Recarga Distribuída de Veículos

Caick Wendell Lopes dos Santos[1], **João Lucas Silva Serafim Silva**[1]

[1]Departamento de Tecnologia – Universidade Estadual de Feira de Santana (UEFS)
Av. Transnordestina, s/n, Novo Horizonte / Feira de Santana – BA, Brasil- 44036-9

Resumo.O aumento no número de usuários de carros elétricos é uma tendência crescente.No entanto, a incerteza da disponibilidade de pontos de recarga em trajetos longos desencoraja o uso desse tipo de veículo em trajetos extensos.Este documento apresenta o desenvolvimento de um sistema de recarga distribuído que permite a reserva de postos ao longo de um trajeto.O sistema utiliza os protocolos *Message Queue Telemetry Transport*(MQTT) e *Hypertext Transfer Protocol*(HTTP) para comunicações cliente-servidor e servidor-servidor.A aplicação permite o planejamento de rotas por parte do usuário em conjunto com o gerenciamento de disponibilidade através da comunicação entre servidores.

Abstract. The increasing number of electric vehicle users is a growing trend. However, the uncertainty of charging point availability on long journeys discourages the use of this type of vehicle for extensive travel. This document presents the development of a distributed charging system that allows the reservation of stations along a route. The system utilizes the Message Queue Telemetry Transport(MQTT) and Hypertext Transfer Protocol(HTTP) protocols for client-server and server-server communications. The application allows users to plan routes in conjunction with availability management through communication between servers.

1. Introdução

A aquisição de um veículo elétrico está totalmente atrelada ao custo-benefício que o produto apresenta.Portanto, a necessidade ou desejo de realizar viagens longas é um fator decisivo ao considerar a compra.O aumento do público alvo pode ser comprometido devido a complexidade que o planejamento de um percurso extenso pode trazer. Empresas distintas possuem sistemas independentes e dificultam o gerenciamento de rotas para o usuário.

O documento a seguir contém a descrição dos meios e ferramentas utilizadas para o aprimoramento de um sistema de recarga inteligente. Com o objetivo de torná-lo viável para o planejamento de rotas e expandir a antiga proposta de recargas individuais.A aplicação utiliza majoritariamente a linguagem *Python* em conjunto com protocolo *Message Queue Telemetry Transport*(MQTT) com broker Mosquitto e arquitetura *Rest*.Além disso foi projetada em sistemas Linux com o auxílio da IDE Visual Studio Code, Insomnia e plataforma GitHub.

O sistema conta com todas as funcionalidades de sua versão anterior, juntamente com a possibilidade de gerenciar rotas e realização de requisições atômicas entre servidores.Este relatório está organizado da seguinte forma.A seção 2 contém o conhecimento teórico utilizado para o desenvolvimento da aplicação.A seção 3 apresenta as ferramentas e métodos utilizados.A seção 4 demonstra os resultados e falhas obtidos.Por último a seção 5 refere-se as conclusões obtidas ao término do projeto.

2. Fundamentação Teórica

Esta seção é dedicada a uma breve introdução dos principais conceitos utilizados para desenvolvimento do projeto e tem como foco o protocolo MQTT e a arquitetura REST.

2.1. *Message Queue Telemetry Transport*(MQTT)

Desenvolvido pela IBM e Eurotech na década de 90, o protocolo MQTT tinha por objetivo enviar dados de forma precisa sob as más condições de rede da época[Soni and Makwana 2017].

Baseado no modelo de publicação/assinatura, o protocolo em questão conta com um intermediário, chamado *broker*, o qual se responsabiliza por envias as mensagens[Torres et al. 2016].O cliente, quando publicador, envia sua mensagem ao *broker* que, por sua vez, a enviará para todos os assinantes interessados.

2.2. *Rest*

Representative State Transfer(REST) é um modelo de arquitetura de software criado por Roy Fielding que visa facilitar a comunicação e a interoperabilidade entre diferentes sistemas. A mesma baseia-se em seis restrições, sendo elas:

1. Interface uniforme
2. Ausência de estado
3. Cache
4. Cliente-servidor
5. Sistema de camadas
6. Código sob-demanda(opcional)

2.2.1. Interface uniforme

A restrição de interface uniforme desacopla a arquitetura de forma a definir a interface entre o cliente e o servidor. Recursos individuais são identificados em requisições através de URLs e manipulados pelo cliente por meio de representações dos mesmos.

2.2.2. Ausência de estado

Essa restrição implica que o estado necessário para lidar com uma requisição está contido na própria, seja como parte da URI, requisição, corpo ou cabeçalho. A URI apenas identifica o recurso enquanto o corpo contém o estado[Surwase 2016].

2.2.3. Cache

Clientes podem ter suas respostas gravadas na cache.Porém, elas devem ser definidas como cacheáveis ou não seja de forma implícita ou explícita[JUNIOR and dos Santos 2019].

2.2.4. Cliente-servidor

Servidores e clientes podem ser desenvolvidos independentemente enquanto a interface não for alterada. O cliente não tem qualquer preocupação com persistência, manipulação ou armazenamento de dados e o servidor não lida com o estado do cliente ou sua interface.

2.2.5. Sistema de camadas

Um cliente qualquer não deve ser capaz de discernir sua conexão seja ela entre um servidor ou um intermediário. Servidores intermediários podem aumentar escalabilidade, desempenho e permitir caches múltiplos.

2.2.6. Código sob-demanda

Servidores podem temporariamente estender ou customizar as funcionalidades do cliente ao transferir dados executáveis.

3. Metodologia

A princípio, fizeram-se necessárias discussões e pesquisas sobre as novas tecnologias que seriam implementadas e quais mudanças teriam que ser feitas no sistema antigo para que atendesse as demandas atuais.

Notou-se que a estrutura lógica da aplicação não sofreria grandes impactos em seu funcionamento habitual e que o entendimento do protocolo MQTT e a substituição do mecanismo de envio de mensagens antigo tornou-se prioridade, visto que a comunicação cliente-servidor já fazia parte do sistema.

No entanto, é necessário lembrar que o protocolo MQTT tem como um de seus paradigmas manter uma conexão estável entre as partes envolvidas, as quais trocam mensagens livremente, enquanto que sua aplicação no projeto se dá como forma de realizar requisições e receber respostas. Assim sendo, foi necessário estabelecer uma dinâmica tal como aquela presente no protocolo TCP-IP anterior (o qual é base para os protocolos MQTT e HTTP).

Esta dinâmica se dá na postagem, pelo cliente, de uma mensagem em um tópico referente a todas as requisições a serem atendidas por um determinado servidor (ou seja, múltiplos servidores podem conectar-se ao mesmo *broker* MQTT, caso assim seja desejado), sendo que em seguida o resultado de tal requisição é postado em um tópico referente ao cliente que requisitou a ação, fazendo uso do mesmo *broker* MQTT (aplicação que recebe, armazena e encaminha mensagens) no qual foi lida a mensagem da requisição. O cliente, por sua vez, espera, assim que termina a postagem da requisição, uma resposta em seu tópico e no *broker* MQTT utilizado por este para postar sua requisição, pela resposta do servidor.

O processo de envio de mensagens se dá, em sequência, pela conexão com o *broker* MQTT escolhido, postagem de tal mensagem, e então o encerramento imediato da conexão, enquanto que o recebimento designa uma *callback* quando recebida uma mensagem, seguido da conexão com o *broker* MQTT escolhido, assinatura no tópico

na qual a mensagem será recebida, espera, dentro de um *timeout*, do recebimento de uma mensagem no tópico, remoção da assinatura, encerramento da conexão e retorno da resposta (vazia, caso atinja *timeout*).

De volta à mudanças necessárias de serem implementadas nas funcionalidades já existentes, foi adicionada uma verificação da existência de reservas dentro de uma janela de tempo para a recarga de um veículo (2 horas antes e depois do horário marcado), o qual, se ocupada por um veículo que não aquele que está solicitando serviços de recarga, impossibilita o uso da estação de carga reservada. Tal mudança é mais que suficiente para integrar todo o sistema de reserva multi-servidor ainda a ser implementado com o sistema de recarga existente.

Por outro lado, o protocolo HTTP-REST (utilizado para comunicação entre servidores) já tem em seu cerne o uso como meio de comunicação por requisições, de forma que não foi necessária adaptação alguma para seu uso no projeto. Por outro lado, sua implementação ainda mostrou-se desafiadora na medida que, em se tratando de um protocolo mais sofisticado, é necessário ou o desenvolvimento de um código mais complexo (ao menos no ator que recebe e trata as requisições), ou uso de *frameworks* para abstração e automatização do processo de desenvolvimento.

Inicialmente, a solução para sua implementação foi o uso de *frameworks*, mais especificamente o Django, visto sua ampla utilização pelos profissionais da área de tecnologia. Entretanto, logo foi percebida a dificuldade em integrar um projeto de alto nível (tal como um projeto em Django) com o código ”cru”já existente para a aplicação.

Portanto, foi tomada a decisão de desenvolver código próprio, o qual faz uso da biblioteca http do Python para recebimento e tratamento de requisições.

Assim como acontece na leitura de mensagens dentro do protocolo MQTT, o protocolo HTTP estabelece uma *callback* para ser ativada no evento do recebimento de uma mensagem com determinados parâmetros. A diferença, além da não-existência de um *broker* (sendo as requisições enviadas diretamente à aplicação do servidor que as processarão), é que os parâmetros são o complemento da url de destino (ex: </submit >, como em <http://localhost:8025/submit>) e o tipo da requisição (GET, POST, PUT, PATCH, DELETE, dentre outros), ao invés de um ”tópico”.

A parte de envio das requisições, tendo em visto o fato de que o protocolo HTTP entrega papéis distintos às partes envolvidas, foi implementada utilizando a biblioteca requests do Python. Tal biblioteca envia uma mensagem contendo um cabeçalho (o qual guarda informações como o endereço do remetente, o complemento de url e o método da requisição) e um corpo para o endereço expresso em uma url (ou seja, excluindo seu prefixo e seu complemento) e espera pela resposta dentro de um *timeout*.

Com os protocolos de comunicação devidamente implementados, restou à equipe de desenvolvimento a implementação das novas funcionalidades.

A primeira nova funcionalidade implementada foi a de criação/remoção de reserva de um posto de carga em um determinado servidor, para um veículo de determinado ID. Tal funcionalidade é implementada por dóis métodos separados: doReservation (tenta fazer a reserva para um determinado veículo) e undoReservation (remove a reserva de um determinado veículo de qualquer estação cadastrada no servidor).

Para que um veículo reserva um posto, é necessário que não haja outro veículo reservado no horário desejado e que tal horário esteja no mínimo 2 horas no futuro (valor padrão, verificação feita para evitar que se façam reservas enquanto outros veículos recarregam).

Os dois métodos anteriormente citados são acionados seguindo o recebimento de uma requisição HTTP com determinados parâmetros de método HTTP-REST, complemento de url e corpo da requisição. Na atual implementação da aplicação do servidor, é esperado que o método seja sempre GET, o complemento sempre /submit, e o corpo seja um artefato JSON contendo os parâmetros da ação requisitada ao servidor.

O corpo da requisição para criação de reservas é uma lista que contém a string ”drr”seguida de uma ”sub-lista”contendo a *string* de ID do veículo requisitante, o horário da reserva (formato EPOCH POSIX), a autonomia do veículo, uma coordenada x e uma coordenada y, e espera como resposta uma lista de tamanho 2 contendo as coordenadas x e y do posto reservado (ou vazia, caso a reserva falhe), enquanto que o corpo da requisição para a remoção de reserva é uma lista que contém a string ”drr”seguida de uma ”sub-lista”contendo apenas a *string* de ID do veículo requisitante, não esperando nenhum retorno.

Logo após, foi adicionada uma funcionalidade que responde ao veículo solicitante o índice de uma rota com base no endereço do servidor (localidade) de destino e no índice entre as rotas válidas para tal destino da rota a ser retornada (permitindo ao cliente avançar ou retroceder na lista de rotas que contemplam o destino escolhido).

A seguir, foi implementada a funcionalidade de reserva de rotas. De maneira resumida, a ação de tentativa reserva de rota deve ser resultado da leitura, pelo módulo de comunicação MQTT de um servidor, de uma mensagem contendo o identificador de um veículo, o índice na lista de rotas do servidor da rota desejada, e uma lista dos horários desejados para cada nó da rota (correspondente a um servidor distinto), e retorna para o veículo requisitante se a reserva foi bem-sucedida ou não.

Quanto ao processo de reserva em si, este se dá pelo envio sequencial de requisições http contendo corpo JSON de conteúdo de lista que contém a string ”drr”seguida de uma ”sub-lista”contendo a *string* de ID do veículo requisitante, o horário (formato EPOCH POSIX) desejado para o posto no servidor atendente, a autonomia do veículo, uma coordenada x e uma coordenada y, sendo que as coordenadas são referentes às atuais do veículo na primeira iteração e do posto anteriormente reservado em iterações seguintes, caso bem-sucedidas, até que a lista de servidores referente à rota desejada seja totalmente percorrida.

Caso ocorra algum erro durante o processo de reserva (recebimento de coordenada nulas ou caso não seja possível estabelecer conexão com um servidor), o processo de reserva será interrompido e os servidores anteriores na lista referente à rota (ou seja, aqueles que fizeram a reserva com sucesso) receberão cada um uma requisição para remover a reserva do veículo solicitante.

Para comportar as novas funcionalidades do sistema, a interface gráfica recebeu algumas atualizações. As principais incluem: adição de janelas secundárias responsáveis pelas funcionalidades de recarga e gerenciamento de rotas além da separação da aba de histórico de compra e inserção de caixas de texto que disponibilizem para o usuário a

possibilidade de inserir os servidores de destino, origem e horário de agendamento.

Para implementação das atualizações anteriormente citadas o programa utiliza de funções que criam as janelas secundárias no momento em que seus respectivos botões são acionados, estas janelas tem acesso a todas as informações necessárias uma vez que a janela principal e as *strings* correspondentes aos *labels* nelas presentes são declarados globalmente.

Por fim, os novos elementos gráficos foram integrados a novos métodos da aplicação do veículo referentes a obtenção de uma rota e reserva de rota (como anteriormente esperado), e o sistema foi testado em seu funcionamento completo.

4. Resultados

A seção a seguir é focada nos aprimoramentos implementados no projeto e funcionamento do sistema.

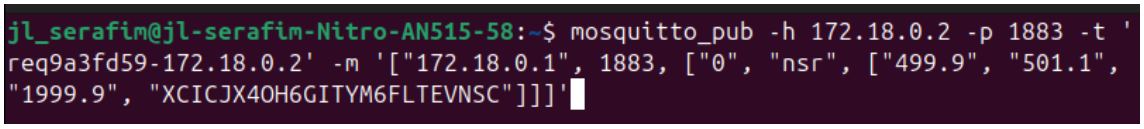

Figura 1. Enviando mensagem por meio de mosquitto-client contendo requisição do veículo com endereço 172.18.0.1 para o *broker* no endereço 172.18.0.2

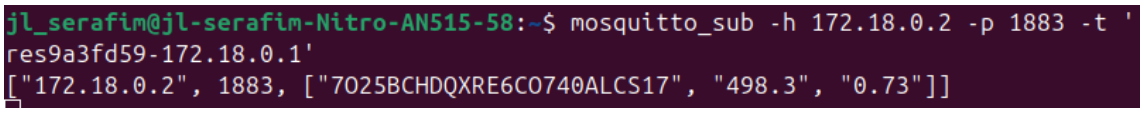

Figura 2. Recebendo resposta por meio de mosquitto-client de uma requisição do veículo com endereço 172.18.0.1 a partir do *broker* no endereço 172.18.0.2

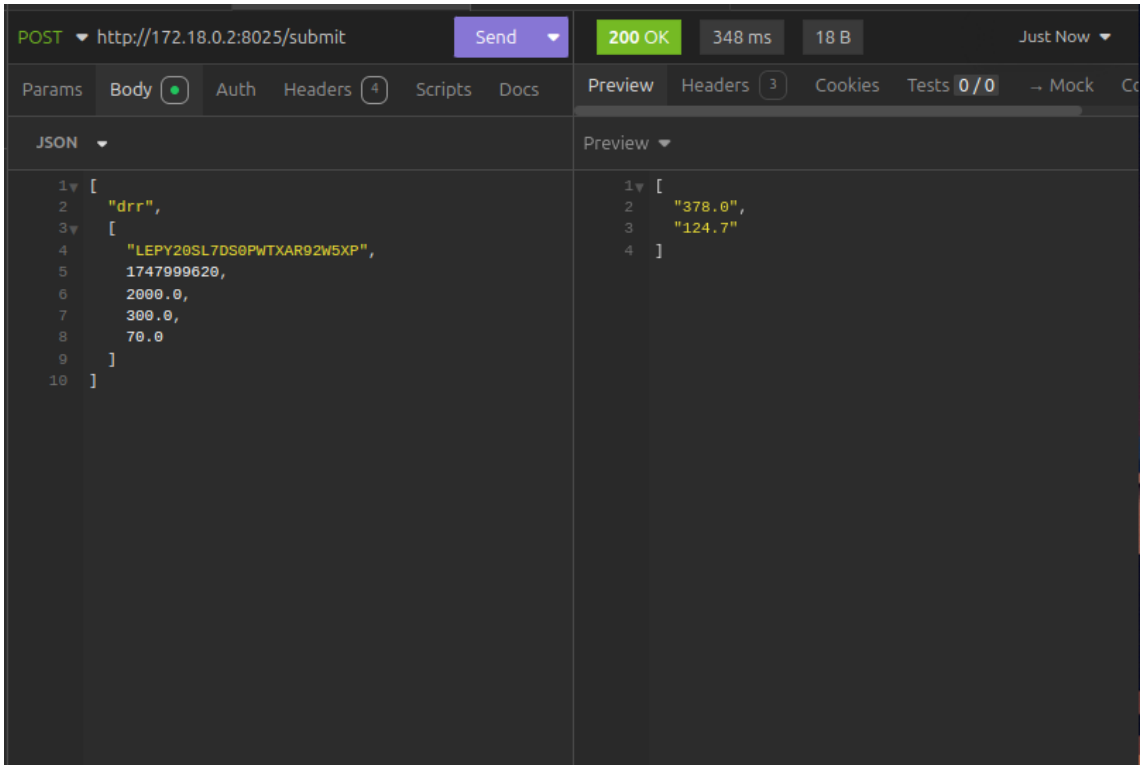

Figura 3. Enviando requisição http para criação de reserva a um servidor utilizando o *software* Insomnia

4.1. Tela principal

Figura 4. Tela principal

4.2. Menu de recarga

Figura 5. Menu de recarga

4.3. Menu de reserva

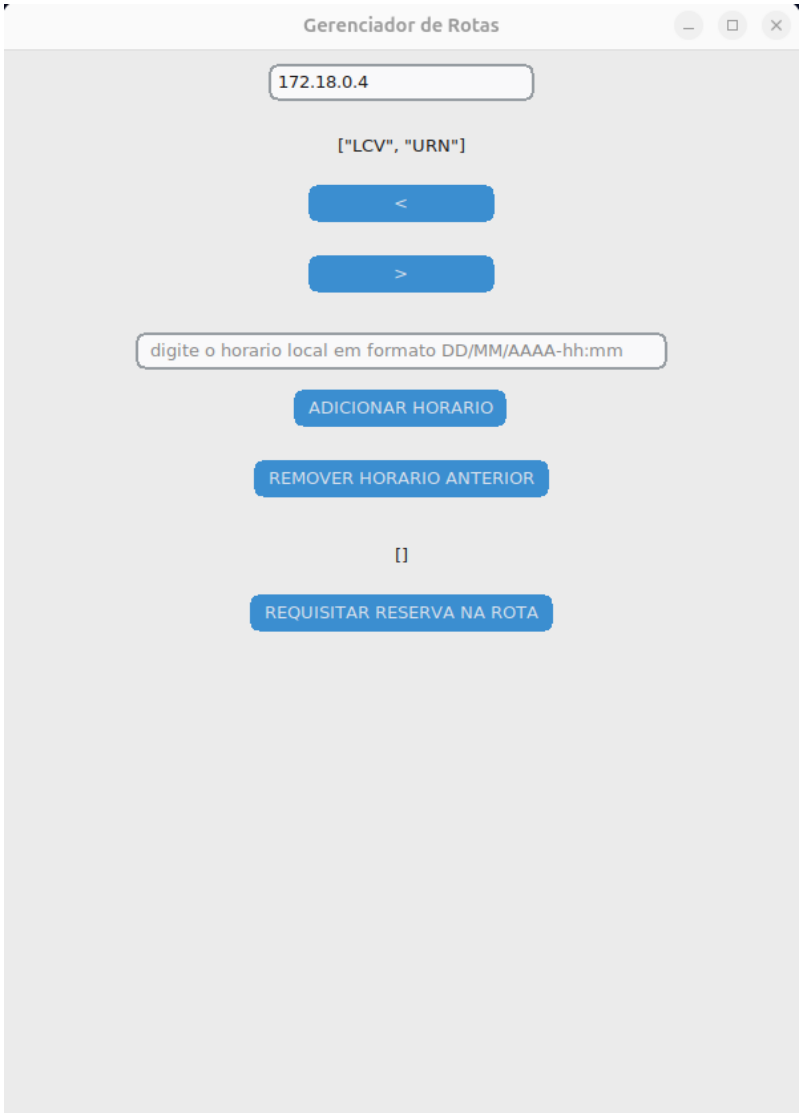

Figura 6. Menu de reserva

4.4. Histórico de compras

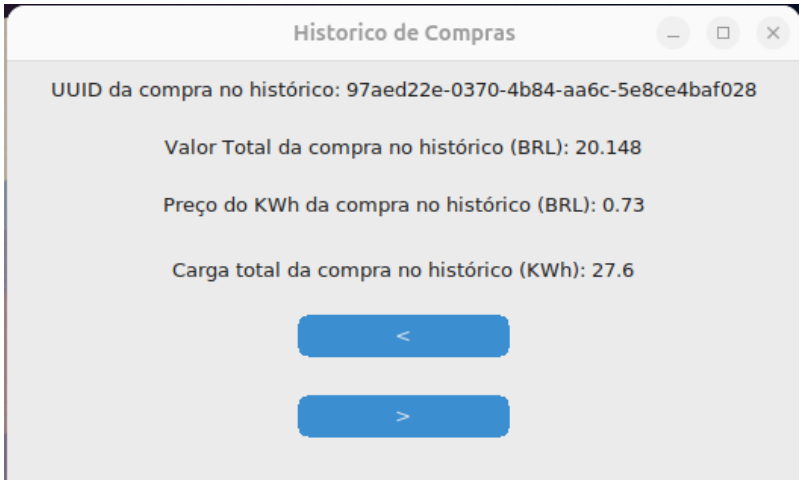

Figura 7. Histórico de compra

5. Conclusão

O sistema é capaz de atender todas as novas demandas: suporte a planejamento de rotas e reserva antecipada de pontos de recarga com apoio de requisições atômicas entre servidores.

Também é possível notar que o sistema é capaz de processar requisições utilizando protocolos de alto nível em todos os seus pontos, trazendo grande poder de abstração e modularidade (em outras palavras, é possível desenvolver aplicações que substituem e/ou fazem comunicação com algumas das partes atualmente implementadas sem possuir conhecimento acerca do código das partes ainda mantidas).

Contudo, observa-se que o atual método para adição de rotas carece de refinamento e pode ser um empecilho na manipulação por parte das empresas responsáveis.Uma forma de contornar este problema seria através da inclusão de um software de banco de dados como MongoDB ou PostgreSQL de modo a terceirizar parte do processo de gerenciamento de rotas.

Outro detalhe é a ausência de algoritmos para lidar com *livelocks* (impasses dinâmicos) em requisições simultâneas de reserva, sendo que atualmente tais situações resultam na falha de reserva para todas as entidades solicitantes. Entretanto, como tais impasses são resultados de ação assíncronas dos veículos solicitantes, é extremamente improvável que sua solução não se dê de forma natural em iterações seguintes das solicitações, causando portanto os impasses apenas uma diminuição no desempenho do sistema, e nunca seu travamento completo.

No que diz respeito ao aproveitamento acadêmico, o desenvolvimento desta aplicação proporcionou grande aprendizado nos tópicos de: protocolos *Machine-to-Machine*(M2M), API REST, comunicações cliente-servidor e servidor-servidor e também na aplicação prática de requisições atômicas.

Referências

JUNIOR, A. R. M. S. and dos Santos, M. (2019). Arquitetura rest api e desenvolvimento de uma aplicação web service. *PROJETOS E RELATÓRIOS DE ESTÁGIOS*, 1(1):1–

59.

Soni, D. and Makwana, A. (2017). A survey on mqtt: a protocol of internet of things (iot). In *International conference on telecommunication, power analysis and computing techniques (ICTPACT-2017)*, volume 20, pages 173–177.

Surwase, V. (2016). Rest api modeling languages-a developer's perspective. *Int. J. Sci. Technol. Eng*, 2(10):634–637.

Torres, A., Rocha, A., and de Souza, J. (2016). Análise de desempenho de brokers mqtt em sistema de baixo custo. In *Anais do XV Workshop em Desempenho de Sistemas Computacionais e de Comunicação*, pages 2804–2815, Porto Alegre, RS, Brasil. SBC.