando os dois protocolos, mas o uso de CPU foi notadamente diferente (Figura 5). O adaptador MQTT integrado ao *Resource Adaptor* processou as 40.000 mensagens antes dos 20s, usando em torno de 85% de CPU, enquanto o servidor HTTP não conseguiu nem mesmo receber as 40.000 mensagens antes do fim do experimento, apesar de ter consumido mais processamento que o adaptador MQTT.

5.3.2. RabbitMQ – Servidor AMQP

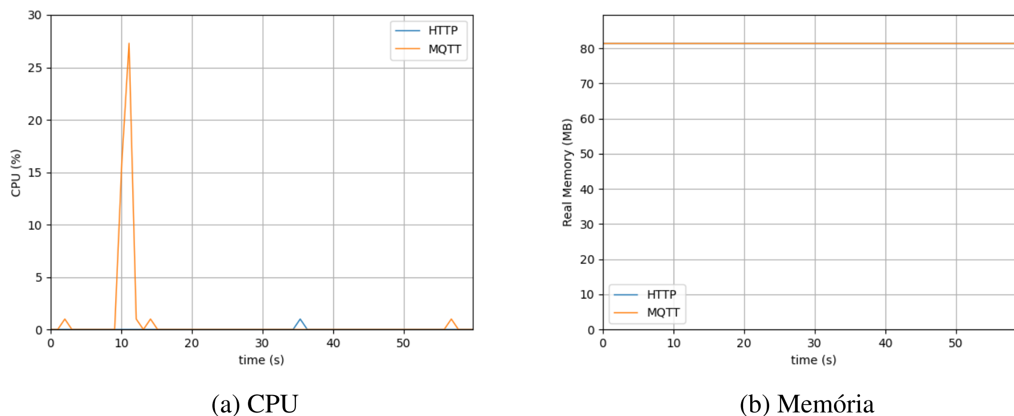


(a) CPU (b) Memória

Figura 6. Consumo de CPU e memória – RabbitMQ

A maior vazão da plataforma usando o MQTT se reflete no maior uso de CPU pelo RabbitMQ durante um curto período de tempo (Figura 6). As mensagens são mais rapidamente processadas pelo *Resource Adaptor* usando o adaptador MQTT, e por isso são mais rapidamente enfileiradas no servidor AMQP para o consumo do *Data Collector*.

5.3.3. Mosquitto – Servidor MQTT



(a) CPU (b) Memória

Figura 7. Consumo de CPU e memória – Mosquitto

O Mosquitto, desenvolvido em C, é eficiente no processamento dos valores enviados pelos dispositivos simulados. Os dados foram recebidos e encaminhados em menos de 5 segundos (a partir de aproximadamente 10s quando começam a ser enviados pelos sensores), com um uso razoavelmente pequeno de CPU, inferior a 30% (Figura 7). Isso indica que a execução do Mosquitto na própria máquina da InterSCity impactaria muito pouco no desempenho da plataforma.

6. Conclusão

Com base nos resultados, pode-se afirmar que o uso do adaptador MQTT desenvolvido neste trabalho para a interação de dispositivos sensores e atuadores com a plataforma InterSCity é recomendado. A maior capacidade de vazão usando o MQTT é benéfica tanto para os dispositivos conectados à plataforma, que podem economizar energia e processamento, quanto para os mantenedores de uma instância da InterSCity, que podem alocar uma máquina de menor custo para atender um mesmo número de dispositivos. Também é possível constatar que aplicações sensíveis à latência podem usar o MQTT sem nenhum prejuízo, pois a adição do *broker* como intermediário na conexão não acarretou em acréscimo desproporcional no tempo total para envio de um comando. Como trabalho futuro, pretende-se estudar o gargalo de desempenho apresentado pelo servidor HTTP.

Agradecimentos

Esta pesquisa é parte do INCT da Internet do Futuro para Cidades Inteligentes, financiado por CNPq (proc. 465446/2014-0), Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001 e FAPESP (procs. 14/50937-1 e 15/24485-9).

Bibliografia

[1] Daniel Macêdo Batista et al. "InterSCity: Addressing Future Internet research challenges for Smart Cities". In: *Anais da 7th International Conference on the Network of the Future (NOF)*. 2016, pp. 1–6.

[2] João Cardoso et al. "Benchmarking IoT Middleware Platforms". In: *Anais do IEEE 18th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. 2017, pp. 1–7.

[3] Rafael Dias da Costa. *Proposta de Arquitetura para Controle de Atuadores em Cidades Inteligentes: Aplicação na Plataforma InterSCity*. Monografia de conclusão de curso. `https://bdm.unb.br/bitstream/10483/25337/1/2018_RafaelDiasDaCosta_tcc.pdf`. Acessado em 28 de Julho de 2021. 2018.

[4] Arthur de M. Del Esposte et al. "Design and Evaluation of a Scalable Smart City Software Platform with Large-Scale Simulations". In: *Future Generation Computer Systems* 93 (2019), pp. 427–441. ISSN: 0167-739X.

[5] Richard Ludvigh et al. "Ruby benchmark tool using docker". In: *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 2015, pp. 947–952. DOI: `10.15439/2015F99`.

[6] *MQTT Version 3.1.1*. `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf`. Acessado em 25 de Junho de 2021. Oasis, 2014.

[7] Stephen Nicolas. *Power Profiling: HTTPS Long Polling vs. MQTT with SSL, on Android*. `http://stephendnicholas.com/posts/power-profiling-mqtt-vs-https`. Acessado em 27 de Julho de 2021. 2012.

[8] Rafik Zitouni et al. "IoT-Based Urban Traffic-Light Control: Modelling, Prototyping and Evaluation of MQTT Protocol". In: *Anais da 12th IEEE International Conference on Cyber, Physical and Social Computing (CPSCom)*. 2019, pp. 182–189.