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

Proposta de Arquitetura para Controle de Atuadores em Cidades Inteligentes: Aplicação na Plataforma InterSCity

Rafael Dias da Costa

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Wilson Henrique Veneziano

Coorientador
Prof.a Dr.a Carla Silva Rocha Aguiar

Brasília
2018

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

Proposta de Arquitetura para Controle de Atuadores em Cidades Inteligentes: Aplicação na Plataforma InterSCity

Rafael Dias da Costa

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Wilson Henrique Veneziano (Orientador)
CIC/UnB

Prof. Dr. Edison Ishikawa
CIC/UnB

Prof. Dr. Renato Coral Sampaio
FGA/UnB

Prof. Dr. José Edil Guimarães de Medeiros
Coordenador do Curso de Engenharia da Computação

Brasília, 13 de dezembro de 2018

Dedicatória

Dedico este trabalho ao meu avô, Weldas, que por toda a sua vida fez o possível e o impossível para cuidar de sua família. Levo comigo pra sempre todos os momentos incríveis e esse amor sem limites que ele transbordava nas pessoas ao seu redor.

Também dedico o esforço e perseverança durante meu tempo de graduação ao meu irmão Arthur, que compartilha comigo o fascínio pela ciência e pelo desconhecido. Que seu futuro seja tão brilhante quanto puder sonhar.

Agradecimentos

Agradeço primeiramente aos meus pais e família, pelo o apoio e carinho durante todos esses anos. Em especial à minha mãe, Juliana, e minha madrinha Luciana, que sempre me incentivaram à leitura, o que despertou meu interesse pela ciência, e por me apresentar ainda pequeno ao meu primeiro computador.

À professora Carla, que me ajudou a escolher o tema deste trabalho e durante todo o processo sempre esteve disposta a me orientar com paciência e atenção, e ao professor Wilson pelo auxílio durante os procedimentos de conclusão de curso.

Agradeço a equipe do InterSCity pelo apoio nas fases iniciais do meu trabalho e pela oportunidade de contribuir para esse projeto.

À todos da equipe do SEPAS-TCU, que durante meu estágio me proporcionaram aprendizados não somente técnicos, mas de trabalho em equipe e organização. Sempre os levarei como mentores incríveis e exemplos a serem seguidos na minha carreira.

Agradeço aos meus amigos do Manos da 50, que há muitos anos compartilham as lutas diárias e momentos inesquecíveis juntos.

Aos grandes amigos que fiz durante a minha graduação, pelas longas aulas encaradas juntos e noites viradas terminando trabalhos. Em especial para meus parceiros de graduação e agora de trabalho e vida - Matheus, Maximillian e Murilo, que incontáveis vezes me salvaram na universidade. Agradeço também a minha amiga Miriã, por todas as conversas sem fim e apoio no que foi o ano mais maluco da minha vida.

Resumo

O InterSCity é uma plataforma de cidades inteligentes de *software livre*, baseada em uma arquitetura de microsserviços e desenvolvida para auxiliar aplicações de cidades inteligentes por meio de serviços reusáveis, interoperáveis e escaláveis.

Essa iniciativa disponibiliza uma série de funcionalidades em nuvem como armazenamento de dados de sensores, gerenciamento de recursos e comandos de atuação, com o intuito de unificar os mais variados tipos de soluções de *smart cities* que vierem a ser desenvolvidas. Atualmente, o InterSCity não dispõe de uma abordagem eficiente no que tange sua funcionalidade de comunicação com dispositivos físicos conectados à plataforma.

O objetivo deste trabalho é aplicar conceitos relacionados à IoT para propor uma nova versão do módulo de controle de atuação do InterSCity de forma a permitir que a troca de comandos seja realizada de acordo com os requisitos não-funcionais do projeto, por meio de uma melhor organização dos módulos e a utilização de protocolos de mensagem *machine-to-machine*, tecnologia chave para o ecossistema de cidades inteligentes.

Após uma análise do funcionamento da camada de atuação da plataforma, foram selecionados alguns pontos de interesse para evolução, visando melhorar a performance interna do sistema, o tempo de entrega de comandos aos dispositivos físicos e o consumo de rede dessa transmissão. A nova arquitetura proposta foi implementada e validada através de testes de caso, os quais mostraram que a refatoração sugerida da plataforma e a adoção do protocolo de mensageria MQTT trouxe ganhos de desempenho interno e de tempo de entrega superiores à 50%, além de reduzir o consumo de rede do dispositivo atuador em aproximadamente 67%.

Ao adotar uma abordagem mais focada no universo de Internet das Coisas, esperamos que o InterSCity possa entregar aos seus usuários um serviço de atuação mais adequado em termos de performance, mantendo a garantia de entrega e o uso de dados reduzido.

Palavras-chave: Cidades inteligentes, Internet das Coisas, InterSCity, protocolos de mensageria

Abstract

InterSCity is a free software smart cities platform, based on a microservice architecture developed to support the application of smart cities through reusable, interoperable, and scalable services.

This initiative provides a number of cloud functionalities such as sensor data storage, resource management and actuation commands, with the purpose of unifying the most varied types of smart cities solutions that will be developed in the future. Today, InterSCity does not use state-of-art technology to communicate with physical devices connected to the platform.

This work aims to design and implement a new actuator controller service that will allow the platform to send commands faster and more efficiently to the in-field actuators through the city using machine-to-machine messaging protocols, which are key technology for smart cities.

After studying the actuator controller module of the platform, several points of interest for evolution were selected, aiming to improve internal processing and command delivery time to the physical devices, and also its network usage. The proposed architecture was developed and validated through test cases, which showed that the suggested refactoring of the platform along with the the adoption of the MQTT messaging protocol reflected on up to 50% of performance gains in both internal and delivery steps. Furthermore, the network consumption on the device was reduced by aproximately 67%.

By adopting a more IoT-centered approach, we expect InterSCity to deliver a better suited actuation service in terms of low latency, while maintaining reliable delivery and low data usage.

Keywords: Smart Cities, Internet of Things, InterSCity, messaging protocols

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

API Application Programming Interface.

AWS *Amazon Web Services*.

EC2 *Elastic Compute Cloud*.

HTTP *Hypertext Transfer Protocol*.

HTTPS *Hypertext Transfer Protocol Secure*.

IoT *Internet of Things*.

ISO *International Organization for Standardization*.

M2M *machine-to-machine*.

MOM *Message-Oriented Middleware*.

MQTT *Message Queueing Telemetry Transport*.

NTP *Network Time Protocol*.

QoS *Quality of Service*.

REST *Representational State Transfer*.

RPC *Remote Procedure Call*.

TCP *Transmission Control Protocol*.

TIC Tecnologias de Informação e Comunicação.

TLS *Transport Layer Security*.

USP Universidade de São Paulo.

UUID *Universally Unique Identifier*.

Capítulo 1

Introdução

Espera-se que 70% da população mundial, mais de seis bilhões de pessoas, já estará vivendo em meios urbanos e regiões adjacentes até o ano de 2050 [2]. Fenômenos provenientes desta metropolização, como o crescimento de congestionamentos, escassez de recursos, impactos ambientais e questões de segurança pública demandam cada vez mais um modelo eficiente e moderno de um ecossistema urbano conectado.

Com o rápido crescimento das tecnologias de informação e comunicações (TIC) o conceito de **cidades inteligentes** vem ganhando relevância, porém ainda existe um esforço considerável para que esses serviços se conectem entre si e entreguem um real valor à sociedade [3]. Iniciativas de implementar tais ecossistemas surgem a cada dia, tendo como exemplo a plataforma SmartSantander[1] desenvolvida na Espanha, que conta com soluções integradas de mobilidade e realidade aumentada [4], e a Amsterdam Smart City[2], outra iniciativa que tem como objetivo fornecer serviços variados de cidades inteligentes com plataformas de *smart grid*, monitoramento ambiental e outros.

A variável em comum entre as aplicações citadas acima é a simples fórmula de adicionar inteligência digital à sistemas urbanos existentes, entregando informações transparentes e em tempo real nas mãos de cidadãos para auxiliar na tomada de decisão. Essas ferramentas podem salvar vidas, evitar crimes, economizar tempo e promover uma melhor interconexão social [5].

A crescente implantação de sensores e dispositivos inteligentes em diferentes âmbitos sociais e econômicos tem tornado o termo *Internet of Things* (IoT) uma grande *buzzword* no universo da tecnologia. A ideia por trás desse conceito é a presença de uma variedade de objetos, dispositivos ou "coisas"(sensores, atuadores, celulares e outros) que têm a capacidade de interagir entre si e executar tarefas em cooperação para atingir metas em comum [6]. Com o advento dessa intercomunicação, partes do mundo físico que antes

[1] `http://www.smartsantander.eu/`
[2] `https://amsterdamsmartcity.com/`

não tinham ligação com o meio virtual revelam novas oportunidades de se tornarem mais inteligentes e gerar mais valor à sociedade: No ecossistema de cidades inteligentes, essas "coisas"podem tomar forma de semáforos, postes, vagas de estacionamento e outras entidades urbanas as quais interagimos no nosso dia-a-dia.

A infraestrutura de soluções de IoT é caracterizada por uma malha heterogênea de dispositivos e aplicações, que devem oferecer maneiras de monitorar e controlar seus artefatos utilizando tecnologias e protocolos adequados. O principal objetivo deste tipo de sistema é conectar múltiplas redes e seus objetos, para que a coleta, compartilhamento, análise e gestão de recursos possa ser realizada através de diferentes ambientes e domínios [7].

Além de trazer benefícios de interação direta e em tempo real, essa adaptação do mundo físico para o virtual tende a gerar um volume extenso de dados que são coletados continuamente em todos os sistemas em funcionamento. Utilizando técnicas de *data mining* e aprendizagem de máquina, informações valiosas podem ser extraídas dos dados brutos fornecidos pelos vários sensores, possibilitando um melhor entendimento do contexto do sistema e gerando novos *insights* para tomada de decisão.

1.1 Motivação

Atualmente, a evolução de sistemas IoT se depara com grandes desafios técnicos, como limitações de rede e identificação, segurança de dados, recursos computacionais, consumo energético, além do recorrente problema de escalabilidade [8]. Novas abordagens e propostas têm sido estudadas nos últimos anos com o objetivo de atingir o estado da arte no que se refere à tecnologias, protocolos e arquiteturas que irão impulsionar o crescimento desse tipo de aplicação no mercado de ciência e inovação. Plataformas de cidades inteligentes como o InterSCity[9] se comunicam ativamente com redes de dispositivos IoT, o que torna a evolução e aprimoramento desses recursos imprescindível para que essas iniciativas possam se tornar bem-sucedidas.

O InterSCity[3] é uma plataforma de *software* livre que tem como principal objetivo simplificar o desenvolvimento de soluções inovadoras no âmbito de cidades inteligentes, disponibilizando serviços e integrações através de um *middleware* em nuvem que busca suprir os principais requisitos funcionais e não-funcionais de projetos de *smart cities* [10]. A plataforma fornece uma API que permite gerenciar recursos IoT e armazenar seus dados coletados.

Além disso, o sistema fornece um controle de atuação que tem como objetivo enviar comandos à dispositivos atuadores cadastrados na plataforma. Com ele, usuários podem

[3] `http://interscity.org`

se comunicar diretamente com recursos implantados pela cidade para executar instruções, alterar estados e atualizar parâmetros, por exemplo.

Com o objetivo de contribuir no crescimento da plataforma que ainda se encontra em desenvolvimento, analisamos as pendências e oportunidades de evolução juntamente com a equipe do InterSCity. A camada de atuação responsável pela comunicação com dispositivos físicos (*hardware*) atualmente funciona de maneira simplificada e necessita de novas iterações de desenvolvimento para se ajustar aos padrões e tecnologias de sistemas IoT, como documentado no repositório[4] do projeto.

1.2 Objetivos

Este trabalho tem como objetivos a análise, o planejamento, o desenvolvimento e os testes de caso de uma nova abordagem para o módulo de atuação do InterSCity. Para isso, há os seguintes objetivos secundários:

- Estudo e análise da plataforma e ferramentas utilizadas;
- Análise de padrões de arquitetura para comunicação;
- Proposta do novo serviço de atuação;
- Implementação do serviço proposto;
- Testes com dispositivos reais da plataforma antes e depois da alteração;
- Análise e discussão dos resultados obtidos.

1.3 Estrutura do trabalho

Apresentamos no Capítulo 2 um referencial teórico que aborda uma visão geral do InterSCity e sua arquitetura, assim como conceitos chave para projetos IoT como abordagens de comunicação e protocolos de mensagem. No Capítulo 3 analisamos os principais problemas e pendências do controle de atuação do InterSCity, apresentamos as propostas para evoluir a arquitetura, listamos as ferramentas utilizadas para executar a proposta e as etapas efetuadas para desenvolver o novo módulo. No Capítulo 4 apresentamos testes de caso realizados para comparar a performance das duas versões da plataforma e seus resultados, e por fim, no Capítulo 5, finalizamos este trabalho discutindo sobre as considerações finais, possíveis contribuições futuras e novos focos de estudo.

[4] `https://gitlab.com/interscity/interscity-platform`

Capítulo 2

Fundamentação Teórica

Neste capítulo, apresentamos o contexto do projeto explorado por este trabalho, definindo seu escopo e arquitetura. São apresentados também alguns conceitos técnicos que serão utilizados durante a fase de proposta e implementação.

2.1 O InterSCity

A plataforma faz parte de um projeto de pesquisa criado pelo Centro de Competência em Software Livre do Instituto de Matemática e Estatística da Universidade de São Paulo (USP), com o apoio do Instituto Nacional de Ciência e Tecnologia (INCT). Tendo como base fundamental o desenvolvimento colaborativo e o uso de software livre, o InterSCity conta com colaboradores de várias instituições acadêmicas que através de práticas ágeis atuam na evolução e manutenção da plataforma [9].

A plataforma é licenciada sob MPLv21(Mozilla Public License Version 2.0), e foi projetada seguindo os padrões de arquitetura de microsserviços (MSA) [11] por meio do *framework* Ruby on Rails[1] e utilizando gerência de configuração em contâineres independentes.

O versionamento do projeto é administrado por meio do GitLab[2], onde é possível ter acesso ao código fonte, documentação, ambientes de teste e atualizações sobre alterações no código. Discussões pertinentes ao uso da plataforma e do projeto como um todo são tratadas via fórum do Google Groups[3], no qual os colaboradores e usuários do sistema podem se comunicar mais livremente.

[1] `https://rubyonrails.org/`
[2] `https://gitlab.com/interscity/interscity-platform`
[3] `https://groups.google.com/forum/#!forum/interscity-platform`

2.1.1 Arquitetura

Com o intuito de solucionar os problemas e desafios mencionados na seção anterior, o InterSCity foi proposto como uma plataforma baseada em microsserviços, tendo como objetivo prover um *middleware*[4] com infraestrutura de alta qualidade, modular, escalável, e reusável em diversas soluções de cidades inteligentes e grupos de pesquisa, assim como no governo e empresas privadas [9]. Para atingir esse objetivo, sua arquitetura é composta de módulos desacoplados e independentes que se comunicam por meio de serviços de mensageria e requisições REST (*Representational State Transfer*), e que podem ser implantados de forma automatizada via contêineres Docker[5] pré-configurados, reduzindo o esforço na gerência de dependências e *deploy*, e promovendo um ambiente mais escalável ao facilitar a replicação de instâncias.

Cada um destes módulos possui funções específicas, de forma a cumprir requisitos não funcionais do InterSCity como interoperabilidade, modularidade via serviços e evolução descentralizada [10].

A Figura 2.1 apresenta a estrutura geral da arquitetura com todos seus módulos e pontos de acesso externo, tanto dos sensores e atuadores que fazem parte da malha de dispositivos como aplicações que acessam os dados contidos na plataforma.

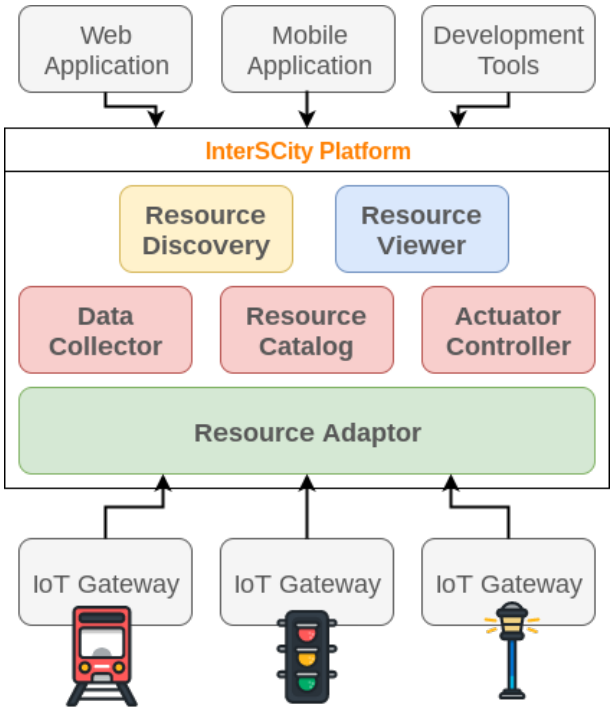

Figura 2.1: Arquitetura do InterSCity (Fonte: [9]).

[4] *Software* que possibilita a comunicação e a gestão de dados para aplicações distribuídas.
[5] `https://www.docker.com/`

A gerência de recursos do InterSCity é baseada em quatro tipos de entidades básicas. Os dispositivos de cidades inteligentes são cadastrados na plataforma como **Recursos**, sendo registrados juntamente com atributos como sua localização geográfica, descrição, *status* e outros. No momento deste registro deve-se informar também as **Capacidades** do dispositivo, as quais definem o tipo de dados recebidos e enviados pelo mesmo. Uma capacidade pode ser classificada com o tipo sensor, atuador ou informação.

Dispositivos com capacidades do tipo 'sensor' ou 'informação' podem enviar material coletado para a plataforma, que os registra como **Dados** na sua base. Já dispositivos com capacidades do tipo 'atuador' podem receber instruções provenientes de outras aplicações, registradas como **Comandos**. Um exemplo de contexto dessa arquitetura é demonstrado na Figura 2.2.

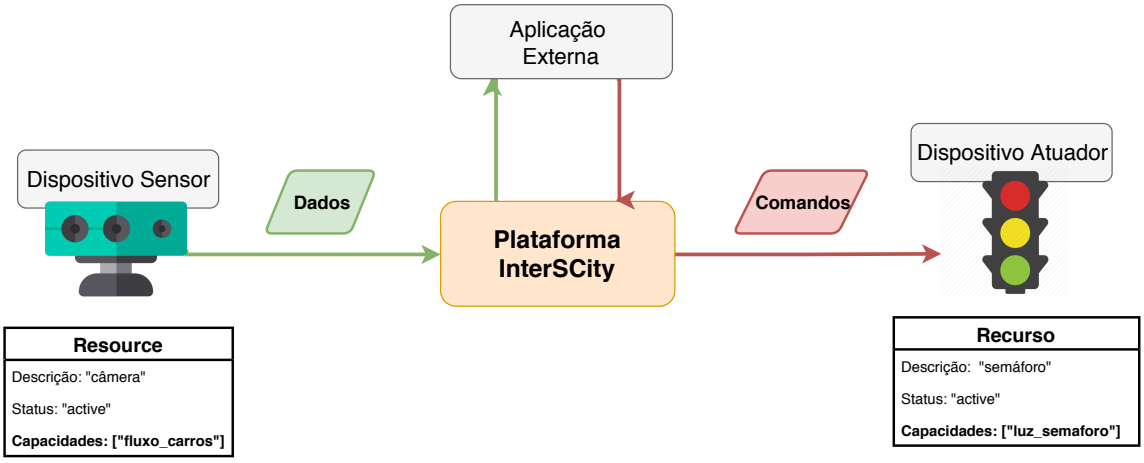

Figura 2.2: Interações dos recursos com a plataforma.

2.1.2 Módulos

Resource Adaptor

Principal ponto de comunicação com **dispositivos** da plataforma, o **Resource Adaptor** serve como um *proxy* que fica responsável por mediar operações de registro e atualização de recursos, a captação dos dados provenientes de sensores e a criação de subscrições para receber comandos de atuação [9]. O módulo apenas verifica a consistência e formato das informações recebidas, e então as encaminha por meio de chamadas REST ou fila de mensagens (RabbitMQ[6]) para outro componente da plataforma que irá armazenar e fazer uso dos dados.

[6] `https://www.rabbitmq.com/`

Resource Catalog

Ao registrar um novo dispositivo via *Resource Adaptor*, o mesmo é enviado através de requisição HTTP para o **Resource Catalog**, módulo que registra e atribui identificadores únicos (UUIDs) aos dispositivos cadastrados na plataforma, assim como notifica esse registro para os outros serviços da plataforma [9]. Todas as operações na base de *capabilities* também são mediadas pelo **Resource Catalog**.

Data Collector

Este módulo é responsável por armazenar os dados coletados pelos sensores conectados à plataforma, utilizando banco de dados NoSQL MongoDB[7]. O **Data Collector** também fornece serviços de busca aos dados históricos e atuais com diversos filtros, permitindo futuras integrações com ferramentas de processamento de dados como o Shock [12].

Resource Viewer e Discovery

Desenvolvidos com o objetivo de prover serviços mais sofisticados e de fácil acesso orquestrando chamadas internas ao *Data Collector* e ao *Resource Catalog*[9]. O **Resource Viewer** é o principal ponto de acesso de aplicações aos dados contidos na plataforma, disponibilizando informações sobre os recursos da cidade como localização, dados de sensores, e estatísticas. Já o **Resource Discovery** tem como função entregar *endpoints* de busca de recursos cadastrados por meio de filtros, possibilitando a descoberta de novos dispositivos implantados.

Actuator Controller

Compondo a camada de comunicação da plataforma, este microsserviço permite o envio de comandos à recursos com funções de atuação conectados ao InterSCity. Esses comandos passam por verificações de consistência e então são repassados ao *Resource Adaptor*, que se comunica diretamente com os dispositivos externos. Além disso, também armazena todos os comandos e seus metadados em sua base para possibilitar auditoria.[9]

2.1.3 Fluxo da plataforma

Envio e obtenção de dados

A Figura 2.3 ilustra o ciclo de dados no InterSCity desde o registro do recurso até sua visualização. Para a utilização de um dispositivo, deve ser feito o cadastro deste recurso IoT por meio do Resource Adaptor **(1)**, especificando atributos como descrição, capacidades

[7] `https://www.mongodb.com/`

(previamente registradas na plataforma), localização e outros parâmetros opcionais. Este registro é comunicado **(2)** ao Resource Catalog, que indexa a informação e retorna um identificador único (UUID) ao Resource Adaptor. Este identificador é retornado ao dispositivo **(3)** para que o mesmo possa executar ações no InterSCity. Quando o sensor envia dados para a plataforma através do Resource Adaptor, este então os persiste **(4)** no Data Collector, que gerencia todas as informações de sensores e as expõe **(5)** para outros módulos como o Resource Viewer. O usuário final ou aplicação externa pode então acessar diretamente o Resource Viewer **(6)** e obter dados da plataforma.

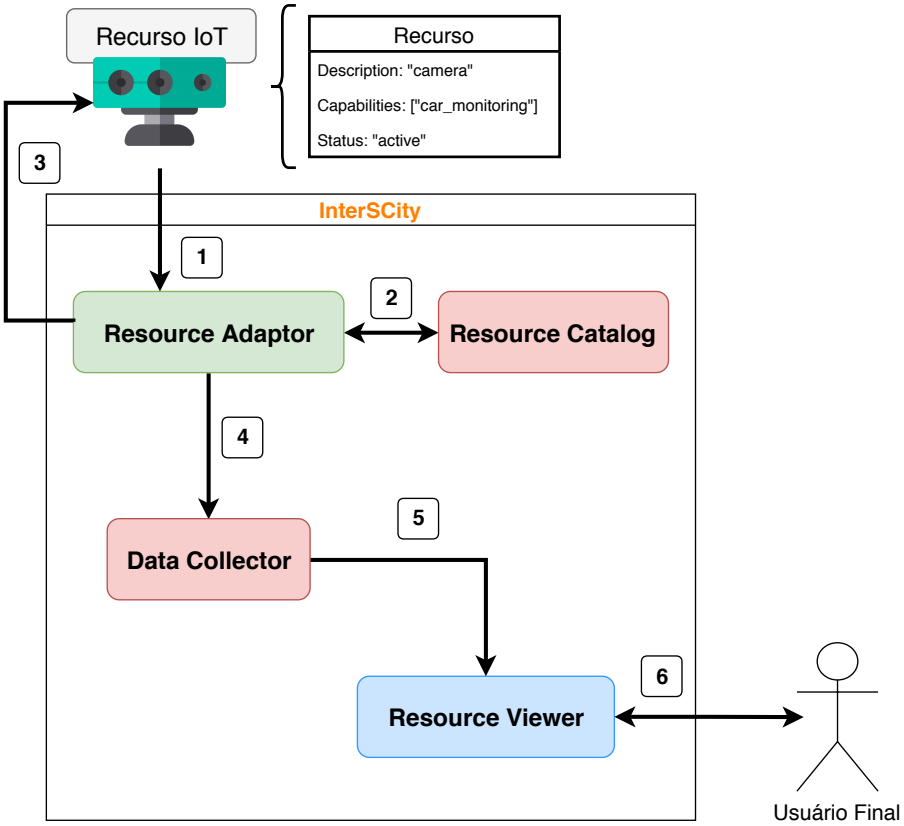

Figura 2.3: Ciclo de dados no InterSCity (Fonte: [12]).

Envio de comandos de atuação

A plataforma permite a interação com os recursos conectados que possuem capacidade de atuação através das etapas definidas na Figura 2.4, que divide o ciclo de **subscrição** (em verde), e **envio de comandos** (em vermelho). Assumindo que o recurso atuador já se encontra cadastrado com seus devidos atributos assim como foi explicado no tópico anterior, o dispositivo deve registrar pelo Resource Adaptor uma subscrição **(1)**, informando seu UUID, a capacidade de atuação (por exemplo, estado do semáforo) que deseja receber

comandos e um endereço HTTP para o qual serão enviados os comandos no formato de requisições POST. No momento em que uma aplicação ou dispositivo externo deseja enviar um comando, este deve fazer uma requisição ao Actuator Controller **(3)** passando o UUID, a capacidade de atuação e seu novo valor como parâmetro. Este comando, após ser armazenado no Actuator Controller para fins de auditoria, é publicado **(4)** numa fila de mensagens gerenciada por uma instância do RabbitMQ. Através de um *worker* no Resource Adaptor que consome mensagens dessa fila, o comando é recebido **(5)** e verificado se possui alguma subscrição relacionada. Caso exista, ele então é enviado via *webhook* **(6)** para o endereço previamente cadastrado na subscrição do dispositivo.

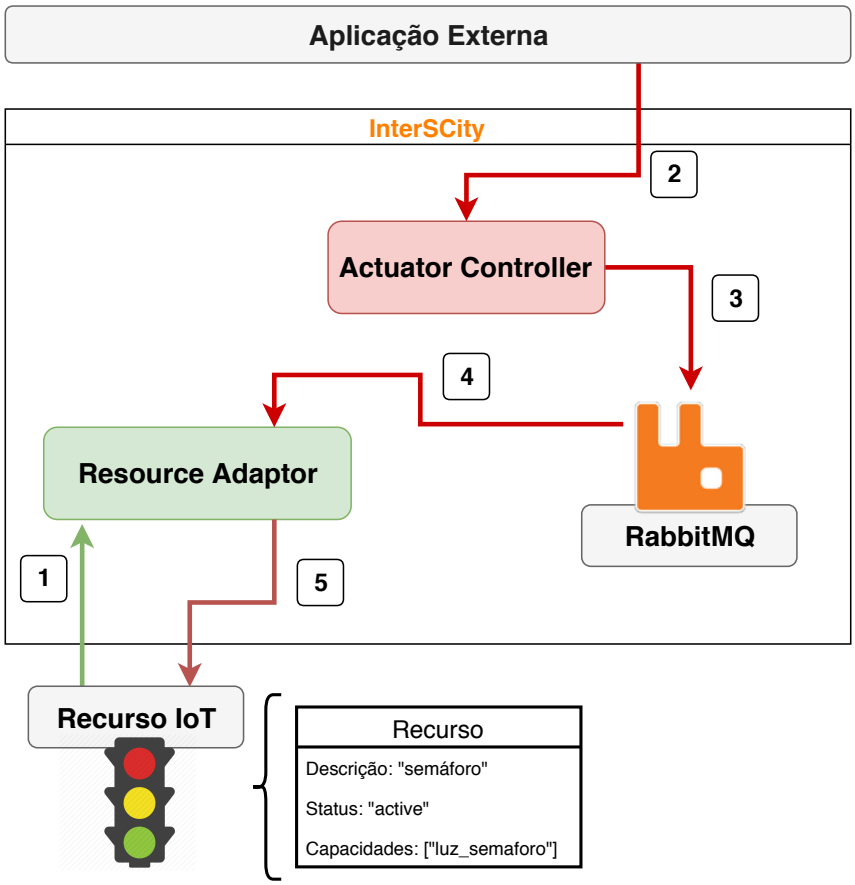

Figura 2.4: Ciclo de atuação no InterSCity (Fonte: [9]).

2.1.4 Controle de atuação

Iniciativas como o InterSCity propõem um ambiente unificado e heterogêneo de monitoramento, o qual possibilita a aplicação de processamento e mineração de dados entre distintas aplicações de cidades inteligentes que podem resultar em novas possibilidades de tomada de decisão dentro do ecossistema urbano. Informações de um ou mais sensores podem

gerar gatilhos para executar ações em outros dispositivos da cidade, tornando a malha de recursos implantados mais inteligente e interconectada.

Porém, a comunicação com aparelhos físicos de uma rede de cidades inteligentes apresenta cada vez mais desafios à medida que o volume de entidades aumenta e que surge a necessidade de respostas em tempo real. A troca de informações com esses dispositivos deve se basear em sistemas robustos e eficientes, que ofereçam garantia e agilidade na entrega dos dados.

Nos últimos anos, o uso de protocolos baseados em fila de mensagens vem tomando espaço no âmbito de aplicações de cidades inteligentes, trazendo novas abordagens de implementação que otimizam o processo de comunicação.

2.2 Protocolos de Mensageria

Com a contínua evolução de sistemas de *software* para modelos distribuídos que ultrapassam barreiras geográficas, organizacionais e comerciais, é necessário levar em consideração a infraestrutura de comunicação mais adequada. Ao apresentar requisitos como implantação flexível, performance em alto *throughput* e segurança, chamadas tradicionais usando RPC (Remote Procedure Call) muitas vezes falham em cumprir tais desafios. Para atingir tais demandas, uma alternativa de comunicação denominada MOM (*Message-Oriented Middleware*) foi concebida.

Um *middleware* orientado à mensagens é um tipo de servidor de aplicação criado com o objetivo de suportar o envio e recebimento de mensagens entre sistemas, para possibilitar quaisquer troca de informações necessária para a integração de serviços.

Ao construir um MOM, se torna necessário o uso de um serviço de enfileiramento de mensagens (*broker*) para mediar o processo de comunicação. Um *broker* de mensagens é um padrão arquitetural não só para troca de mensagens, mas também para sua validação, transformação e roteamento. Esse tipo de ferramenta geralmente é utilizada para administrar uma fila de mensagens ou *stream* de dados para múltiplos receptores, provendo armazenamento seguro, entrega de mensagens garantida e uma possível gestão de transações. Com isso, muitas vezes se torna possível desacoplar serviços REST, cumprir requisitos não-funcionais específicos e permitir reuso de funções intermediárias do sistema.

O uso de sistemas de filas trazem benefícios principalmente para o meio de aplicações IoT nos quais os dispositivos envolvidos possuem limitações físicas e de conexão, permitindo que dados sejam encaminhados de maneira assíncrona e confiável para outros ambientes garantindo que as informações serão persistidas e processadas eventualmente; mesmo em situações de pico de tráfego e perdas momentâneas de conexão.

Diversos padrões ISO para implementação desse tipo de *middleware* existem no mercado, como o XMPP, AMQP e MQTT. O AMQP[8] é um protocolo seguro e robusto com *overhead* reduzido em sua comunicação, se tornando uma boa escolha para aplicações de *Internet of Things*. É uma abordagem avançada, com vastas funcionalidades de segurança, roteamento flexível, persistência prolongada e a possibilidade de clusterização de dados de maneira nativa. Porém, é um protocolo "pesado"no que tange seu tráfego de pacotes e uso de rede, geralmente utilizado em sistemas de *gateway* e gerenciamento de mensagens dentro de redes privadas.

Visando uma alternativa que priorize o uso em dispositivos compactos, escolhemos para a análise neste capítulo o protocolo MQTT.

2.2.1 MQTT

Um exemplo de protocolo de mensageria presente no mercado é o MQTT (*Message Queueing Telemetry Transport*), baseado no tipo de comunicação *pub/sub*. Concebido pela IBM nos anos 90 e transformado em um padrão aberto OASIS[9] em 2014, esse protocolo implementa uma troca de mensagens simples e leve que minimiza o uso de rede, o consumo de energia e os requisitos do dispositivo, tornando-se uma escolha ideal para ambientes de comunicação *machine-to-machine* (M2M)[13].

Aplicações no ramo da saúde, monitoramento, telemetria e automação já utilizam esse protocolo, por representar uma alternativa ideal para ambientes IoT provendo um canal de troca de informações pequeno, barato, e com baixo custo de memória e energia[14].

Publicação/Assinatura

O modelo *pub/sub* adotado pelo MQTT define dois tipos de entidades: um *broker*, responsável pelo gerenciamento de mensagens e seu envio/recebimento, e inúmeros clientes, caracterizados como qualquer conexão que interage com o *broker* como um sensor de campo ou aplicativo móvel. Este processo, ilustrado na Figura 2.5, ocorre da seguinte forma:

1. Um cliente se conecta ao *broker* com a intenção de receber certo tipo de conteúdo;
2. Após a conexão inicial, este cliente informa o tipo de mensagem a ser ouvida especificando um **tópico**;
3. Um cliente que pretende enviar dados se conecta ao *broker* e publica sua mensagem em um tópico específico;

[8] `https://www.amqp.org/`
[9] `https://www.oasis-open.org/standards#mqttv3.1.1`

4. O broker então encaminha essa mensagem para todos os clientes que assinam esse tópico.

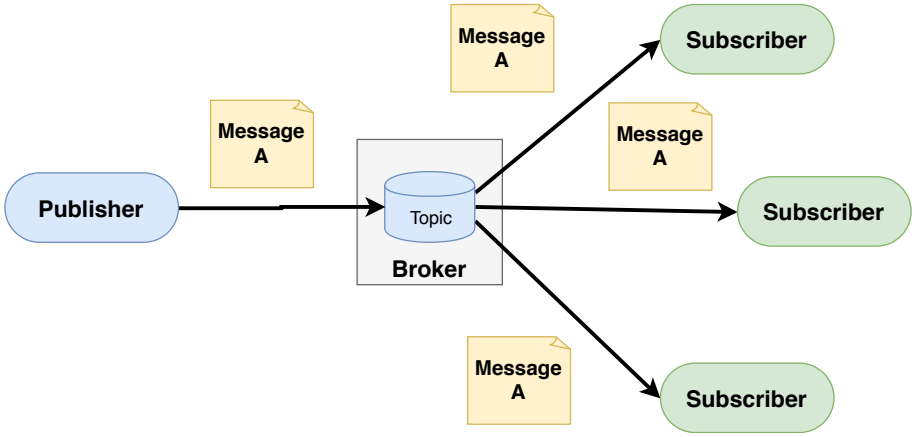

Figura 2.5: Formato *Pub/Sub*.

Ao utilizar esse padrão, o MQTT entrega um envio de mensagens com baixo acoplamento na relação cliente-servidor e permite o processamento assíncrono das informações repassadas pelo *broker*. Além disso, sua organização por tópicos torna o acesso aos dados mais eficiente e flexível, possibilitando a inscrição de clientes em diferentes escopos.

Formato da mensagem

Diferente do padrão HTTP, o MQTT funciona por meio de conexões TCP/IP persistentes, o que tende a economizar uma quantia significativa de recursos em contextos onde várias mensagens são trocadas. Sua estrutura de pacote (ilustrada na Figura 2.6) conta com um cabeçalho fixo simples para especificar o tipo de mensagem e outros indicadores, o tópico que ela será publicada, e uma carga útil (*payload*) binária de formato arbitrário, tal como JSON, XML, Base64 e outros.

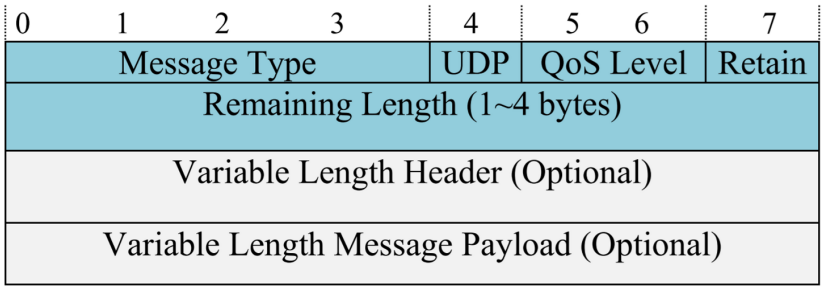

Figura 2.6: Pacote de mensagem do MQTT (Fonte: [14]).

Segurança e Qualidade de Serviço

O MQTT utiliza o protocolo de transporte TCP, que originalmente não fornece uma comunicação encriptada. Para solucionar esse problema, diversas implementações de *brokers* MQTT (como o HiveMQ[10] e o Eclipse Mosquitto[11]) disponibilizam o uso do certificado *Transport Layer Security* (TLS) para a validar a conexão entre cliente e servidor, no nível da camada de transporte. Existe um custo adicional de CPU e tráfego para a comunicação ao utilizar o TLS, que embora normalmente seja insignificante no *broker*, pode se tornar um problema em dispositivos com recursos computacionais limitados [15].

Para a camada de aplicação, o protocolo também fornece autenticação por meio de usuário/senha no momento da conexão com o *broker*. O escopo de acesso também pode ser delimitado caso a implementação do *MQTT* a ser utilizada ofereça funcionalidades de autorização, criando regras específicas para cada tópico que podem definir as operações (*publish*/*subscribe*/ambos) e níveis de qualidade de serviço (0/1/2) permitidos.

Um problema comum ao se comunicar pela rede é a garantia de recebimento da informação. O MQTT permite o uso de diferentes níveis de qualidade de serviço (QoS) para indicar com que consistência as mensagens devem ser entregues aos clientes de determinado tópico [13]. As opções de nível de qualidade de entrega do protocolo e suas descrições estão listadas na Tabela 2.1.

O nível de qualidade de serviço é um acordo firmado entre o cliente e o *broker*, possibilitando que um *publisher* envie mensagens com QoS 1 e outros clientes *subscribers* as recebam com QoS 2, por exemplo.

[10] `https://www.hivemq.com/`
[11] `https://mosquitto.org/`

Tabela 2.1: Níveis de Qualidade de Serviço

Nível	**Estratégia**	**Descrição**
QoS 0	no máximo uma vez	Utiliza a técnica de *best effort* (melhor esforço), semelhante ao transporte UDP. A mensagem é enviada apenas uma vez e não fornece confirmação de entrega, sendo aplicável em situações onde se deseja o máximo *throughput* e a perda de algumas mensagens não impacta no funcionamento do sistema.
QoS 1	pelo menos uma vez	Garante que a mensagem enviada pelo *publisher* foi recebida por todos os clientes inscritos conectados, enviando-a repetidamente até que seja recebida uma confirmação de entrega. Esse nível pode acarretar no recebimento de mensagens duplicadas nos clientes, caso ocorra um atraso no envio da confirmação.
QoS 2	exatamente uma vez	Para situações nas quais não se pode ocorrer o recebimento de duplicatas, esta opção garante o envio único por meio de confirmações bilaterais no momento da comunicação. É o nível mais seguro, porém acarreta no aumento do tráfego e devido ao número de confirmações trocadas, impactando na latência dos envios.

Comparativo HTTP e MQTT

Atualmente, o envio de comandos da plataforma do InterSCity é executado por meio de comunicação HTTP, Experimentos conduzidos para comparar a performance entre o protocolo HTTP e o MQTT para soluções de IoT já apresentam diferenças significativas na vazão das mensagens, consumo de energia e uso de dados de rede, principalmente em redes 3G comumente utilizadas em ambientes de cidades inteligentes. A Tabela 2.2 apresenta os dados obtidos por um experimento [1] que testou em um dispositivo móvel o recebimento de 1024 mensagens em dois tipos de redes para comparar a performance entre o HTTPS e o MQTT com TLS ativado. A vazão de envio dos pacotes é expressivamente maior ao utilizar o protocolo de mensageria, assim como sua eficácia na entrega.

Tabela 2.2: Envio de mensagens usando HTTPS/MQTT (Fonte: [1]).

	3G		**Wifi**	
	HTTPS	**MQTT**	**HTTPS**	**MQTT**
% Battery / Hour	18.43%	**16.13%**	**3.45%**	4.23%
Messages / Hour	1708	**160278**	3628	**263314**
% Battery / Message	0.01709	**0.00010**	0.00095	**0.00002**
Messages Received	240 / 1024	**1024 / 1024**	524 / 1024	**1024 / 1024**

Essa defasagem deve-se principalmente pelo fato do tipo de conexão estabelecida, que apesar de se tratar de implementações do mesmo protocolo TCP, na versão do MQTT a conexão entre cliente e servidor é mantida durante todo o período de troca de informação. No uso do HTTPS, a conexão é renovada múltiplas vezes, agravando o impacto na perfomance ainda mais por utilizar SSL e ter que lidar com alto volume de requisições concorrentes.

2.2.2 Processamento Assíncrono em Segundo Plano

A maior parte dos sistemas de cidades inteligentes requerem um funcionamento constante dos seus serviços, recebendo e processando dados em nuvem em rotinas de quase sempre 24 horas por dia e 7 dias na semana. Com o intuito de garantir a execução da totalidade das tarefas requisitadas pelos clientes conectados, aplicações nesse contexto devem dispor de mecanismos de armazenamento, gerenciamento e execução de processos de maneira assíncrona e em segundo plano, permitindo melhor gestão de recursos e o uso de computação paralela.

Um processo em segundo plano é uma tarefa de um sistema executada em um contexto paralelo ao programa principal, e que geralmente efetuam procedimentos de monitoramento, *logging* e troca de informações. Esse tipo de ação geralmente não requer interação do usuário/plataforma e ocorrem sem bloquear o fluxo de atividades do serviço primário.

Em aplicações onde o programa recebe instruções de maneira assíncrona e com alta vazão de requisições, o processamento em segundo plano pode ser aplicado através do uso de ferramentas que facilitam a gerência das tarefas por meio de filas, permitindo que o sistema possa escolher parâmetros de decisão de escalonamento para a sua execução. Um exemplo de funcionamento desse tipo de ferramenta é demonstrado na Figura 2.7.

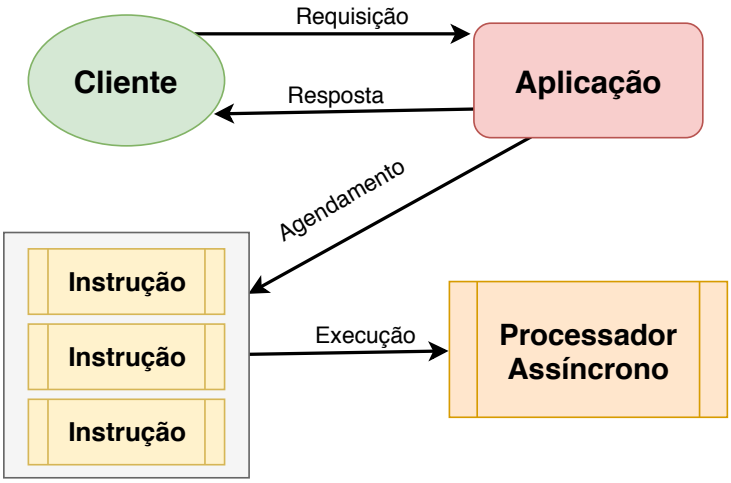

Figura 2.7: Topologia do processamento assíncrono.

Apesar de suas vantagens, o processamento de instruções assíncronas pode acarretar em alguns problemas, devido à falhas no momento da execução, falta de recursos físicos, parâmetros incorretos e outros. Em um ambiente ideal, a implementação dessa funcionalidade deve estar preparada para lidar com possíveis erros de execução e possuir mecanismos de solução, seja por meio de tentativas adicionais após certo tempo, notificações ao processo primário ou simplesmente no descarte do pedido. Ferramentas como o Sidekiq[12] e o Resque[13] são exemplos de implementações que disponibilizam controle de filas de processos, monitoramento de desempenho, registro de atividades e outros aspectos pertinentes ao uso de processamento em segundo plano, viabilizando uma gerência mais eficiente e segura das instruções recebidas pelo programa.

[12] `https://sidekiq.org/`
[13] `https://github.com/resque/resque`

Capítulo 3

Proposta e Implementação

Neste capítulo discutimos os principais problemas encontrados no processo de atuação do InterSCity, entendendo seus impactos e possíveis causas, e propomos soluções que visam otimizar sua estrutura e desempenho detalhando a arquitetura proposta e detalhes sobre o processo de implementação.

3.1 Proposta

Na implementação atual, o fluxo do atuador está fortemente acoplado ao módulo do *Resource Adaptor*, o que não se encaixa nos requisitos não-funcionais propostos pela plataforma [10] e acaba criando um nível indesejado de interdependência dos módulos. Essa dependência gera comunicações desnecessárias entre as camadas do InterSCity e podem afetar na perfomance e tempo de resposta das suas funcionalidades. A Tabela 3.1 lista as funcionalidades de atuação presentes na plataforma e os módulos que as executam.

Tabela 3.1: Ações do fluxo de atuação

Funcionalidade	**Módulo responsável**
Cadastro de Subscrição	Resource Adaptor
Armazenamento de Subscrição	Resource Adaptor
Envio de comandos (plataforma)	Actuator Controller
Obtenção de log de comandos	Actuator Controller
Encaminhamento de comandos (dispositivo)	Resource Adaptor
Atualização de status do comando	Resource Adaptor

A Figura 3.1 apresenta em maior nível de detalhamento o fluxo descrito anteriormente na Seção 2.1.3, apontando os principais passos do envio de comandos no InterSCity.

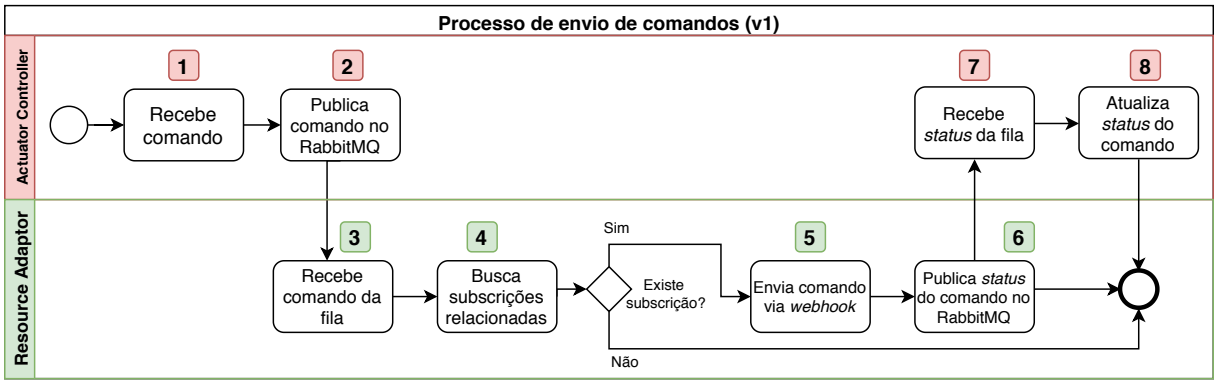

Figura 3.1: Fluxo de envio na versão atual.

No momento que a plataforma recebe **(1)** um comando, o mesmo é criado e armazenado na base de dados do Actuator Controller. Após a criação, o objeto referente ao comando é publicado em uma fila de mensagens do RabbitMQ **(2)** e consumido pelo Resource Adaptor **(3)**. A partir da informação contida no comando, é efetuada uma busca **(4)** pelas subscrições existentes para o recurso e capacidade especificados. No caso positivo, o comando é então enviado **(5)** por meio de uma requisição POST para a URL definida na subscrição. De acordo com o código de resposta da requisição de envio, o *status* do comando é publicado em outra fila **(6)** que é monitorada pelo Actuator Controller. A nova informação de *status* é consumida **(7)** e então atualizada na base **(8)**.

Tendo como base essas etapas, é possível identificar uma possível perda de performance nos passos 2 e 6 por dependerem de um repasse da informação do comando entre dois módulos diferentes. Esse problema se dá por conta da localização da base de subscrições, que atualmente é gerenciada no Resource Adaptor, e força o fluxo do comando a passar por esse módulo para efetuar a etapa de busca.

Ao construir uma solução que torne as funcionalidades de atuação mais contidas e modulares, as subscrições seriam administradas pelo próprio Actuator Controller, permitindo que o processo seja executado localmente sem precisar de agentes externos.

Durante a etapa de envio do comando, o Resource Adaptor utiliza de uma ferramenta de processamento em *background* chamada Sidekiq para enfileirar os pedidos de encaminhamento dos comandos. Dessa maneira, um grande volume de requisições pode ser efetuado pois o gerenciamento das tarefas é escalonado e administrado pelo Sidekiq. Porém, é necessário rever os parâmetros utilizados em sua configuração, assim como adaptar os *workers* às outras alterações desta proposta.

Outro ponto de atenção é que a comunicação externa entre o InterSCity e os dispositivos atuadores é feita por meio de *webhooks*, o que pode limitar o volume de dados trafegados e forçar a utilização de servidores dedicados nos dispositivos apenas para recebimento de

comandos. Protocolos baseados em mensageria e formatos *Pub/Sub* têm tomado mais espaço no universo de Internet das Coisas [16], por apresentar melhores indicadores de performance, uso de dados, segurança e outros. A implementação de subscrições que forneçam a entrega de comandos através de protocolos centrados em aplicações IoT além do tradicional *webhook* poderia aprimorar o desempenho enquanto mantém compatibilidade e flexibilidade.

Com base nas informações apresentadas acima, foram propostas as seguintes alterações:

1. Centralização das funcionalidades de atuação no Actuator Controller
2. Mudança da base de subscrições para o módulo de atuação
3. Ajuste no processamento assíncrono para envio de comandos
4. Uso de protocolo de mensageria no envio de comandos

Para atingir estes objetivos, uma nova topologia do controle de atuação foi desenhada com o intuito de agregar todas as propostas acima em um formato de serviço que atenda também os requisitos não-funcionais do InterSCity.

3.1.1 Arquitetura Proposta

A Figura 3.2 ilustra o novo fluxo completo de atuação na implementação proposta. Ao receber um comando de uma aplicação externa **(1)**, o Actuator Controller efetua uma validação dos dados contidos na requisição. Com todos os campos validados, o módulo busca em sua base **(2)** todas as subscrições registradas referentes ao recurso IoT e à capacidade especificados no comando. Caso existam subscrições ativas **(3)**, o controlador cria uma tarefa de execução assíncrona utilizando o Sidekiq para cada uma delas. Na condição de que o tipo da subscrição seja *webhook*, o comando é enviado no formato de uma requisição POST para a URL cadastrada na base. Já para o tipo MQTT, comando é publicado no tópico referente à seu recurso e capacidade. Por fim, o *status* do comando é atualizado **(4)** na base de dados, sendo classificado como pendente, processado, rejeitado ou falha.

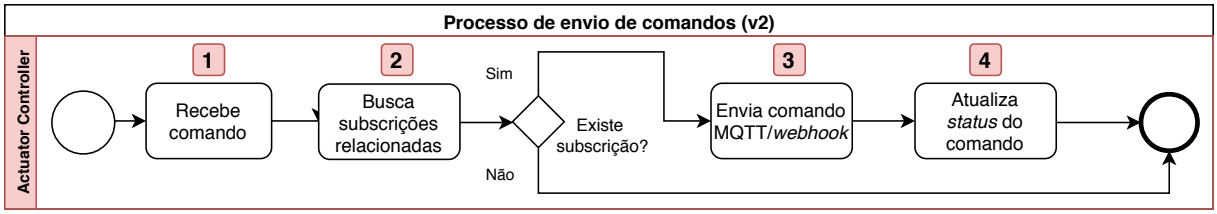

Figura 3.2: Fluxo de envio na versão proposta.

3.2 Implementação

A implementação do controle de atuação desenvolvido neste trabalho tem como meta aliar os requisitos propostos pelo projeto InterSCity aos métodos e protocolos mais adequados à sistemas IoT, visando criar uma arquitetura com melhor modularização e escalabilidade, assim como garantir um canal de comunicação com os dispositivos de cidades inteligentes mais robusto, seguro e eficiente.

Todo o código produzido durante a fase de implementação deste trabalho segue os padrões de desenvolvimento de *software* da plataforma do InterSCity, como documentado nos manuais fornecidos pela equipe do projeto. Além de passar por checagens de qualidade de código e folhas de estilo, o material presente no repositório do trabalho possui toda a documentação atualizada das chamadas da API via Swagger[1]. Os testes unitários automatizados foram adaptados para refletir as mudanças arquiteturais do módulo de atuação, com o intuito de manter a compatibilidade com os *pipelines* de teste adotados pelo InterSCity caso a contribuição ao projeto seja efetivada.

Dividimos a implementação do novo módulo em três etapas: **(1)** a refatoração e centralização das funções de atuação; **(2)** uso de tarefas de processamento assíncrono para enfileirar e executar o envio de comandos; e **(3)** implantação do protocolo MQTT para acesso dos comandos da plataforma.

3.2.1 Ferramentas Utilizadas

Para executar os envios em segundo plano de forma robusta e eficiente, será utilizada a ferramenta Sidekiq, já presente na implementação atual da plataforma e que possui um sistema nativo de *logs*, tratamento de exceções e parâmetros de priorização de filas.

Escolhemos o MQTT como tecnologia de *broker* de mensagens a ser implantada, por se tratar de uma ferramenta desenvolvida com foco em aplicações de Internet das Coisas. Apesar do fato de que o InterSCity já utiliza o RabbitMQ para outras funções internas, estas são puramente voltadas para a comunicação entre seus módulos, e não refletem o contexto de troca de informações com dispositivos físicos inseridos em ambientes de grande malha de sensores e atuadores. Além disso, o MQTT já possui diversas implementações *open-source* de cliente e servidor para várias linguagens de programação, diminuindo o custo técnico necessário para adotá-lo em projetos já existentes.

Durante a fase de testes, para simular as chamadas REST de criação de comandos, foi empregado o ambiente de desenvolvimento de API Postman[2] que fornece uma interface

[1] `https://swagger.io/`
[2] `https://www.getpostman.com/`

amigável para prototipação e teste de chamadas HTTP por meio de execuções programáveis e monitoradas.

Para organizar e analisar os resultados obtidos, a biblioteca de análise de dados Pandas[3], baseada em Python, foi aplicada, o que facilitou o tratamento dos dados e a geração de representações gráficas de maneira programática. O código-fonte do *pipeline* em Pandas para análise e exportação dos dados capturados pelos testes está incluído no repositório deste trabalho.

3.2.2 Centralização de Funcionalidades

Como primeiro passo, iniciamos a migração dos serviços com uma alteração na base de subscrições, que utilizava tecnologia de banco de dados relacional PostgreSQL[4] localizada no Resource Adaptor apenas para essa finalidade. Adaptamos o modelo do objeto de subscrição para o formato de registro do MongoDB, banco de dados já utilizado no Actuator Controller para armazenar os comandos enviados à plataforma. Ao transferir os dados de subscrição para o banco do módulo de atuação, foi possível eliminar a instância Docker do PostgreSQL da máquina do Resource Adaptor, reduzindo seu custo de processamento, tempo de inicialização e espaço em disco.

Após essa etapa, todos os microsserviços de controle de subscrições (criação, busca, edição e remoção) foram movidos para o módulo do Actuator Controller, disponibilizando aos usuários as mesmas requisições da antiga implementação.

Além de centralizar funcionalidades correlatas em um mesmo domínio, outro benefício encontrado nessa migração foi observado no momento da criação de subscrições. Na antiga implementação, o Resource Adaptor precisava se comunicar com o Resource Cataloguer para validar as capacidades do recurso definido na requisição, o que não acontece na versão proposta pelo fato de que o Actuator Controller possui uma réplica local da base de recursos cadastrados na plataforma.

Uma das pendências para evoluções futuras documentadas no repositório da plataforma[5] é a implementação de um sistema de checagem de disponibilidade dos *webhooks* cadastrados. Com as subscrições presentes no mesmo banco de dados que os comandos recebidos, futuramente pode ser desenvolvida uma tarefa periódica que execute uma análise rápida dos *status* dos comandos mais recentes e com isso identificar *webhooks* indisponíveis ou incorretos.

[3] `https://pandas.pydata.org/`
[4] `https://www.postgresql.org/`
[5] `https://gitlab.com/interscity/interscity-platform/resource-adaptor/issues/4`

3.2.3 Processamento Assíncrono

A solução de processamento em segundo plano adotada atualmente é administrada pelo Resource Adaptor, o qual utiliza o Sidekiq para disparar tarefas em filas sempre que um comando deve ser enviado para um dispositivo. Ao unificar as funções de atuação, também foi feita a migração da instância do Sidekiq e seu banco de dados Redis para o Actuator Controller.

O processo original foi mantido na nova implementação, adicionando novas configurações e parâmetros para que fosse adaptado ao método proposto de envio de comandos. A topologia foi alterada para duas filas distintas, com o intuito de organizar separadamente o fluxo de comandos com envio *webhook* e *MQTT*. Dessa forma, a prioridade de corrida de cada uma pode ser configurada sob demanda assim como o tipo de *workers* a serem utilizados para cada função. Essa divisão também contribui para que os registros (*logs*) da ferramenta sejam melhor categorizados, facilitando sua manutenção e monitoramento.

A Figura 3.3 ilustra a atividade do Sidekiq na plataforma para os dois tipos de subscrição.

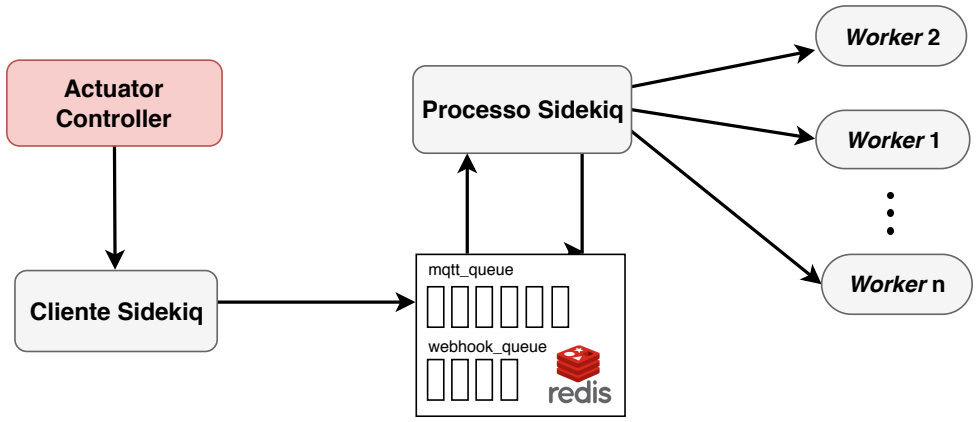

Figura 3.3: Sistema de processamento assíncrono do atuador.

3.2.4 Envio de Comandos via MQTT

Como discutido nos capítulos anteriores, o uso do protocolo de mensageria é um dos pontos-chave para a evolução da plataforma. A abordagem proposta consiste em organizar o sistema de atuação para que seja possível a escolha tanto do método tradicional (*webhook*) quanto o novo formato de mensagem baseada em *pub/sub*. Essa decisão foi tomada com o objetivo de contemplar os requisitos não-funcionais do InterSCity[9] e manter a base de código legado dos usuários ainda funcional. Disponibilizar a comunicação de maneira

híbrida com chamadas REST e *brokers* de mensagem é uma solução já abordada no meio acadêmico [17] e também comercial, em plataformas IoT como o DeviceHub[6].

Servidor

Para inserir o *broker* na plataforma, optamos pelo Eclipse Mosquitto [18], uma implementação *open-source* (EPL/EDL) do protocolo MQTT que faz parte da Eclipse Foundation[7]. Com o intuito de seguir a padronização de gerência de configuração e dependências do InterSCity, uma instância do servidor Mosquitto foi adicionada na imagem Docker do *gateway* da plataforma, juntamente com o servidor do RabbitMQ. Essa instância será responsável por receber mensagens do Actuator Controller contendo comandos, e então repassá-los aos clientes conectados. Tanto o servidor como o cliente *publisher* foram configurados com o nível de qualidade de serviço *at least once* (**QoS 1**), para garantir a entrega da mensagem ao cliente *subscriber*.

Tipo de subscrição

Visando manter a compatibilidade com projetos atuais e criar certa flexibilidade na plataforma, o uso do MQTT ou *webhook* pode ser escolhido pelo usuário no momento do cadastro da subscrição por meio do campo `"tipo"`, adicionado na nova implementação e que pode tomar os valores de `"webhook"`ou `"mqtt"`. No primeiro caso, o funcionamento continua o mesmo e a URL de destino deve ser informada. Já no segundo, não é necessário informar uma URL específica, e este campo será preenchido no registro com o endereço do servidor do MQTT.

Estrutura de tópicos

Enquanto no modelo de *webhook* o dispositivo deve manter um servidor local com um endereço previamente cadastrado no InterSCity aguardando o recebimento de requisições da plataforma, a nova implementação permite que os dispositivos se conectem por meio de um cliente MQTT especificando o tópico a ser monitorado. Os comandos são publicados pelo Actuator Controller no *broker* da plataforma em tópicos que seguem a estrutura abaixo:

```
hostname: mqtt://interscity.org:1883
topic: actuator/{resource_uuid}/capability/{capability_name}
```

Uma das vantagens do modelo *pub/sub* nessa aplicação é que por mais que existam várias subscrições referentes à um mesmo recurso/capacidade, a mensagem precisa ser

[6] `https://www.devicehub.net/`
[7] `https://eclipse.org/`

publicada no *broker* apenas uma vez, enquanto com o uso de chamadas HTTP o comando seria enviado N vezes de acordo com o volume de subscrições repetidas.

O uso do MQTT também permite a assinatura de tópicos "curingas"(*wildcard*). Caso seja necessário monitorar os comandos de **todas** as capacidades de certo recurso, basta se conectar ao *broker* assinando o seguinte tópico:

```
actuator/{resource_uuid}/capability/#
```

Dessa forma, todas as mensagens destinadas ao atuador serão capturadas independente da capacidade.

As Figuras 3.5 e 3.4 apresentam exemplos da etapa de envio para os dois tipos de subscrição.

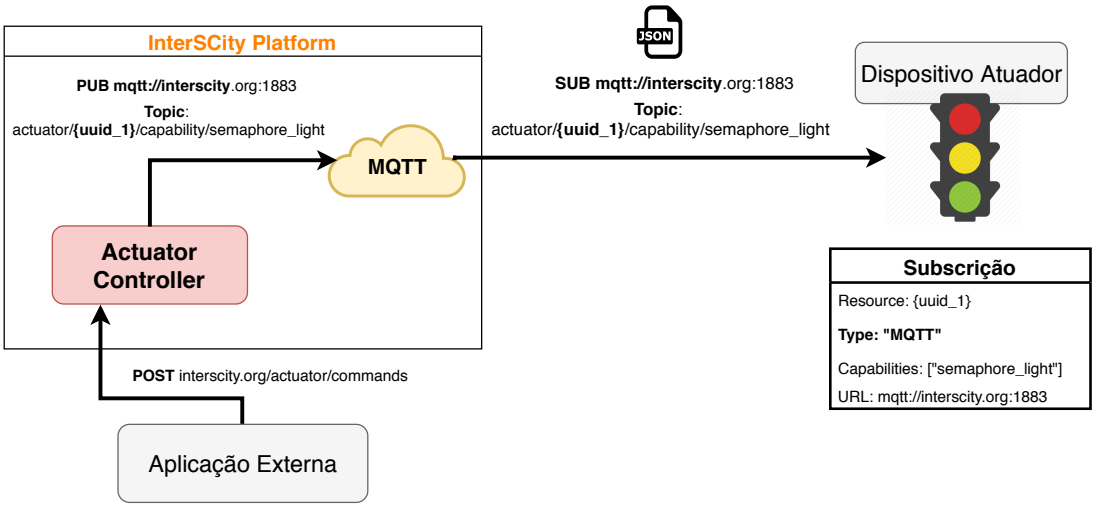

Figura 3.4: Funcionamento do sistema via MQTT.

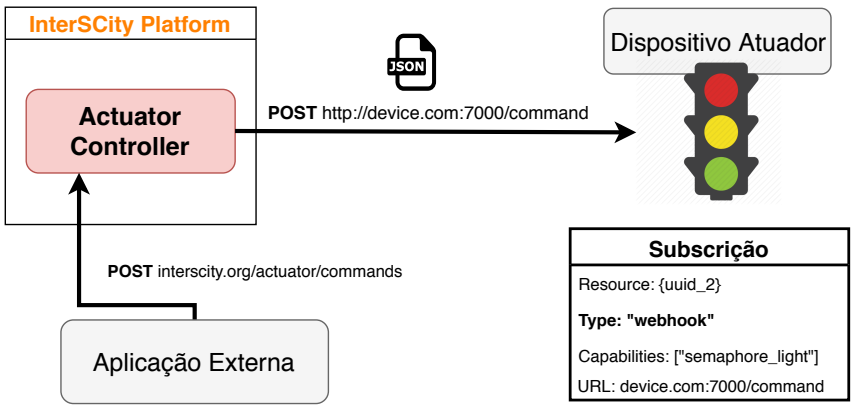

Figura 3.5: Funcionamento do sistema via *webhook*.

Capítulo 4

Testes e Resultados

Neste capítulo serão apresentados os testes aplicados nas duas versões do módulo de atuação e seus resultados, com o objetivo de comparar a performance e eficiência do sistema de envio de comandos. A nova arquitetura propõe soluções que deverão simplificar o processo realizado dentro do InterSCity assim como otimizar a comunicação externa com os recursos no que diz respeito ao tempo de resposta e carga total trafegada.

Para compor os testes, os seguintes indicadores foram selecionados:

1. Tempo de processamento do comando na plataforma
2. Tempo de entrega do comando ao dispositivo
3. Uso de dados no recebimento do comando no dispositivo

A partir desses fatores, podemos analisar o impacto das alterações propostas nos processos internos e externos do fluxo de atuação e atestar se as premissas apresentadas no Capítulo 3 são verdadeiras.

O código-fonte das duas versões da plataforma pode ser acessado publicamente pelo GitLab do InterSCity[1] e pelo repositório deste trabalho[2].

4.1 Testes

O cenário proposto foi construído com o objetivo de simular um ambiente de atuação simples do InterSCity, o qual precisa contemplar tanto a plataforma sendo executada em nuvem como um dispositivo IoT. Além disso, para recriar as condições reais de implantação do sistema o módulo atuador deve estar rodando em uma máquina diferente do restante da

[1] `https://gitlab.com/interscity/interscity-platform`
[2] `https://gitlab.com/SkiaBr23/interscity-dev`

plataforma, seguindo as lógicas de independência e escalabilidade definidas no InterSCity [10].

Foi escolhido o serviço de computação em nuvem da *Amazon Web Services* (AWS)[3] para os testes deste trabalho, por se tratar de uma alternativa utilizada em larga escala no mercado por empresas como a Netflix, Spotify, AirBnB e muitas outras. Além de oferecer máquinas em diversos locais de rede espalhados pelo mundo, o serviço dispõe de pacotes de uso gratuito para estudantes acadêmicos com o intuito de facilitar trabalhos como este. Utilizando uma instância do *Elastic Compute Cloud* (EC2) foi efetuado o *deploy* do módulo de atuação, e o restante dos módulos do InterSCity foram implantados em uma outra instância separadamente.

Para simular o dispositivo atuador selecionamos o Raspberry Pi Zero W, um computador de placa única que vem sendo aplicado em vários ecossistemas de cidades inteligentes [14] e que foi desenvolvido tendo como objetivo entregar uma placa de baixo custo, alta performance e de fácil uso para ser inserido em meios de aprendizado de Ciência da Computação e eletrônica. Diferente de alternativas de microcontroladores e placas de prototipagem como o Arduino[4], a pequena máquina criada pela Raspberry Pi Foundation[5] possui o próprio sistema operacional baseado em Linux e capacidade de conexão Wi-Fi. Sua versão Zero W é a mais compacta, tornando sua aplicação em campo mais flexível e simples. A Tabela 4.1 apresenta algumas especificações técnicas das máquinas utilizadas nos testes, e a Figura 4.1 ilustra a topologia de comunicação entre elas.

Tabela 4.1: Especificação das máquinas utilizadas

	Máquina	**CPU**	**RAM**
Actuator Controller	AWS EC2 t2.small	Intel(R) Xeon(R) E5-2686 2.3 GHz	2 GB
InterSCity Platform	AWS EC2 t2.small	Intel(R) Xeon(R) E5-2686 2.3GHz	2 GB
Actuator IoT Device	Raspberry Pi Zero W	ARM11 Broadcom 1GHz	512 MB

4.1.1 Cenário de Teste

Os testes aplicados consistem no envio de comandos nas duas versões da plataforma para que sejam analisados **(1)** o tempo de processamento da requisição dentro da plataforma, **(2)** o tempo de entrega do comando ao dispositivo IoT, e **(3)** o tamanho total da mensagem a ser trafegada.

Para identificar os pontos de interesse no processamento do comando foram inseridos registros de tempo no *log* dos módulos em cada etapa do fluxo de atuação (anteriormente

[3] `https://aws.amazon.com/`
[4] `https://www.arduino.cc/`
[5] `https://www.raspberrypi.org/`

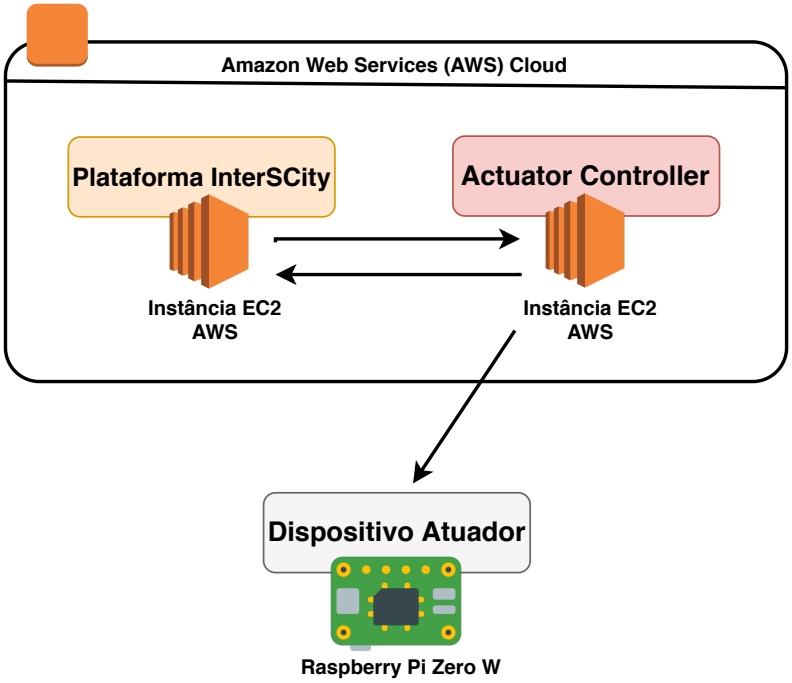

Figura 4.1: Topologia do ambiente de testes.

detalhados nas Figuras 3.1 e 3.2), capturando sua *timestamp*. Nas duas implementações, esses pontos são definidos em: recebimento, criação, envio, entrega e atualização do comando, como ilustrado na Figura 4.2. Ao gravar o registro de cada etapa, podemos identificar o processo interno (da fase de criação até o momento da atualização) e externo (tempo de entrega) da plataforma.

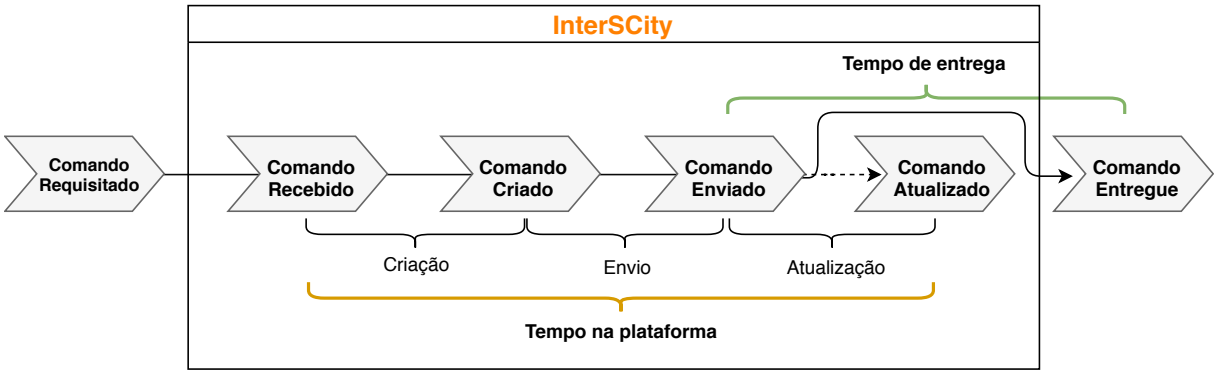

Figura 4.2: Etapas registradas no teste de performance.

Nos testes referentes ao recebimento de comandos via *webhook*, foi configurado na Raspberry Pi um pequeno servidor HTTP utilizando o *framework* Flask[6], uma solução

[6] `http://flask.pocoo.org/`

leve e ágil baseada em Python. Para os testes de recebimento via MQTT empregamos a biblioteca Eclipse Paho[7], também para Python.

A ferramenta de análise de tráfego de rede TCPDUMP[8] foi aplicada nos testes do dispositivo para acompanhar o recebimento dos pacotes e registrar dados como seu tamanho total em *bytes*, códigos de resposta e outros.

4.2 Resultados

Os resultados aqui apresentados foram obtidos através da execução do cenário de simulação proposto situado em um contexto de envio de **1000** comandos com uma latência de **100ms** para um dispositivo atuador previamente cadastrado na plataforma. As três versões apresentadas nos resultados abaixo se referem ao **(1)** envio via *webhook* atual (Webhook V1), **(2)** a nova implementação proposta do envio via *webhook* (Webhook V2), e finalmente **(3)** a emissão do comando através do uso de protocolo de mensageria (MQTT V2).

4.2.1 Desempenho Interno da Plataforma

A partir dos registros obtidos durante os testes de cada etapa do processamento interno do comando, foi possível traçar o gráfico apresentado na Figura 4.3 que representa o tempo total (criação + envio + atualização) proveniente de cada requisição. A duração desse processamento em ambas as estratégias da nova implementação, quando comparada com a atual, chegou a ser em média mais de **50% menor**, reduzindo principalmente o intervalo referente ao envio e atualização do comando. Esse ganho deve-se essencialmente pelo fato de que a atual abordagem executa os passos referidos em módulos distintos, dependendo de comunicação via RabbitMQ para executar funcionalidades que agora são nativas do *Actuator Controller* na versão proposta. A Tabela 4.2 traz os dados dos tempos médios observados, presentes também na Figura 4.4 onde pode se observar a evolução dos intervalos referentes à cada etapa, agrupados por sua implementação, e o desvio padrão de cada amostra representado pelo traçado em preto.

[7] `https://www.eclipse.org/paho/`
[8] `http://www.tcpdump.org/`

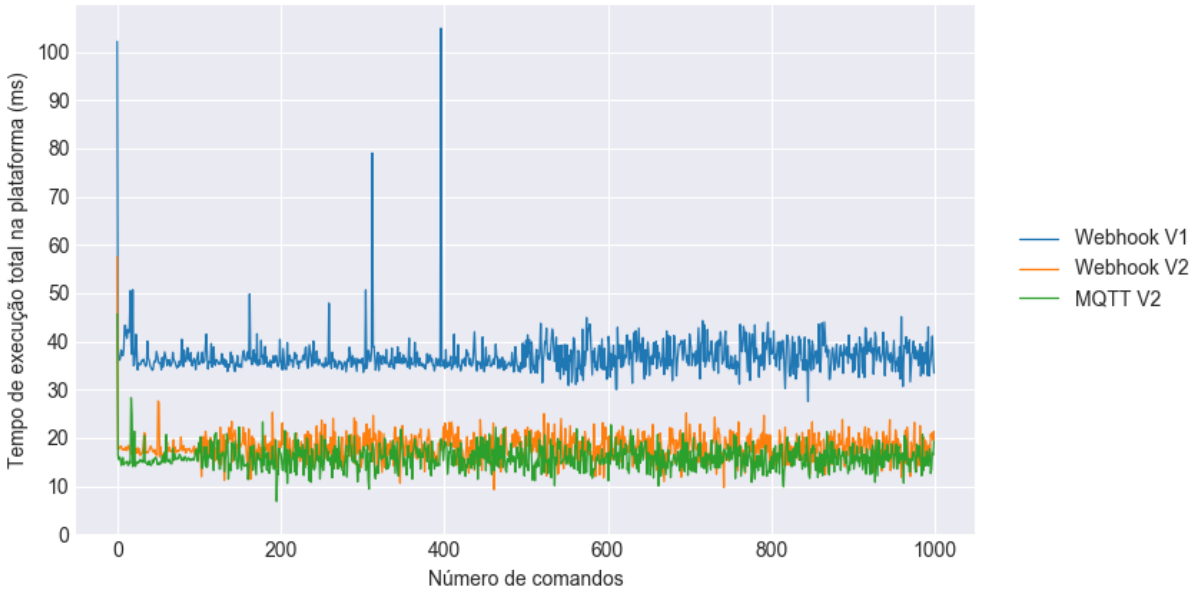

Figura 4.3: Tempo de processamento total do comando na plataforma.

Tabela 4.2: Tempo médio dos procedimentos internos

	Webhook V1	**Webhook V2**	**MQTT V2**
Criação (ms)	3.97	3.73	3.78
Envio (ms)	17.39	10.23	6.76
Atualização (ms)	15.54	4.05	5.35
Tempo total na plataforma	36.9	18.01	15.89

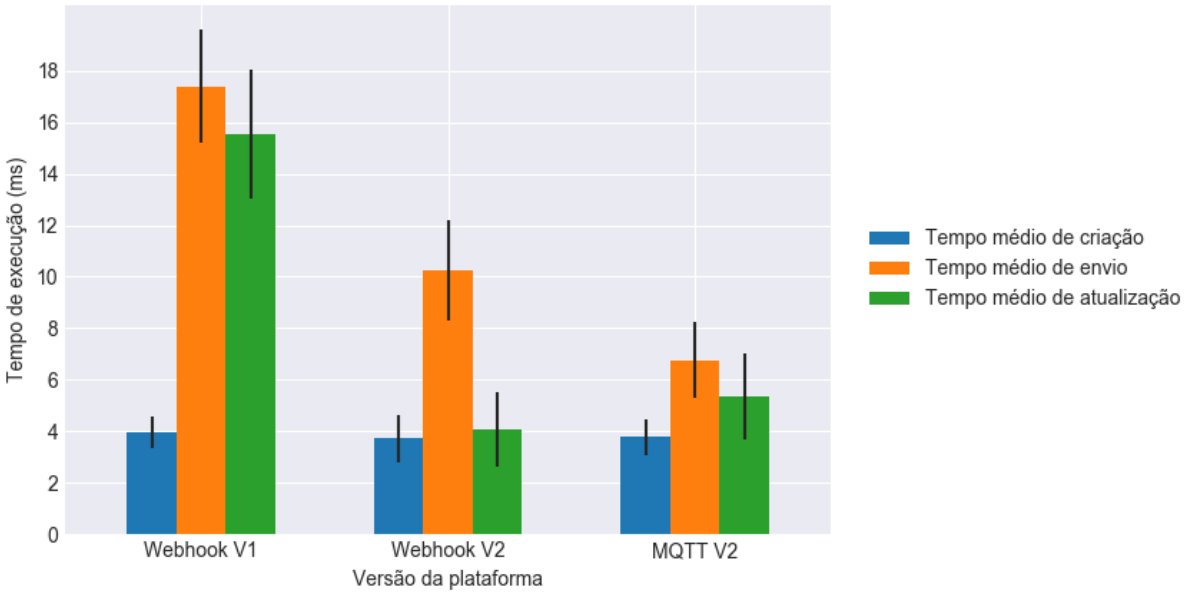

Figura 4.4: Tempo médio de processamento em cada etapa.

No contexto de uma malha extensa de dispositivos se comunicando com o InterSCity, é de suma importância que o trabalho exercido pelo sistema seja o mais eficiente possível para que o volume de dados recebido não tenha nenhum tipo de atraso e perda, além de evitar o gasto com mais infraestrutura ao necessitar de mais máquinas rodando em paralelo.

4.2.2 Tempo de Entrega

Após todo o processo executado no *Actuator Controller* detalhado acima, o comando foi então enviado para o receptor físico do teste. Por meio da mesma abordagem da plataforma utilizamos de registros de *timestamp* na Raspberry Pi para identificar o tempo exato do recebimento do comando, para as três implementações analisadas neste trabalho.

Como se tratam de duas partes (InterSCity e dispositivo) sendo executadas em diferentes máquinas físicas, foi necessário que o relógio dos dois sistemas fossem sincronizados para evitar erros de defasagem no momento do experimento. Para isso, foi empregado o *Network Time Protocol* (NTP)[9], que fornece um meio de trocar informações de tempo entre diferentes entidades em rede. Apesar de não entregar uma sincronização de alta precisão à nivel de milissegundos, podemos obter periodicamente o *delay* residual entre os pares e ajustar os dados obtidos.

[9] `http://www.ntp.org/`

Com os registros das etapas internas à plataforma, o tempo de entrega total de cada requisição foi calculado comparando as *timestamps* do momento de envio pelo *Actuator Controller* e a de recebimento na Raspberry Pi. Nos casos de uso de *webhook*, por se tratar de tráfego HTTP, foi observado um tempo elevado nas primeiras requisições antes de estabilizar em um valor médio. As Figuras 4.5 e 4.6 apresentam os tempos registrados nos primeiros 10 comandos recebidos e os restantes do teste, respectivamente.

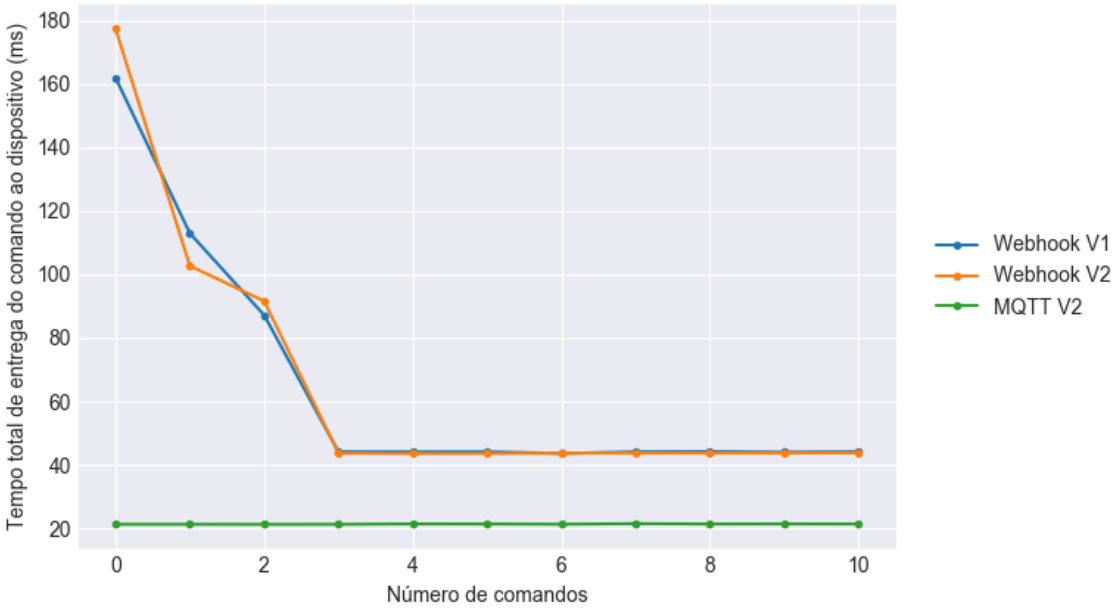

Figura 4.5: Tempo em milissegundos da entrega nos primeiros 10 comandos enviados.

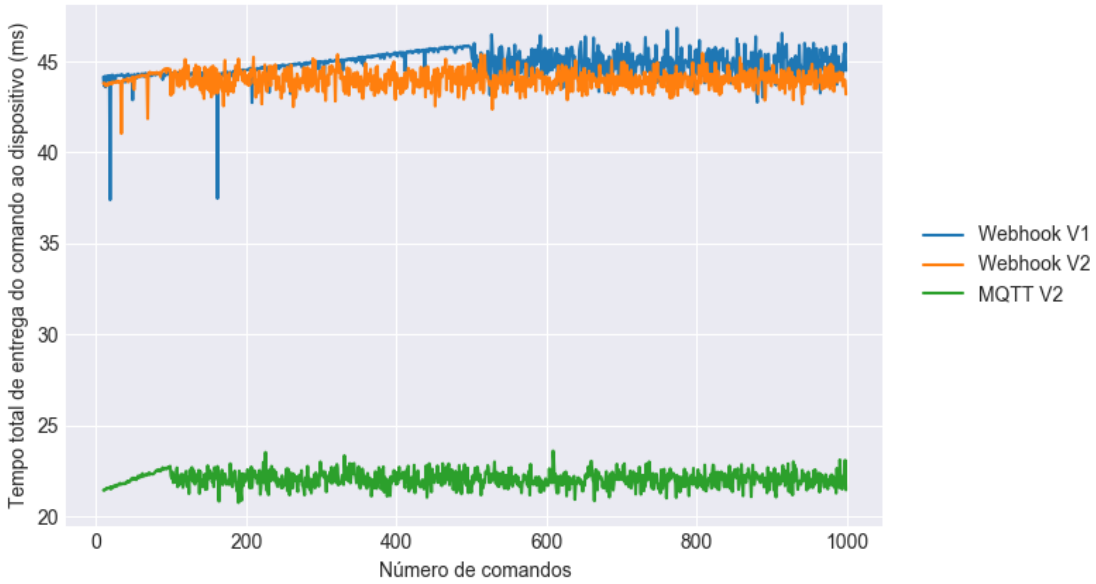

Figura 4.6: Tempo em milissegundos da entrega dos comandos após estabilização.

Como esperado, os tempos referentes às duas implementações do envio via *webhook* não apresentaram diferenças significativas, pois o processo se manteve o mesmo no que diz respeito à fase de entrega. Já nas subscrições que utilizaram MQTT, em todo o período de testes houve um recebimento com redução expressiva no tempo e uma maior estabilidade durante as 1000 iterações. Em média, a nova implementação proposta obteve um ganho de **51,1%** na resposta quando comparado com a atual abordagem. Os valores médios observados, assim como seus desvios e picos, foram calculados e dispostos na Tabela 4.3.

Tabela 4.3: Tempo médio de entrega do comando registrados no teste

	Webhook V1	**Webhook V2**	**MQTT V2**
Tempo médio (ms)	45.05	44.26	22.03
Desvio Padrão	4.54	4.86	0.42
Mínimo (ms)	37.39	41.04	20.75
Máximo (ms)	161.88	177.30	23.59

4.2.3 Tempo Total

Com base nos resultados detalhados acima de cada etapa, podemos finalmente contabilizar o tempo total útil do processo, representado pelo momento da requisição de um comando

até sua entrega no dispositivo. Neste momento, o tempo de atualização é ignorado por se tratar de uma tarefa que ocorre de forma paralela à entrega e não é perceptível para o usuário da plataforma. A Tabela 4.4 apresenta os intervalos de tempo médio total (criação + envio + entrega) para cada implementação, assim como seus desvios e picos. A nova implementação se mostrou mais eficiente que a versão atual em ambos os métodos, com reduções de aproximadamente **12,3%** para as subscrições via *webhook* e **51%** nos recebimentos por meio do MQTT.

A Figura 4.7 traz uma visualização acumulada dos valores médios de tempo para cada etapa, com o objetivo de comparar as diferentes evoluções alcançadas após a aplicação das alterações propostas neste trabalho.

Tabela 4.4: Tempo total médio da requisição até o recebimento do comando

	Webhook V1	**Webhook V2**	**MQTT V2**
Tempo médio total (ms)	66.41	58.22	32.56
Desvio Padrão	5.61	5.96	1.72
Mínimo (ms)	59.30	50.91	26.88
Máximo (ms)	208.19	217.14	48.38

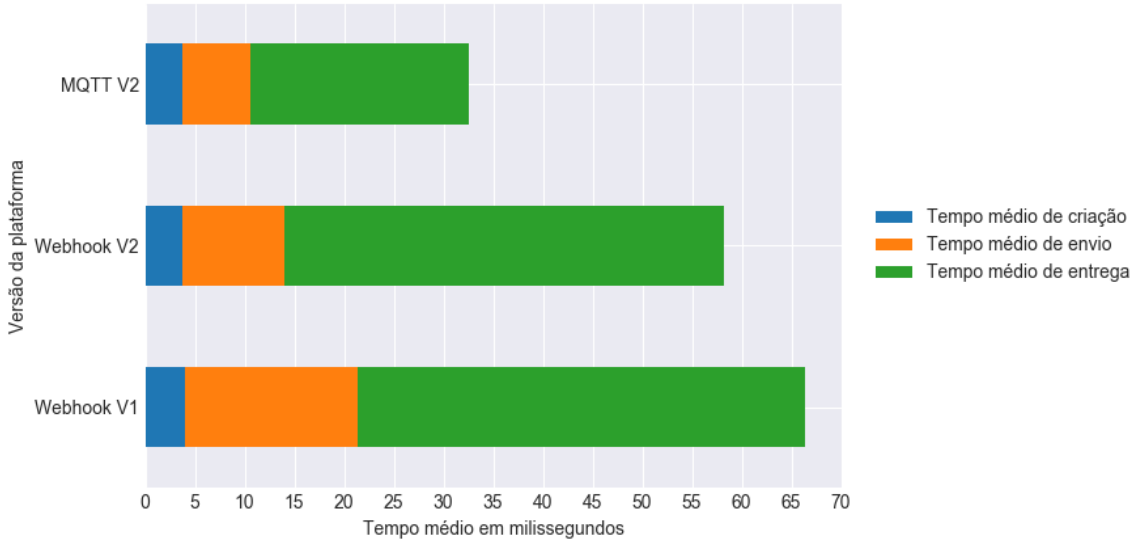

Figura 4.7: Tempo médio das etapas de criação, envio e entrega.

4.2.4 Uso de Dados

Um dos pontos de atenção em aplicações IoT é o uso de dados nos dispositivos em campo, devido à possíveis limitações de rede como acesso, força do sinal, largura de banda e outros. Para estimar o volume de dados trafegado durante o recebimento de um comando, capturamos com o uso da ferramenta TCPDUMP a atividade de rede na Raspberry Pi no decorrer de uma amostra de 100 comandos requisitados, especificando as portas usadas por cada aplicação. Foi efetuado apenas o teste da abordagem via *webhook* da atual plataforma que utiliza o HTTP e da nova implementação do MQTT, para comparar o impacto dos diferentes protocolos.

Por meio da análise do comportamento dos pacotes trafegados, foi possível estimar os tamanhos reais em *bytes* dos pacotes recebidos/enviados durante o processo de chegada do comando. A Tabela 4.5 aponta os valores observados.

Tabela 4.5: Tamanho em bytes das partes de um comando recebido

	Tamanho em bytes	
	HTTP	**MQTT**
Conexão	214	753
Payload	553	360
Header	132	66
Requisições adicionais	621	66
Total de dados / comando	1520	1245/492*

Pelo fato de utilizar um sistema de conexão persistente, o pacote de conexão do MQTT se mostrou aproximadamente 71,5% maior que seu correspondente no protocolo HTTP. Porém, a conexão do MQTT é mantida durante todo o tempo dos testes, fazendo com que esse pacote seja trafegado apenas uma vez. O tamanho do cabeçalho e *payload* entre as duas abordagens também é díspar, mostrando a eficiência do pacote MQTT em termos de dados transmitidos.

Após o envio de 100 comandos, registramos a movimentação de pacotes e as informações obtidas podem ser visualizadas na Tabela 4.6. O total de uso de dados foi aproximadamente **66,7%** menor na versão proposta, o que demonstra o impacto substancial que a troca de protocolos pode exercer em uma aplicação real de cidades inteligentes onde o volume de informações sendo transmitidas é de grande vazão.

Tabela 4.6: Dados obtidos pelo TCPDUMP durante o envio de 100 comandos para o dispositivo

	HTTP	**MQTT**
Comandos recebidos	100/100	100/100
Número de pacotes / conexão	13	13
Número de pacotes / comando	12	2
Total de pacotes	1200	200
Total de dados trafegados (bytes)	152000	49953

A redução dos dados trafegados observada ao adotar o método proposto por este trabalho tende a impactar ainda mais na economia de rede do dispositivo em situações nas quais o atuador recebe comandos com alta frequência, o que pode permitir o dimensionamento mais eficiente da banda alocada aos equipamentos em campo e reduzir o custo com infraestrutura do projeto de implantação dos mesmos.

Capítulo 5

Considerações finais

Neste trabalho, a camada de controle de atuação da plataforma do InterSCity foi analisada com o intuito de propor uma nova versão que atenda requisitos de projetos IoT. A partir dessa análise foi possível elencar os principais problemas presentes na implementação atual, que giravam em torno da organização interna dos módulos da plataforma e sua abordagem de comunicação externa. Com esse insumo, propomos mudanças arquiteturais que visaram otimizar a troca de informações interna do sistema e o acesso às suas bases de dados, além de adicionar uma nova alternativa de envio de comandos à dispositivos atuadores que utiliza o protocolo de mensageria MQTT.

O objetivo da proposta foi tornar os processos internos e externos do fluxo de atuação mais eficientes no que tange seu tempo de processamento e uso de dados consumidos na rede pelo dispositivo atuador. Para validar a arquitetura sugerida neste trabalho, foram executados testes de caso para comparar as duas versões da plataforma em um cenário simples de atuação composto por instâncias do InterSCity em máquinas na nuvem e uma Raspberry Pi Zero W exercendo o papel do dispositivo atuador.

Os resultados obtidos atenderam às expectativas, mostrando ganhos de desempenho significativos no processamento interno após as alterações arquiteturais. Além disso, o uso do protocolo MQTT no envio dos comandos trouxe uma redução no tempo de entrega em mais de 50%, e otimizou o volume de dados consumido no dispositivo em aproximadamente 67%. Essas reduções implicam diretamente no desempenho final da plataforma à medida que a solução for escalada para várias aplicações.

As contribuições deste trabalho podem auxiliar o InterSCity a lidar com um volume maior de requisições ao controle de atuação por reduzir o custo de processamento do módulo, e trazer benefícios aos usuários da plataforma ao entregar os comandos com uma menor latência. A versão do módulo desenvolvido neste trabalho será submetida no repositório do InterSCity por meio de uma *Pull Request* com o objetivo de contribuir com o crescimento e evolução do projeto.

Como trabalhos futuros, sugereimos que o novo módulo de atuação possa ser evoluído acrescentando o uso de outras funcionalidades disponíveis no MQTT, como criptografia, mensagens retidas e *wildcards*. Com base nos estudos aplicados durante o desenvolvimento e os resultados observados, a utilização do MQTT se mostrou de grande impacto na comunicação entre a plataforma e o dispositivo, e uma outra sugestão de evolução futura seria a implementação desse protocolo também no envio de dados de sensores para otimizar ainda mais a plataforma.

Referências

[1] Nicholas, Stephen: *Power Profiling: HTTPS Long Polling vs. MQTT with SSL, on Android*, 2012. `http://stephendnicholas.com/posts/power-profiling-mqtt-vs-https`, acesso em 2018-11-05. x, 14, 15

[2] Jin, Jiong, Jayavardhana Gubbi, Slaven Marusic e Marimuthu Palaniswami: *An information framework for creating a smart city through internet of things*. IEEE Internet of Things Journal, 1(2):112–121, 2014, ISSN 23274662. 1

[3] Batty, M., K. W. Axhausen, F. Giannotti, A. Pozdnoukhov, A. Bazzani, M. Wachowicz, G. Ouzounis e Y. Portugali: *Smart cities of the future*. European Physical Journal: Special Topics, 214(1):481–518, 2012, ISSN 19516355. 1

[4] Gutiérrez, Verónica, Jose A. Galache, Luis Sańchez, Luis Munõz, José M. Hernández-Muñoz, Joao Fernandes e Mirko Presser: *SmartSantander: Internet of things research and innovation through citizen participation*. Em *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7858 LNCS, páginas 173–186. Springer, Berlin, Heidelberg, 2013, ISBN 9783642380815. `http://link.springer.com/10.1007/978-3-642-38082-2{_}15`. 1

[5] Chui, Michael, Susan Lund, Anu Madgavkar, Jan Mischke e Sree Ramaswamy: *Smart Cities: Digital solutions for a more livable future*. 2018. 1

[6] Misra, Gourav, Vivek Kumar, Arun Agarwal e Kabita Agarwal: *Internet of Things (IoT) – A Technological Analysis and Survey on Vision, Concepts, Challenges, Innovation Directions, Technologies, and Applications (An Upcoming or Future Generation Computer Communication System Technology)*. American Journal of Electrical and Electronic Engineering, Vol. 4, 2016, Pages 23-32, 4(1):23–32, 2016. `http://pubs.sciepub.com/ajeee/4/1/4`. 1

[7] Lin, Jie, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang e Wei Zhao: *A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications*. IEEE Internet of Things Journal, 2017, ISSN 23274662. 2

[8] Atzori, Luigi, Antonio Iera e Giacomo Morabito: *The Internet of Things: A survey*. Computer Networks, 54(15):2787–2805, 2010, ISSN 13891286. 2

[9] M. Del Esposte, Arthur, Fabio Kon, Fabio M. Costa e Nelson Lago: *InterSCity: A Scalable Microservice-based Open Source Platform for Smart Cities*. Em *Proceedings of*

the 6th International Conference on Smart Cities and Green ICT Systems, páginas 35–46, 2017, ISBN 978-989-758-241-7. http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006306200350046. 2, 4, 5, 6, 7, 9, 22

[10] Batista, Daniel Macedo, Alfredo Goldman, Roberto Hirata, Fabio Kon, Fabio M. Costa e Markus Endler: *InterSCity: Addressing Future Internet research challenges for Smart Cities*. Em *2016 7th International Conference on the Network of the Future (NOF)*, páginas 1–6, 2016, ISBN 978-1-5090-4671-3. http://ieeexplore.ieee.org/document/7810114/. 2, 5, 17, 26

[11] Dragoni, Nicola, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin e Larisa Safina: *Microservices: Yesterday, Today, and Tomorrow*. Em *Present and Ulterior Software Engineering*, páginas 195–216. 2017, ISBN 9783319674254. http://link.springer.com/10.1007/978-3-319-67425-4{_}12. 4

[12] Jefferson, Dylan, Maurício Guimarães, Guedes Orientador, Paulo Roberto e Miranda Meirelles: *Processamento de dados em uma plataforma de cidades inteligentes*. 2017. 7, 8

[13] OASIS: *MQTT Version 3.1.1*, 2014. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf, acesso em 2018-07-09. 11, 13

[14] Al-Fuqaha, Ala, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari e Moussa Ayyash: *Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications*. IEEE Communications Surveys and Tutorials, 17(4):2347–2376, 2015, ISSN 1553877X. 11, 13, 26

[15] Singh, Meena, M. A. Rajan, V. L. Shivraj e P. Balamuralidhar: *Secure MQTT for Internet of Things (IoT)*. Em *Proceedings - 2015 5th International Conference on Communication Systems and Network Technologies, CSNT 2015*, páginas 746–751, 2015, ISBN 9781479917976. 13

[16] Whitmore, Andrew, Anurag Agarwal e Li Da Xu: *The Internet of Things—A survey of topics and trends*. Information Systems Frontiers, 17(2):261–274, 2015, ISSN 15729419. 19

[17] Collina, Matteo, Giovanni Emanuele Corazza e Alessandro Vanelli-Coralli: *Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST*. Em *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC*, páginas 36–41, 2012, ISBN 9781467325691. 23

[18] A Light, Roger: *Mosquitto: server and client implementation of the MQTT protocol*. The Journal of Open Source Software, 2(13):265, may 2017, ISSN 2475-9066. http://joss.theoj.org/papers/10.21105/joss.00265. 23