# AVALIAÇÃO DA INTEGRAÇÃO DO PROTOCOLO MQTT EM UMA PLATAFORMA DE CIDADES INTELIGENTES

Mechanisms To Improve Security in IoT Platforms

Bruno Carneiro da Cunha
Orientador: Prof. Daniel Batista

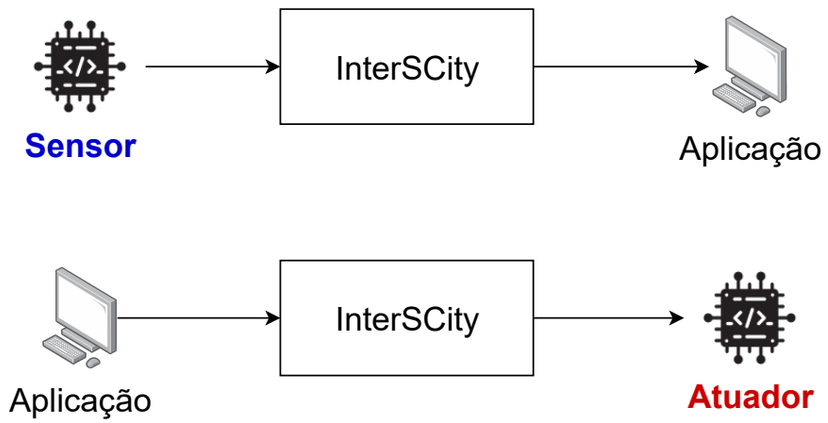
- Explicar o que será coberto na apresentação
    - Boa Tarde
    - Bruno Carneiro da Cunha (IME-USP)
    - Avaliação da Integração do protocolo MQTT em uma plataforma de cidades inteligentes
    - Prof. Daniel Batista (IME-USP)
    - Subprojeto: Mechanisms To Improve Security in IoT Platforms

# PLATAFORMA DE CIDADES INTELIGENTES

INCT
InterSCity
Future Internet for Smart Cities

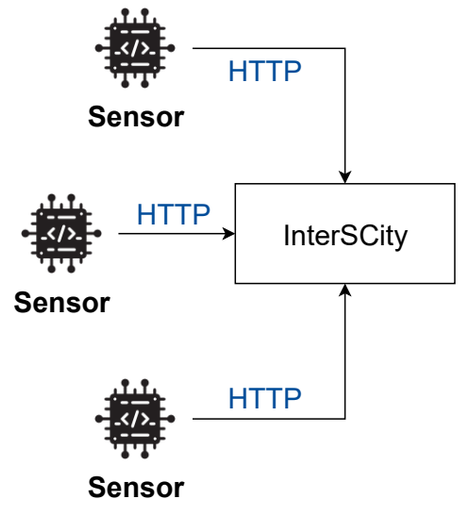
- InterSCity
    - 2016
    - Coordenador: Prof. Fábio Kon
    - Projeto de pesquisa colaborativo
- Como funciona uma plataforma de cidades inteligentes?

## PLATAFORMA DE CIDADES INTELIGENTES




- Cenário Sensor e Cenário Atuador
- Sensor
    - Dispositivo que gera dados
    - Plataforma disponibiliza esses dados para serem consumidos por uma aplicação
- Segundo Cenário (Atuador)
    - Com base nos dados coletados, uma aplicação pode alterar o estado de algum dispositivo na cidade
    - O dispositivo que recebe esse comando se chama atuador
    - Semáforo, luz de rua, etc
- InterSCity fornece uma infraestrutura pronta para o desenvolvimento de aplicações de Cidades Inteligentes

## PLATAFORMA DE CIDADES INTELIGENTES

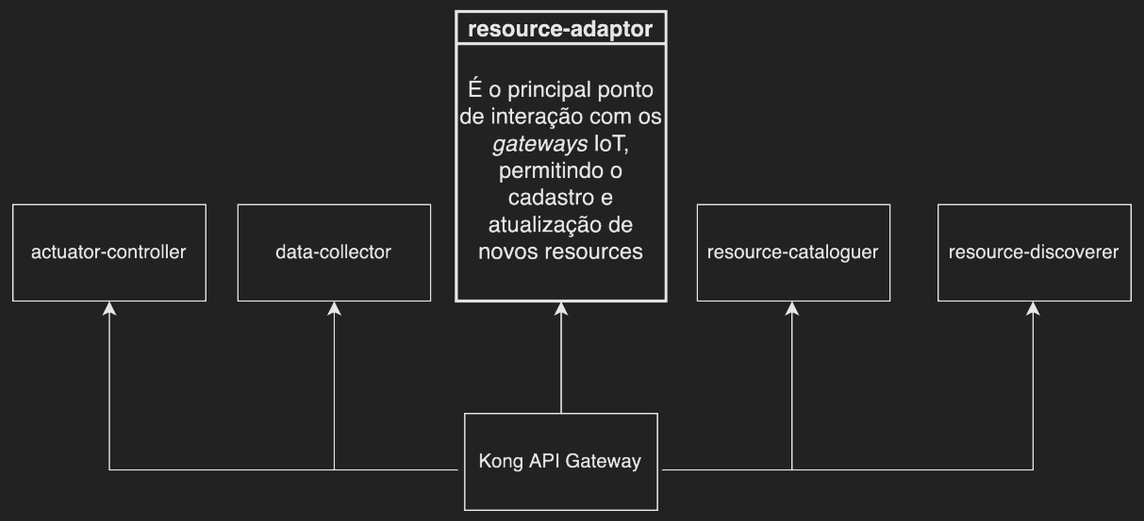




```
{
  "data": [
    {
      "temperature": 10,
      "humidity": 45,
      "pressure": 25,
      "timestamp": "2017-06-14T17:52:25.428Z"
    }
  ]
}
```


- Focando somente na interação da plataforma com os dispositivos
- Como funciona atualmente o InterSCity
  - API
  - Requests em formato JSON
- Há algum problema no uso do HTTP?
  - A princípio, não
  - É possível melhorar com o uso do MQTT?
    - Sim, pois o MQTT é considerado mais eficiente, projetado para IoT
    - **Pretendo demonstrar na apresentação que o uso do MQTT possibilitou um grande ganho de eficiência, comparado ao uso do HTTP**

## ARQUITETURA DO INTERSCITY



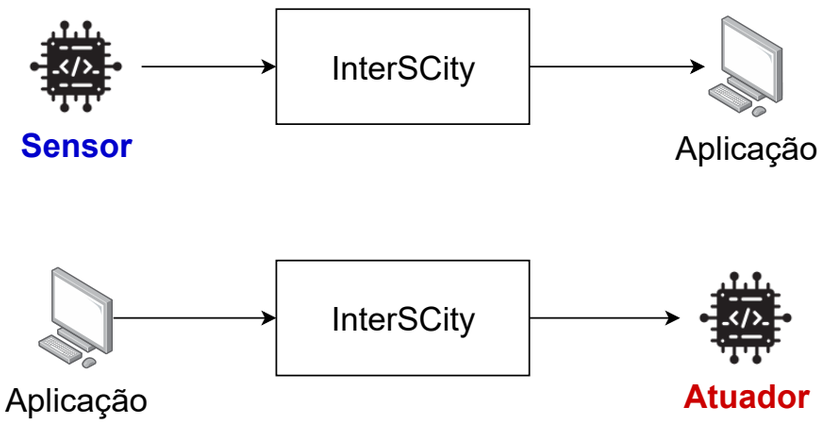

- Visão geral dos microserviços do InterSCity
- Kong: gerenciador de API, redireciona os requests pros microserviços adequados
    - Kong que de fato recebe os requests dos clientes
- *Resource Adaptor*
    - Interage com qualquer dispositivo externo
    - Criação e atualização de resources
    - Publicação de dados de sensores
    - Cadastro de *webhooks* para receber comandos pros dispositivos atuadores
    - Todas as modificações foram feitas somente no resource-adaptor

## MODIFICAÇÕES NO RESOURCE ADAPTOR

- Dois novos *workers:*
  - mqtt_subscriber.rb
  - mqtt_publisher.rb

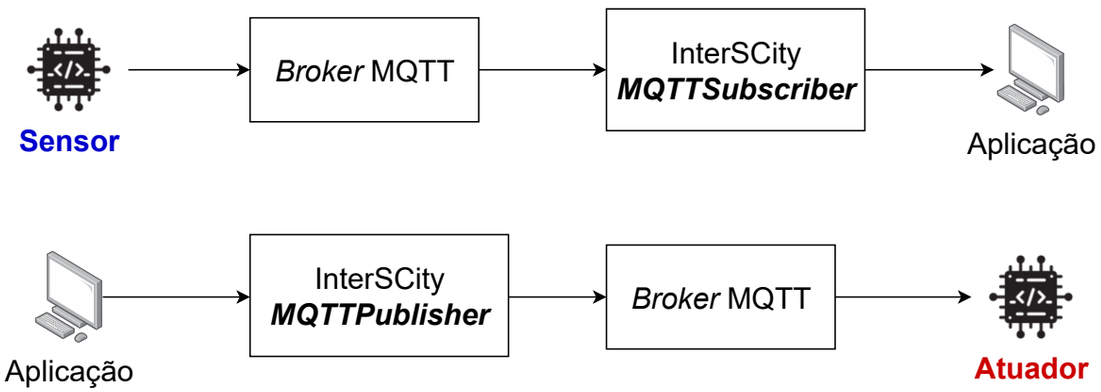
- *Worker* em Rails é uma abstração que executa código fora da *thread* principal
- MQTTSubscriber
  - Cenário Sensor
  - Recebe os dados dos dispositivos sensores
- MQTTPublisher
  - Cenário atuador
  - Publica os comandos para os dispositivos atuadores

# HTTP




- Recapitulando como funciona com HTTP

## MQTT




- Um *hop* a mais em cada envio de valor ou envio de comando

AVALIAÇÃO DA INTEGRAÇÃO

## EXPERIMENTOS

1. Vazão
2. Latência
3. Uso de CPU e memória

- Três experimentos para comparar o uso dos protocolos
    1. Vazão
        - Foram criadas um número variável de *threads* (10, 20, 40, 80) enviando valores simultaneamente para a plataforma, e medimos a taxa média de mensagens por segundo
        - Simulou um cenário com múltiplos sensores enviando valores à plataforma
        - Atendido pelo *MQTT Subscriber*
    2. Latência
        - O objetivo foi medir o impacto da introdução de um novo *hop* no envio de comandos a dispositivos atuadores
        - Foi feito um envio de um comando e mediu-se o tempo para que a plataforma repassasse ao dispositivo atuador usando ambos os protocolos
        - Atendido pelo *MQTT Publisher*
    3. Impacto no servidor — métrica uso de CPU e memória
        - Repetiu-se o primeiro experimento com um número fixo de *threads* (40)
        - Foi feito o monitoramento ao longo de um minuto do teste em cada um dos componentes da plataforma

## AMBIENTE

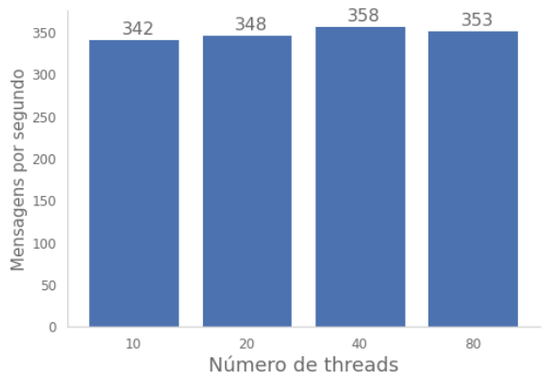
- Intel i5-8400 2.8GHz (6 núcleos), 16GB DDR4
- Mosquitto v.31
- Puma 3.12.1
- Ubuntu 18.04 LTS
- Interface de l*oopback*

- Configuração do ambiente onde foram executados os testes
- Todas as mensagens foram enviadas através da interface de *loopback*

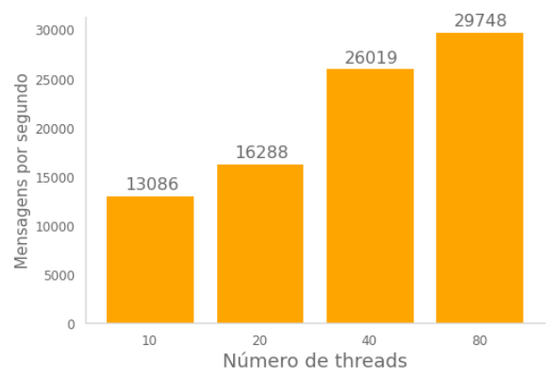
RESULTADOS

# 1. VAZÃO

## HTTP

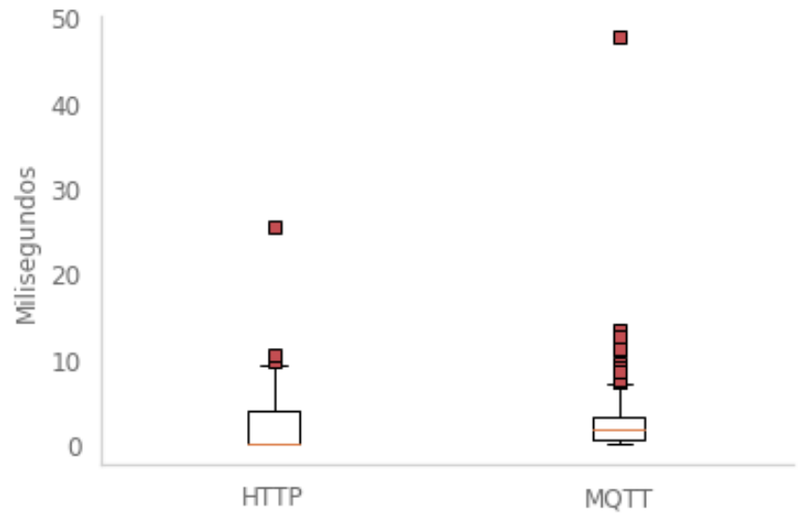



## MQTT



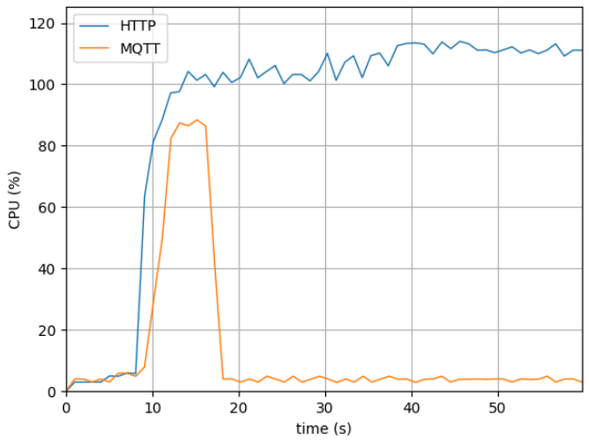

- Eixo Y — mensagens por segundo
- Eixo X — Número de *threads*: 10, 20, 40, 80
- HTTP
  - Se mantém estável — o aumento do número de *threads* não acarreta em mais vazão
- MQTT
  - O aumento do número de *threads* aumenta também o número de mensagens por segundo
- **Análise**
  - Servidor HTTP da plataforma roda na implementação de Ruby conhecida como **MRI**, que não possui *multithreading* real
  - Servidor MQTT *Mosquitto,* implementado em C, é muito mais eficiente e consegue usar mais os recursos da máquina, como veremos adiante

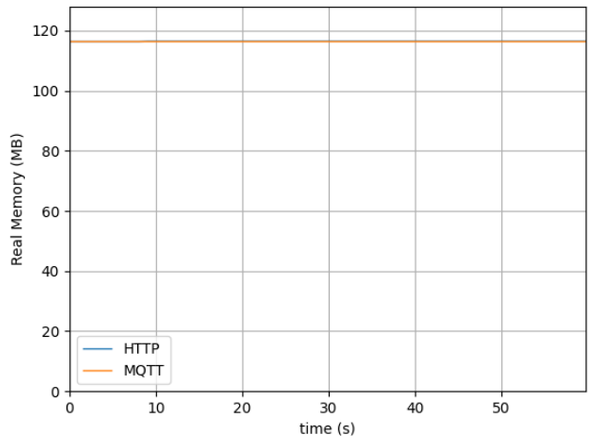
## 2. LATÊNCIA




- Eixo Y — latência em milissegundos
- Eixo X — HTTP ou MQTT
- **Análise**
  - Resultados são muito semelhantes — HTTP tem uma pequena vantagem (média 1,67ms contra 1,78ms do MQTT) que já era esperada pela adição do novo *hop*
  - A diferença não é grande o suficiente para caracterizar uma vantagem a favor do HTTP

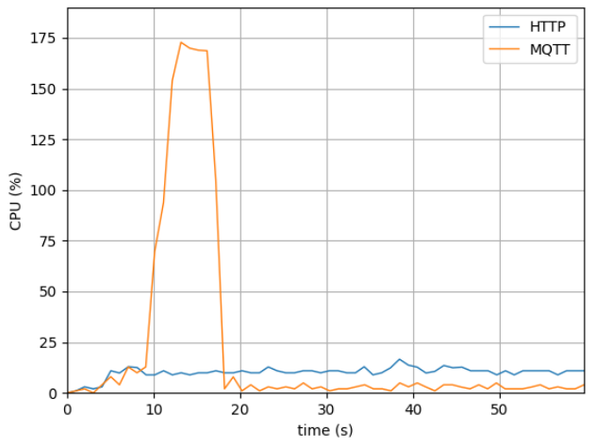
# 3. CONSUMO DE CPU E MEMÓRIA – RESOURCE ADAPTOR

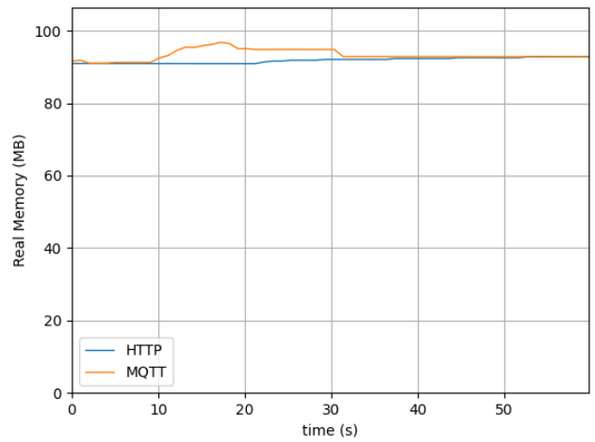



- Medimos dos 5 microserviços mais o servidor MQTT
  - Somente os resultados que apresentaram diferença são mostrados
- **Análise**
  - Cliente MQTT — MQTT Subscriber — consome menos CPU e termina o trabalho antes dos 20s
  - Enquanto HTTP continua com uso elevado até o fim e **não termina de receber** as 40.000 mensagens dos clientes antes da marca do primeiro minuto
  - Memória se mantém o mesmo

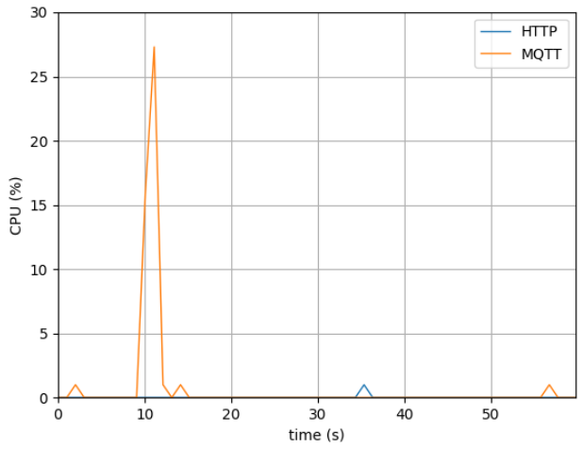
# 3. CONSUMO DE CPU E MEMÓRIA – RABBITMQ (SERVIDOR AMQP)

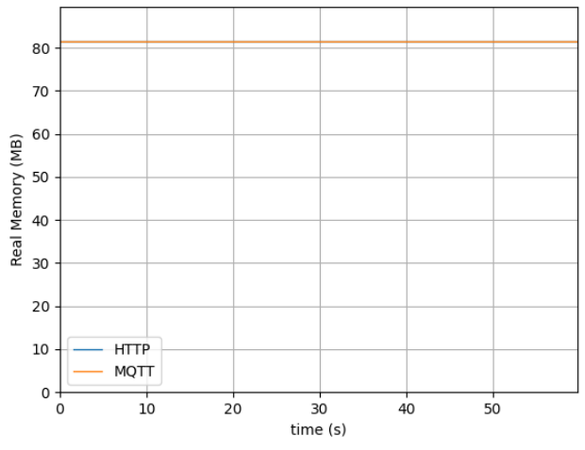



- RabbitMQ é o serviço que faz a comunicação entre os componentes — resource adaptor e data collector
  - Usa filas para que um microserviço encaminhe mensagens para outro
- **Análise**
  - Como a taxa de recebimento de mensagens do MQTT é maior, a comunicação entre os dois microserviços é mais rápida também

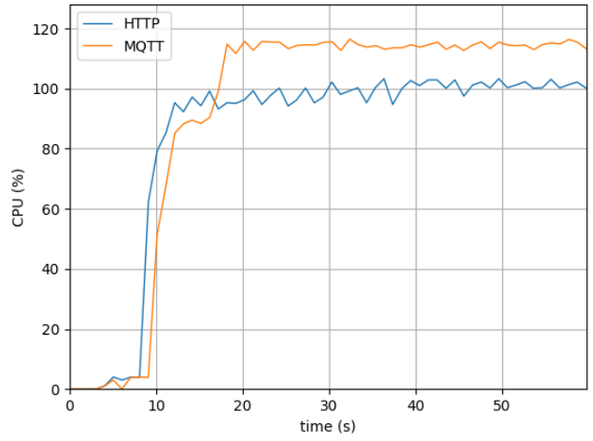
# 3. CONSUMO DE CPU E MEMÓRIA – MOSQUITTO

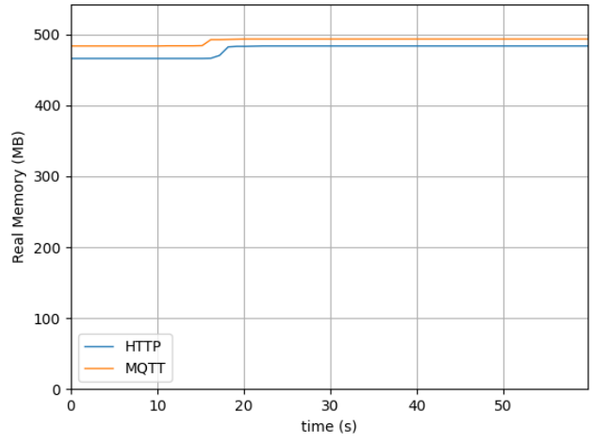



- MQTT processa as mensagens muito rapidamente com pouco uso de CPU
- **Análise**
    - Prova que a execução do Mosquitto não compromete os recursos da máquina

# 3. CONSUMO DE CPU E MEMÓRIA – DATA COLLECTOR




- *Data Collector* é o serviço que recebe os valores do *Resource Adaptor* e os cadastra no banco de dados
- **Análise**
  - Existe um gargalo grande no cadastro de valores, pois ambos os experimentos mostram que 1 minuto não foi o suficiente para terminar de processar as mensagens
  - Mostra que existe uma disputa por recursos entre os microserviços da plataforma, pois quando *Resource Adaptor* termina (MQTT), o *Data Collector* aumenta ainda mais seu uso de CPU

## CONCLUSÕES

- Adaptador MQTT apresentou uma maior vazão para os dispositivos
  - 10 *threads:* 38,26 vezes maior
  - 80 *threads:* 84,27 vezes maior
- Não houve prejuízo à latência no envio de comandos usando o MQTT

- Diferença expressiva na capacidade de vazão entre os protocolos
  - Principalmente para o lado dos dispositivos
  - O cadastro dos valores na plataforma foi mais rápido no MQTT mas ainda foi limitado pelo processamento no *Data Collector*
  - A maior capacidade de vazão usando o MQTT é benéfica tanto para os dispositivos conectados à plataforma, que podem economizar energia e processamento, quanto para os mantenedores de uma instância da InterSCity, que podem alocar uma máquina de menor custo para atender um mesmo número de dispositivos.
- Aplicações sensíveis à latência podem usar o MQTT sem nenhum problema

## PRÓXIMOS PASSOS

- *Deploy* e *Pull Request*
- Rede de experimentação na USP

- Preparar o código para encaixar no *deploy* existente do InterSCity
- Fazer o *pull request* se for do interesse dos mantenedores da plataforma
- Rede de experimentação:
    - Segunda parte do projeto
    - Instalar as placas no campus ou outra localidade

# OBRIGADO🌻👨🏽‍💻

- Agradecer ao Prof. Daniel
- Agradecer a todos pela atenção
- Perguntas, sugestões, críticas

- Agradecer ao Prof. Daniel
- Agradecer a todos pela atenção
- Perguntas, sugestões, críticas

## FUNCIONAMENTO DO MQTT SUBSCRIBER

| HTTP | MQTT |
| --- | --- |
| /adaptor/resources/{uuid}/data | resources/{uuid} |
| /adaptor/resources/{uuid}/data/{capability} | resources/{uuid}/{capability} |

- **SLIDE EXTRA: SOMENTE PARA RESPONDER PERGUNTAS SE NECESSÁRIO**
- **Subscriber**
    - “Espelha" esses dois *endpoints* da API do *resource-adaptor*
    - Formato da mensagem é idêntico ao formato usado no HTTP
    - MQTTSubscriber é um cliente MQTT dentro da plataforma que assina o tópico **resources/#**

## FUNCIONAMENTO DO MQTT PUBLISHER

| HTTP | MQTT |
| --- | --- |
| Enviadas diretamente ao *webhook* cadastrado | commands/{uuid} |

- **SLIDE EXTRA: SOMENTE PARA RESPONDER PERGUNTAS SE NECESSÁRIO**
- **Publisher**
  - Recebimento dos comandos a partir da assinatura do tópico **commands/{uuid}**
  - MQTTPublisher é um cliente MQTT dentro da plataforma que publica no tópico **commands/{uuid}**
  - Formato da mensagem é idêntico ao formato usado no HTTP
- **Trabalho de desenvolvimento realizado e concluído ao longo do 2o semestre do ano passado, passamos à avaliação**

## RESOURCES E CAPABILITIES NO INTERSCITY

- Cada resource é um dispositivo que envia dados ou recebe comandos
  - Ex.: Estação metereológica, sensor de presença residencial
- Capabilities são o tipo de informação que cada um desses resources pode ter
  - Ex.: Temperatura, humidade, pressão sonora

- **SLIDE EXTRA: SOMENTE PARA RESPONDER PERGUNTAS SE NECESSÁRIO**
- Essas são as definições de resource e capability