ACH2017

Gustavo Tsuyoshi Ariga

27 de junho de 2023

Sumário

Ocorreu-me que mostrar a sequência de operação do sistema é interessante.

O servidor é atingido através de seu endereço IP. O dispositivo tem, codificado, o IP do servidor. O Browser também atinge o servidor através do seu IP.

A especificação do sensor faz parte do código do servidor.

O código do servidor usa WoTPy, que é aderente à recomendação W3C WoT. A recomendação define maneiras de descobrir e configurar *servients*. No caso, o servidor deve ser capaz de configurar-se para receber os dados do dispositivo, desde que o dispositivo envie a informação necessária (no momento a informação está codificada no servidor).

A descoberta do dispositivo pode ser mais complicada porque o dispositivo não usa WoTPy, consequentemente, o programador precisará implementar a funcionalidade sem o auxílio da biblioteca. Uma forma de fazer descoberta automática é através de mDNS (multicast DNS). Esse recurso precisa ser suportado também pelo *Access Point Wi-Fi* e permite que quando um nó da rede envia uma requisição contendo um domínio nomeado, o nome do domínio seja encaminhado para outros nós da rede para resolver o endereço IP que corresponde ao nome. A localização da informação de configuração do *servient* é colocada em uma sub-URL padronizada (por exemplo, wot). Por exemplo, quando um *servient* é iniciado, ele adquire um IP do access point e associa seu nome (internamente codificado) ao IP. Caso uma requisição de solução de nome de domínio lhe chegue, ele responde com seu IP. Caso haja gateways internet no caminho, os gateways

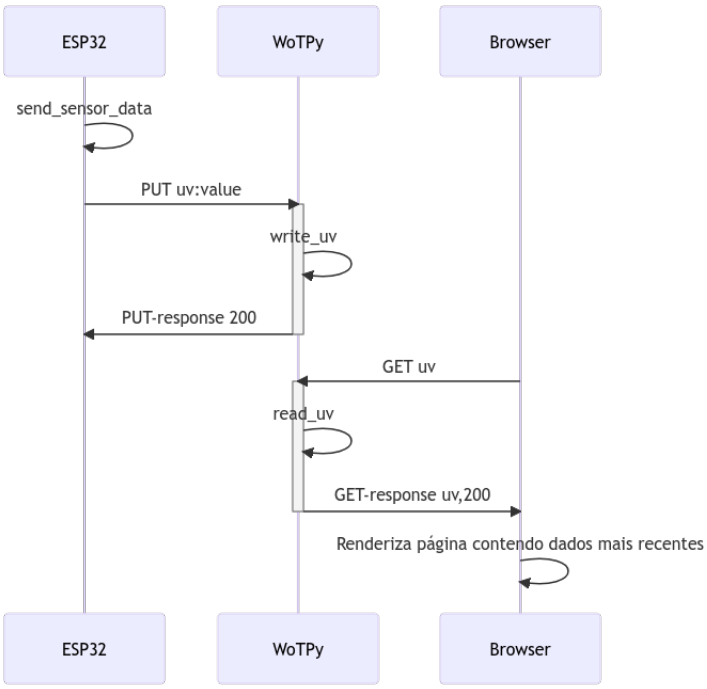

Figura 1: O dispositivo (ESP32) executa um loop que, a cada 5 segundos executa a função send_sensor_data. Esta função envia uma requisição PUT para o servidor (WoTPy). Em resposta a essa requisição o servidor executa o handler (função) write_uv que contém o envio da resposta. Um usuário pode acessar a informação no servidor através do Browser. Entrar a URL na barra de endereços do Browser o faz enviar ao servidor uma requisição GET. Em resposta à requisição o servidor executa o handler (função) read_uv, que contém o envio da resposta - no caso, o valor de uv mais recente. O Browser recebe essa informação e renderiza a página contendo a informação.

que estiverem no caminho da resposta bem sucedida vão manter essa associação, o que acelera o roteamento da informação. Isto só é feito para servidores web, não é feito para clientes. Logo, para o dispositivo poder ser descoberto através desse mecanismo ele precisa ser um servidor web, o que pode não ser possível, dependendo da capacidade computacional e disponibilidade de energia. O ESP32 pode ser configurado como servidor, logo, é compatível com este mecanismo.

Outros mecanismos podem usar diretórios de serviços (servidores possuidores de nome de domínio que mantém tabelas de IPs e características).

O uso (experimentação) dos mecanismos de configuração e descoberta recomendados pela W3C e, em princípio, implementados no WoTPy torna-se possível agora, quando possui-se um *servient* baseado em WoTPy. Esta atividade pode fazer parte das próximas atividades deste TCC.

Capítulo 1

Plano de Atividades

Aula 1: Mensagem 1

23 Março 2023

1.1 Introdução

O objetivo deste trabalho é aproximar a implementação de um ambiente com dispositivos Web of Things e conectar as ICs dos alunos do grupo. Para isso, foi avaliado um gateway para web of things chamado WoTPy. O artigo que o apresenta é "WOTPY: A framework for Web of Things applications", disponível para download na internet.

O WoTPy foi escolhido porque foi desenvolvido dentro de um grupo de pesquisa em Web Of Things, é codificado em Python e possui código aberto no Github. Além disso, tem potencial para ser usado em Semantic Web of Things (SWoT). No entanto, não está maduro o suficiente para ser simplesmente baixado e utilizado. Por isso, o objetivo do TCC é levar o WoTPy a um ponto de maturidade adequado.

Para alcançar esse objetivo, será feito um fork do WoTPy e ajustadas as dependências. Em seguida, será encapsulado em um container (Docker) e testado com um dispositivo baseado em ESP32. Se houver tempo suficiente, será implementada no WoTPy a versão mais recente da especificação da W3C para WoT.

Este trabalho é importante para o avanço da tecnologia Web of Things e para a conexão das ICs dos alunos do grupo. Com a conclusão do projeto, espera-se contribuir para a evolução do WoTPy e torná-lo uma ferramenta mais acessível para a comunidade.

https://www.overleaf.com/read/bjszxjfvdpnk

Aula 2: Reunião 1 27 Março 2023

1.2 Apresentação

O objetivo deste trabalho é implementar uma recomendação da W3C em um ambiente acadêmico. Diferente dos softwares comerciais, o foco está na interface do usuário. Embora a aderência aos padrões seja importante, o objetivo principal é construir um ambiente que atenda às necessidades dos usuários.

Para atingir esse objetivo, será utilizado o WoTPy, um software desenvolvido na área acadêmica capaz de implementar um gateway web of things. O WoTPy está dentro da recomendação da W3C, o que garante a qualidade e segurança da implementação.

É importante destacar que a implementação da recomendação da W3C trará diversos benefícios para o ambiente acadêmico. Isso inclui a melhoria da qualidade e segurança dos dados, a facilidade de integração de novos dispositivos e a padronização das comunicações entre eles. Além disso, a implementação da recomendação da W3C permitirá que o ambiente acadêmico esteja em conformidade com os padrões internacionais, o que é importante para a reputação da instituição.

Exposed Thing and Consumed Thing Abstractions

1.3 Problema

O objetivo deste projeto é explorar o potencial do WoTPy, uma ferramenta para implementação de gateways em Web of Things, e justificar por que ela é interessante para a nossa aplicação. Durante o primeiro semestre, vamos trabalhar na adaptação de dispositivos para conversar com o WoTPy e testar sua funcionalidade internamente. Também vamos procurar soluções para os problemas encontrados, contribuindo assim para a comunidade que criou a ferramenta.

Para facilitar o uso do WoTPy, vamos implementá-lo em um contêiner Docker. Além disso, vamos explorar a possibilidade de gerar automaticamente o código necessário para adaptar dispositivos ao WoTPy, aumentando assim sua usabilidade.

Durante o segundo semestre, vamos aprofundar nosso conhecimento do WoTPy e desenvolver ferramentas para geração de código e criação de grafos de conhecimento. Essas ferramentas podem ajudar a simplificar ainda mais a implementação de gateways em Web of Things e aumentar sua aderência a padrões.

- quarta-feira 17:30
- 07/04 ultima versão pronta
- 31/03 versão intermediária

Aula 3: Reunião 2

29 Março 2023

1.4 Objetivo

https://www.w3.org/WoT/IG/wiki/Terminology

O objetivo específico deste projeto é colocar o WoTPy para funcionar, suas extensões e casos específicos, durante o primeiro semestre, e posteriormente, no segundo semestre, integrá-lo com outras ferramentas, como a geração de código e a utilização de GPT (Generative Pre-trained Transformer). Para isso, será necessário compreender melhor o contexto mais amplo do WoT e suas definições.

Web of Things (WoT) é uma área que busca a interoperabilidade entre diferentes dispositivos, permitindo que eles se comuniquem utilizando diferentes protocolos e atendam a diferentes requisitos. Desde 2010, pesquisadores como Dominique Dagardi e Ciocia têm trabalhado em soluções para o desenvolvimento de dispositivos semânticos, ou seja, que possam interpretar dados de diferentes máquinas e realizar ações de forma autônoma.

Uma das iniciativas mais importantes na área do WoT é a recomendação da W3C (World Wide Web Consortium), que busca estabelecer uma terminologia comum para a área e identificar padrões para a interoperabilidade. Além disso, a cronologia do IoT mostra que a interoperabilidade é um problema não resolvido desde 2007, quando diferentes dispositivos começaram a ser utilizados para coletar e transmitir dados.

Para resolver este problema, uma das soluções propostas é a utilização de agentes de software, conhecidos como gateways, que possibilitam a comunicação entre diferentes dispositivos utilizando diferentes protocolos. O WoTPy é uma ferramenta voltada para a área acadêmica que permite a implementação de gateways WoT e sua extensão para diferentes casos específicos.

Durante o primeiro semestre, o objetivo será colocar o WoTPy para funcionar, utilizando o Docker para sua instalação e realizando testes internos para verificar seu desempenho. Também serão construídos dispositivos adaptados para conversar com o WoTPy, utilizando a geração automática de código.

No segundo semestre, serão exploradas outras ferramentas para a geração de código, como a utilização de grafos de conhecimento, e será realizada a integração com outras ferramentas, como o GPT e o OpenAPI, buscando aprimorar a capacidade do WoTPy de integrar diferentes dispositivos e protocolos.

Aula 4: Reunião 3

03 Maio 2023

1.5 Próximo Passo

Definir a topologia possível: Identificar e planejar a estrutura da rede em que o ESP32 e os dispositivos consumidores estarão conectados.

Descrever o serviço API: Utilizar a descrição semântica para definir o serviço que será implementado no ESP32.

Traduzir para OpenAPI: Converter a descrição semântica em um documento OpenAPI (openapi.json) que descreve a API de forma padronizada e legível por máquinas.

Gerar o código usando Swagger Codegen: Utilizar a ferramenta Swagger Codegen para gerar automaticamente o stub (esqueleto) do código necessário para implementar o servidor e o cliente da API no ESP32 e nos dispositivos consumidores, respectivamente.

Integrar o stub gerado com o servidor web do ESP32: Incluir o código gerado pelo Swagger Codegen no servidor web do ESP32, que estará esperando por requisições.

Implementar as funções do WoTpy: No código gerado, preencher os stubs com as funções específicas do WoTpy, uma biblioteca para trabalhar com a Web das Coisas (WoT).

Conectar dispositivos consumidores: Os dispositivos consumidores se conectam ao servidor web do ESP32, fazendo requisições e interagindo com o serviço disponibilizado pela API.

O OpenAPI gera 'stubs' de código, mas não o código inteiro. A intenção seria rechear estes 'stubs' com chamadas à métodos do WotPy, trazendo a funcionalidade completa do programa. Parece ser interessante passar o código do WotPy na ferramenta do Prof. Dr. Andre e carregá-lo em meu gerador de código, de tal forma que, com "chamadas semânticas", seja possível gerar as chamadas aos métodos do WotPy na ordem correta com os parâmetros certos...

- Towards a secure API client generator for IoT devices
- Deployment of APIs on Android Mobile Devices and Microcontrollers
- `https://openapi-generator.tech/docs/generators/cpp-tiny/`

Servidor gateway IoT: Criar um servidor que atue como gateway IoT, conectando dispositivos, como sensores, à Internet.

Armazenamento de dados: O gateway deve armazenar os dados dos sensores conectados, com um formato não claramente definido.

Disponibilizar dados através de um endpoint SPARQL: O servidor deve disponibilizar os dados armazenados através de um endpoint SPARQL, permitindo consultas e acesso aos dados.

Utilizar a especificação Web of Things (WoT): O gateway deve manter os dados de acordo com a especificação WoT e não utilizar o OpenAPI.

Gerar dados no formato Thing Description (TD) com Thing Scripting: Os dados devem ser gerados e mantidos no formato TD, seguindo o padrão W3C, utilizando a abordagem de Thing Scripting.

José utilizaria um tradutor Web of Things para OpenAPI: Para criar consumidores desses dados, José poderia utilizar um tradutor que converte as descrições de Thing Description para OpenAPI.

Sensores ESP conectados ao gateway: Os sensores ESP devem ser conectados ao gateway, enviando dados para armazenamento e respondendo às solicitações.

Formato de dados semânticos com Thing Description: Os dados devem ser armazenados e disponibilizados no formato semântico definido pela especificação Thing Description.

Implementação compatível com Thing Description: Certificar-se de que o servidor seja compatível com a especificação Thing Description, mesmo que ainda não tenha sido totalmente implementado.

- https://www.sciencedirect.com/science/article/pii/S2542660522001561

Capítulo 2

WoTPy

Aula 1: Problemas de Instalação

12 Abril de 2023

2.1 Semana 01

Trabalhei no projeto WotPy, um projeto disponível no GitHub, cujo link pode ser encontrado em: https://github.com/agmangas/wot-py. Este relatório apresenta os passos realizados e os problemas encontrados durante a execução e construção do projeto.

2.1.1 Fork e Clone do Projeto

Primeiramente, realizei um fork do projeto WotPy e clonei o repositório para o meu computador.

2.1.2 Construção do Projeto com Docker

Tentei construir o projeto usando o Docker com o seguinte comando:

```
docker build .
```

Durante a construção do projeto com o Docker, enfrentei alguns erros relacionados à versão do pacote numpy e à versão do pip. Os erros foram os seguintes:

```
ERROR: Could not find a version that satisfies the
    requirement numpy==1.22.0 (from versions: 1.3.0,
    1.4.1, 1.5.0, 1.5.1, 1.6.0, 1.6.1, 1.6.2, 1.7.0,
    1.7.1, 1.7.2, 1.8.0, 1.8.1, 1.8.2, 1.9.0, 1.9.1,
    1.9.2, 1.9.3, 1.10.0.post2, 1.10.1, 1.10.2, 1.10.4,
    1.11.0, 1.11.1, 1.11.2, 1.11.3, 1.12.0, 1.12.1,
    1.13.0, 1.13.1, 1.13.3, 1.14.0, 1.14.1, 1.14.2,
    1.14.3, 1.14.4, 1.14.5, 1.14.6, 1.15.0, 1.15.1,
```

```
    1.15.2, 1.15.3, 1.15.4, 1.16.0, 1.16.1, 1.16.2,
    1.16.3, 1.16.4, 1.16.5, 1.16.6, 1.17.0, 1.17.1,
    1.17.2, 1.17.3, 1.17.4, 1.17.5, 1.18.0, 1.18.1,
    1.18.2, 1.18.3, 1.18.4, 1.18.5, 1.19.0, 1.19.1,
    1.19.2, 1.19.3, 1.19.4, 1.19.5, 1.20.0, 1.20.1,
    1.20.2, 1.20.3, 1.21.0, 1.21.1, 1.21.2, 1.21.3,
    1.21.4, 1.21.5, 1.21.6)
ERROR: No matching distribution found for numpy==1.22.0
WARNING: You are using pip version 22.0.4; however,
    version 23.0.1 is available.
You should consider upgrading via the '/usr/local/bin/
    python -m pip install --upgrade pip' command.
The command '/bin/sh -c pip install -r requirements.txt'
    returned a non-zero code: 1
```

2.1.3 Execução dos Testes

Iniciei a execução dos testes do WoTPy com o seguinte comando:

```
./pytest-docker-all.sh
```

Após a execução da construção, percebi o seguinte erro:

```
Traceback (most recent call last):
        File "<string>", line 2, in <module>
        File "<pip-setuptools-caller>", line 34, in <
            module>
        File "/app/setup.py", line 58, in <module>
          setup(
        File "/usr/local/lib/python3.10/site-packages/
            setuptools/__init__.py", line 87, in setup
          return distutils.core.setup(**attrs)
        File "/usr/local/lib/python3.10/site-packages/
            setuptools/_distutils/core.py", line 185, in
            setup
          return run_commands(dist)
        File "/usr/local/lib/python3.10/site-packages/
            setuptools/_distutils/core.py", line 201, in
            run_commands
          dist.run_commands()
        File "/usr/local/lib/python3.10/site-packages/
            setuptools/_distutils/dist.py", line 968, in
            run_commands
          self.run_command(cmd)
        File "/usr/local/lib/python3.10/site-packages/
            setuptools/dist.py", line 1217, in run_command
          super().run_command(command)
```

```
    File "/usr/local/lib/python3.10/site-packages/
        setuptools/_distutils/dist.py", line 987, in
        run_command
      cmd_obj.run()
    File "/usr/local/lib/python3.10/site-packages/
        wheel/bdist_wheel.py", line 395, in run
      self.egg2dist(self.egginfo_dir, distinfo_dir)
    File "/usr/local/lib/python3.10/site-packages/
        wheel/bdist_wheel.py", line 534, in egg2dist
      pkg_info = pkginfo_to_metadata(egginfo_path,
          pkginfo_path)
    File "/usr/local/lib/python3.10/site-packages/
        wheel/metadata.py", line 160, in
        pkginfo_to_metadata
      for key, value in generate_requirements({extra:
          reqs}):
    File "/usr/local/lib/python3.10/site-packages/
        wheel/metadata.py", line 138, in
        generate_requirements
      for new_req in convert_requirements(depends):
    File "/usr/local/lib/python3.10/site-packages/
        wheel/metadata.py", line 103, in
        convert_requirements
      parsed_requirement = Requirement(req)
    File "/usr/local/lib/python3.10/site-packages/
        wheel/vendored/packaging/requirements.py",
        line 37, in __init__
      raise InvalidRequirement(str(e)) from e
wheel.vendored.packaging.requirements.
    InvalidRequirement: Expected end or semicolon (
    after version specifier)
    coverage>=5.0<6.0
             ~~~~~^

[end of output]
```

2.1.4 Atualização

https://github.com/T16K/wot-py/commit/b8e9d090e738b343a825cd404f63dde950954a70

Aula 2: Revisão da Semana Passada

19 Abril 2023

2.2 Semana 02

Ao fazer o teste da implementação da semana passada, executando o arquivo "pytest-docker-all.sh", todos os testes são aprovados.

2.2.1 Warning

Durante a execução do "docker build .", ocorreram alguns avisos que não foram mencionados anteriormente.

```
WARNING: Running pip as the 'root' user can result in
    broken permissions and conflicting behaviour with the
    system package manager. It is recommended to use a
    virtual environment instead: https://pip.pypa.io/
    warnings/venv

WARNING: You are using pip version 22.0.4; however,
    version 23.1 is available.
You should consider upgrading via the '/usr/local/bin/
    python -m pip install --upgrade pip' command.
```

2.2.2 Atualização das Dependênciais

Optei por utilizar a versão original do Python (3.7) e, especificamente no arquivo "examples/benchmark/requirements.txt", reverti à versão modificada pelo [dependabot[bot]].

https://github.com/T16K/wot-py/commit/61bec7348c6342c05fd8d2c3ccb21dad60aed58b

2.2.3 Resultados

Dessa forma, o WoTPy consegue construir corretamente o Docker e passar nos testes propostos no "pytest-docker-all.sh".

Aula 3: Documentação

28 Abril 2023

2.3 Semana 03

https://github.com/Endum/WoTLamp/blob/master/lamp.py

A descoberta automática de dispositivos é um recurso que permite que os dispositivos sejam identificados e conectados a uma rede sem a necessidade de configuração manual. Embora não seja mencionado nos textos, é possível que haja soluções específicas para cada protocolo que possam auxiliar na descoberta automática de dispositivos em um ambiente de Internet das Coisas (IoT).

Na documentação, não há menção explícita à descoberta automática de dispositivos para os protocolos de binding HTTP, MQTT e CoAP. Eles focam principalmente no mapeamento entre ações de alto nível que podem ser executadas em um "Thing"e as mensagens trocadas ao usar esses protocolos de binding específicos.

Além disso, a documentação não especifica um tipo de servidor em particular para os protocolos de binding HTTP, MQTT e CoAP. No entanto, cada protocolo possui características diferentes e, portanto, pode exigir tipos de servidores distintos:

HTTP: O protocolo HTTP é geralmente utilizado em servidores web. Um servidor HTTP pode ser baseado em soluções como Apache, Nginx, Microsoft IIS, entre outros, para processar e responder às solicitações HTTP de clientes.

MQTT: O MQTT requer a presença de um broker MQTT externo, que é um servidor especializado em lidar com a troca de mensagens entre dispositivos em um padrão de publicação/assinatura. Exemplos de brokers MQTT incluem Mosquitto, HiveMQ, EMQ X e RabbitMQ com o plugin MQTT.

CoAP: O CoAP é um protocolo leve projetado para uso em dispositivos de recursos limitados, como sensores e atuadores em redes de IoT. Um servidor CoAP pode ser construído com bibliotecas e ferramentas específicas, como Californium (Java), libcoap (C), CoAP.NET (C#) e outras.

É importante ressaltar que o tipo de servidor utilizado depende do protocolo e das necessidades do sistema, podendo ser adaptado de acordo com as restrições e requisitos específicos do ambiente.

2.3.1 WebSockets

O protocolo WebSockets é utilizado para executar ações de alto nível em um Thing, trocando mensagens no formato JSON-RPC 2.0 com o servidor. Os elementos de formulário associados ao WebSockets têm o formato:

```
{
"href": "ws://host.fundacionctic.org:9393/temperaturething",
"mediaType": "application/json"
}
```

As interações com o servidor WebSocket ocorrem através da troca de mensagens JSON-RPC, como mensagens de solicitação enviadas pelo cliente, mensagens de resposta enviadas pelo servidor e mensagens de erro retornadas em caso de problemas. Mensagens de Item Emitido são enviadas para inscrições ativas quando novos eventos são emitidos.

O mapeamento do modelo de interação inclui: Ler Propriedade, Escrever Propriedade, Invocar Ação, Observar Mudanças de Propriedade, Observar Evento, Observar Mudanças no TD (Thing Description) e Descartar Inscrição. Cada interação possui um formato de mensagem específico para solicitação e resposta.

2.3.2 HTTP

A seção descreve o mapeamento entre ações de alto nível que podem ser executadas em um Thing e as mensagens trocadas com o servidor ao usar o HTTP Protocol Binding. As mensagens são serializadas em formato JSON e os elementos de formulário variam de acordo com o tipo de interação.

O mapeamento do modelo de interação inclui: Ler Propriedade, Escrever Propriedade, Invocar Ação, Observar Mudanças de Propriedade e Observar Evento. Cada interação possui um formato específico de solicitação e resposta, e o protocolo HTTP adota o padrão de long-polling para lidar com mensagens do lado do servidor.

Ler e Escrever Propriedades envolvem requisições GET e PUT, respectivamente, e ambos retornam uma resposta HTTP 200 com o valor da propriedade. Invocar Ação envolve uma requisição POST, que inicia a invocação e retorna um UUID único. O status da invocação pode ser obtido através de uma requisição GET.

Observar Mudanças de Propriedade e Observar Evento são realizados por meio de requisições GET, e as respostas seguem o formato das ações Ler Propriedade e Observar Evento, respectivamente. As inscrições são gerenciadas automaticamente pelo HTTP Binding, sendo inicializadas em cada solicitação e canceladas após a emissão de um valor.

2.3.3 MQTT

Esta seção descreve o mapeamento entre as ações de alto nível que podem ser executadas em um Thing e as mensagens trocadas com o broker MQTT ao usar o MQTT Protocol Binding.

Diferente de outros bindings, o binding MQTT não é autossuficiente e requer a presença de um broker MQTT externo. As mensagens são serializadas em formato JSON e os elementos de formulário variam de acordo com o tipo de interação.

Os tópicos do MQTT são divididos em seis tipos diferentes usados pelos clientes e servidores para troca de mensagens. O mapeamento do modelo de interação inclui: Ler Propriedade, Observar Mudanças de Propriedade, Escrever Propriedade, Invocar Ação e Observar Evento.

Ler Propriedade envolve a publicação de uma mensagem no tópico de solicitação de propriedade, que força o servidor a publicar o valor atual da propriedade no tópico de atualização de propriedade.

Observar Mudanças de Propriedade ocorre automaticamente, com todas as mudanças de propriedade publicadas no tópico de atualização de propriedade, sem intervenção adicional do cliente.

Escrever Propriedade requer que o cliente publique uma mensagem no tópico de solicitação de propriedade, e o servidor reconhece a escrita publicando uma

mensagem no tópico de confirmação de escrita de propriedade.

Invocar Ação inicia-se com a publicação de uma mensagem no tópico de invocação de ação, e o resultado da invocação é publicado no tópico de resultado de ação.

Observar Evento acontece automaticamente, com todas as emissões de evento publicadas no tópico de emissão de evento, sem intervenção adicional do cliente.

Não há necessidade de gerenciar manualmente as inscrições, pois o servidor MQTT mantém uma inscrição interna para todas as propriedades e eventos durante sua vida útil.

2.3.4 CoAP

Esta seção descreve o mapeamento entre as ações de alto nível que podem ser executadas em um Thing e as mensagens trocadas com o servidor ao usar o CoAP Protocol Binding.

Todas as mensagens são serializadas em formato JSON. Os elementos de formulário produzidos pelo binding CoAP variam dependendo do tipo de interação. Os servidores no binding CoAP expõem três recursos distintos, um para cada tipo de interação (propriedade, ação e evento).

O mapeamento do modelo de interação inclui: Ler Propriedade, Escrever Propriedade, Observar Mudanças de Propriedade, Invocar Ação e Observar Evento. O binding CoAP aproveita o CoAP Observe para implementar mensagens do lado do servidor ao invocar ações ou assinar propriedades/eventos.

Ler Propriedade envolve o envio de uma solicitação GET para o URL CoAP correspondente. A resposta incluirá o valor da propriedade em questão.

Escrever Propriedade requer o envio de uma solicitação PUT para o URL CoAP apropriado, incluindo o novo valor da propriedade no corpo da mensagem.

Observar Mudanças de Propriedade é semelhante à ação de Ler Propriedade, exceto que o cliente deve se registrar como observador para começar a receber mensagens do lado do servidor.

Invocar Ação inicia-se com uma solicitação POST para o URL CoAP relevante, incluindo o argumento da ação no corpo da mensagem. O cliente pode verificar o status da invocação observando o recurso e passando o ID da invocação no payload.

Observar Evento envolve a criação de inscrições para o evento observando o recurso. Cada resposta do servidor para uma inscrição ativa conterá a emissão de evento mais recente.

Aula 4: Exemplos de Uso

03 Maio 2023

2.4 Semana 04

Para estudar os exemplos do wotpy, utilizei o comando docker para executar a imagem. O diretório "asdf"foi escolhido arbitrariamente, enquanto "64bf8cf085ff"representa a identificação da imagem. É importante lembrar de garantir que o Docker esteja em execução, utilizando o comando "systemctl start docker".

```
docker container run -ti --rm -v $PWD:/asdf 64bf8cf085ff sh
```

2.4.1 temperature

Arquivo **server.py** (Simulação de Temperatura)

Este é um servidor Web das Coisas (WoT) que expõe uma entidade simulada de temperatura com duas propriedades e um evento.

O servidor WoT de temperatura expõe uma entidade com as seguintes propriedades e evento:

Propriedade temperature: um número de ponto flutuante que representa a temperatura simulada, com somente leitura e observável. **Propriedade high-temperature-threshold**: um número de ponto flutuante que representa o limite superior da temperatura, observável e pode ser alterado. **Evento high-temperature**: acionado quando a temperatura simulada ultrapassa o limite superior definido. Para acessar a entidade WoT e suas propriedades, você pode fazer uma solicitação HTTP GET para a seguinte URL: `http://localhost:9494/properties/temperature`. Isso retornará a temperatura simulada atual em formato JSON.

Para definir o limite superior da temperatura, você pode fazer uma solicitação HTTP PUT para a seguinte `URL:http://localhost:9494/properties/high-temperature-threshold`. Inclua o valor do limite superior no corpo da solicitação em formato JSON.

O servidor WoT de temperatura também permite a comunicação bidirecional usando WebSocket. Para se conectar ao servidor WebSocket, abra uma conexão WebSocket para `ws://localhost:9393`.

Uma vez conectado, você pode enviar mensagens para atualizar as propriedades ou receber notificações quando o evento "high-temperature"é acionado.

2.4.2 subscriber

Arquivo **client.py** (Inscrição)

Este é um cliente WoT que consome uma entidade WoT a partir de sua URL de descrição de coisa (Thing Description URL) e se inscreve em todas as propriedades e eventos observáveis da entidade consumida.

Execute o seguinte comando para iniciar o cliente:

```
python wot_client.py --url <thing_description_url> --time <total_subscription_time>
```

Substitua $< thing_description_url >$ pela URL da descrição da coisa que deseja consumir e $< total_subscription_time >$ pelo tempo total de inscrição em segundos.

O cliente WoT de inscrição irá consumir a entidade WoT, se inscrever em todas as propriedades e eventos observáveis e imprimir os valores recebidos em tempo real. O cliente permanecerá inscrito por um período de tempo especificado pelo argumento –time.

Aula 5: Especificações e Dependências

10 Maio 2023

2.5 Semana 05

2.5.1 Especificações do W3C-WoT

https://www.w3.org/TR/wot-architecture11/

https://www.w3.org/TR/wot-usecases/

https://www.w3.org/TR/wot-thing-description/

WoT Thing Description [WOT-THING-DESCRIPTION]: fornece um formato de dados legível por máquina para descrever a metadados e interfaces voltadas para a rede de Things.

WoT Binding Templates [WOT-BINDING-TEMPLATES]: oferece diretrizes informativas sobre como definir interfaces voltadas para a rede em Things para protocolos específicos e ecossistemas IoT.

WoT Discovery [WOT-DISCOVERY]: define um mecanismo de distribuição para metadados WoT (Thing Descriptions).

WoT Scripting API [WOT-SCRIPTING-API]: habilita a implementação da lógica de aplicação de um Thing usando uma API JavaScript comum semelhante às APIs do navegador da Web.

WoT Security and Privacy Guidelines [WOT-SECURITY]: fornece diretrizes para a implementação segura e configuração de Things e discute questões que devem ser consideradas em sistemas que implementam a W3C WoT.

2.5.2 Dependências do W3C-WoTPy

O arquivo setup.py define um conjunto de dependências de terceiros necessárias para a execução da biblioteca WotPy. Algumas dessas dependências são obrigatórias, enquanto outras são opcionais e só serão instaladas se estiverem disponíveis no sistema.

A seguir, estão as principais dependências definidas no arquivo setup.py:

tornado: uma biblioteca assíncrona usada para criar aplicativos Web em Python. É usada pela WotPy para criar servidores HTTP e WebSocket.

jsonschema: uma biblioteca que valida esquemas JSON e os dados JSON correspondentes. A WotPy usa essa biblioteca para validar as descrições de coisa (Thing Descriptions) recebidas e produzidas.

six: uma biblioteca que permite escrever código Python 2 e Python 3 compatível com ambos os ambientes. É usada pela WotPy para garantir a compatibilidade com as duas versões do Python.

rx: uma biblioteca de programação reativa que permite escrever código que responde a mudanças de estado de forma assíncrona. É usada pela WotPy para suportar a API WoT Scripting e as interações com as propriedades e eventos observáveis.

python-slugify: uma biblioteca que converte strings para "slug", um formato de texto que usa somente caracteres ASCII, números e traços. É usada pela WotPy para criar identificadores únicos para as coisas.

Existem também algumas dependências opcionais que a WotPy usa se estiverem instaladas no sistema:

aiocoap: uma biblioteca Python para o protocolo de transferência de dados Constrained Application Protocol (CoAP), usada pela WotPy para suportar o protocolo CoAP.

hbmqtt: uma biblioteca Python para implementar o protocolo Message Queue Telemetry Transport (MQTT), usada pela WotPy para suportar o protocolo MQTT.

websockets: uma biblioteca Python para suportar a comunicação WebSocket, usada pela WotPy para suportar o protocolo WebSocket.

zeroconf: uma biblioteca Python para suportar o protocolo DNS Service Discovery (DNS-SD), usada pela WotPy para descobrir serviços e dispositivos na rede.

Essas dependências são verificadas em tempo de execução e adicionadas ao conjunto de dependências, se estiverem disponíveis no sistema.

Aula 6: ESP32

17 Maio 2023

2.6 Semana 06

2.6.1 Sensor Solar

https://t16k-ach2157.readthedocs.io/en/latest/

Antes de começar, serão necessários os seguintes componentes: https://github.com/T16K/ACH2157#lista-de-materiais

- Sensor UV ML8511
- Microcontrolador ESP32
- Jumpers e protoboard para a conexão
- Ambiente de desenvolvimento Python configurado com WoTPy

Conectar o sensor UV ML8511 ao ESP32: https://github.com/T16K/ACH2157#conex%C3%B5es

- GND (ML8511) para GND (ESP32)
- VCC (ML8511) para 3V3 (ESP32)
- OUT (ML8511) para um pino analógico 34 (ESP32)

Instalar a biblioteca do WoTPy. https://github.com/T16K/wot-py

Escrever um código que leia os dados do sensor UV ML8511 e os exiba no ESP32, usando a linguagem de programação MicroPython, que é uma versão do Python 3 otimizada para microcontroladores como o ESP32. https://t16k-ach2157.readthedocs.io/en/latest/comp/esp.html#introduzindo-o-micropython

Usar o WoTPy para expor os dados do sensor UV como um "Thing"na Web of Things. Isso permitirá que outros dispositivos e aplicações IoT interajam com seu sensor UV de maneira padronizada.

Testar as configuração para garantir que tudo esteja funcionando corretamente. É possível fazer isso usando o WoTPy para descobrir o sensor UV na rede e lendo os dados do sensor.

2.6.2 Expor os Dados do Sensor

Para expor os dados do sensor usando WoTPy, criar um Thing que represente o sensor. Cada Thing tem um conjunto de propriedades, ações e eventos que descrevem suas capacidades. Neste caso, criar uma propriedade para representar o valor lido do sensor UV.

Adicionar o WoTPy para expor esse valor como uma propriedade em um Thing. Exemplo simplificado:

```
from wotpy.wot.servient import Servient
from wotpy.wot.thing import Thing
from wotpy.wot.forms import Form
from wotpy.protocols.http.server import HTTPServer

# Crie um novo Thing
uv_sensor_thing = Thing(id="urn:dev:ops:uv-sensor-1234",
    title="UV Sensor")
```

```
# Adicione uma propriedade ao Thing para representar o
    valor do sensor UV
uv_sensor_thing.add_property(name="uvValue", description
    ={"@type": "number", "unit": "UV Index"}, value=0,
    forms=[Form(href="/uvValue", op="readproperty")])

# Este valor sera atualizado pelo seu codigo de leitura
    de sensor

# Crie um servient e adicione o Thing
servient = Servient()
servient.add_thing(uv_sensor_thing)

# Inicie o servidor HTTP para expor o Thing na rede
http_server = HTTPServer(port=8080)
http_server.start(servient)

# Agora o Thing esta disponivel em http://localhost:8080/
    uvValue
```

Este é um exemplo simplificado. Em um caso real, provavelmente teria que adicionar um loop que continuamente lê o valor do sensor UV e atualiza a propriedade uvValue.

Além disso, precisaria adicionar segurança e manipulação de erros apropriadas. Por exemplo, restringir o acesso à propriedade uvValue para apenas certos dispositivos ou usuários.

Finalmente, configurar o seu ESP32 para que ele envie os dados do sensor para o servidor WoTPy, possivelmente através de um protocolo como MQTT ou HTTP.

2.6.3 cliente HTTP

É necessário definir primeiro o tipo de entidade no nosso contexto: cliente ou servidor.

No caso de um servidor, a instalação do WoTPy é necessária. Note que para esta configuração, não há necessidade de um gateway.

Já para um cliente, um gateway se torna essencial. Este atua como intermediário entre o ESP32 e o mundo externo, facilitando a comunicação entre os dois.

Para comunicar os valores do sensor ML8511 usando o ESP32 e o WoTPy:

Leitura do sensor ML8511 com o ESP32: Primeiro, configurar o ESP32 para ler os valores do sensor ML8511. Programando o ESP32 com um código que lê os

valores do sensor a intervalos regulares. Este código será executado no ambiente de desenvolvimento **MicroPython**.

Servidor no ESP32: O ESP32 é programado para funcionar como um servidor WoT. Ele expõe a propriedade que representa o valor do sensor ML8511 através de um protocolo de rede, como HTTP.

Comunicação de valores do sensor: Sempre que o ESP32 lê um valor do sensor, ele atualiza a propriedade do sensor no servidor WoT. Dependendo da sua implementação, isso pode envolver o envio de um POST request ao servidor WoTPy ou atualizar o valor localmente se o servidor WoT estiver sendo executado no próprio ESP32.

Criação de um Thing WoTPy para representar o sensor: No servidor WoTPy, criar um Thing que represente o sensor ML8511, utilizando a biblioteca WoTPy Python. O Thing deve ter uma propriedade que represente o valor atual do sensor.

Atualização do valor do sensor no Thing WoTPy: Cada vez que o ESP32 enviar um novo valor do sensor para o servidor WoTPy, atualizar o valor da propriedade no Thing WoTPy que representa o sensor. Isto pode ser feito manipulando a requisição POST que o ESP32 envia e atualizando o valor da propriedade apropriadamente.

Exposição do Thing WoTPy na rede: Finalmente, você configurar o WoTPy para expor o Thing na rede. Isto permitirá que outros dispositivos e serviços descubram o Thing e leiam o valor do sensor.

Interagindo com o Thing: Os clientes na rede, que também podem ser implementados usando o WoTPy ou qualquer outra biblioteca que suporte o padrão WoT, podem agora interagir com o Thing. Eles podem ler o valor atual do sensor, assinar atualizações de valor e, se suportado, acionar ações no Thing.

Aula 7: Servidor WoT

31 Maio 2023

2.7 Semana 07

O Wotpy é uma implementação do protocolo Web of Things (WoT) da W3C. A biblioteca foi escrita em Python e é compatível com Python 3.7 ou superior. A princípio, é possível executar essa biblioteca em qualquer ambiente onde possa executar Python. No entanto, é importante lembrar que algumas bibliotecas Python podem ter dependências de sistema que não estão disponíveis em todas as plataformas.

O MicroPython é uma versão compacta do Python 3 projetada para rodar em microcontroladores como o ESP32. Embora seja muito poderoso e flexível, ele não suporta todas as bibliotecas e recursos do Python completo devido à limitações de memória e processamento de microcontroladores.

A biblioteca Wotpy parece ser bastante complexa e pode exigir recursos que não estão disponíveis no MicroPython. Portanto, é possível que você não consiga instalar e executar essa biblioteca no MicroPython no ESP32.

2.7.1 Servidor ESP32

Para confirmar a compatibilidade, tentei instalar a biblioteca diretamente no ambiente MicroPython, usando o *rshell* para transferir o arquivo *setup.py* do WoTPy. Este arquivo é usado para instalar a biblioteca em um ambiente Python completo. No entanto, esse arquivo não foi adequado para ser carregado diretamente para o ESP32.

Por isso, tentei modificar a biblioteca para torná-la compatível com o MicroPython. Mas seria um processo complexo e que consumiria muito tempo. Aqui estão algumas considerações:

- Dependências: As dependências de uma biblioteca devem ser compatíveis com o MicroPython. Muitas bibliotecas Python padrão, como jsonschema e tornado no caso do wotpy, não funcionam no MicroPython devido a suas complexidades e ao uso de recursos que não estão disponíveis nos microcontroladores. Seria preciso encontrar ou criar versões alternativas dessas bibliotecas que sejam compatíveis com o MicroPython.
- Recursos: As bibliotecas Python padrão podem usar recursos e módulos que não estão disponíveis no MicroPython, como threading ou multiprocessing. Seria preciso reescrever partes da biblioteca para não usar esses recursos, ou encontrar maneiras alternativas de implementar a mesma funcionalidade.
- Tamanho da memória: O MicroPython é projetado para dispositivos com recursos de memória limitados. Portanto, precisaria garantir que sua biblioteca não consuma muita memória. Isso pode envolver reescrever partes da biblioteca para ser mais eficiente em termos de memória.
- Testes: Depois de modificar a biblioteca, precisaria testá-la extensivamente para garantir que ainda funcione como esperado. Isso pode ser difícil, pois os erros podem ser menos previsíveis e mais difíceis de depurar no MicroPython do que no Python padrão.

Dada a complexidade envolvida na modificação da biblioteca para torná-la compatível com o MicroPython, geralmente é mais fácil procurar bibliotecas alternativas que já são compatíveis ou projetadas especificamente para o MicroPython.

O ESP32 é um dispositivo de hardware limitado e pode não ser capaz de lidar com a complexidade da implementação completa de uma biblioteca como o wotpy. Em alguns casos, pode ser mais apropriado usar um dispositivo mais poderoso, como um Raspberry Pi, que pode executar o Python padrão.

2.7.2 Cliente ESP32

Usar o ESP32 como cliente em uma configuração de Internet das Coisas (IoT), com um servidor mais poderoso rodando o wotpy ou outra implementação do protocolo WoT.

- Configurar o servidor WoT: Configurar um servidor WoT em um dispositivo que suporte Python completo, como um computador. Este servidor poderia rodar o wotpy ou outra implementação do protocolo WoT.
- Definir a interação com o ESP32: Descrever as interações que você deseja ter com o ESP32 na forma de "Thing Description"(TD) do WoT. Isso pode incluir a leitura de valores de sensores no ESP32, o controle de atuadores ligados ao ESP32, etc.
- Implementar o cliente no ESP32: Em seguida, precisaria implementar o cliente no ESP32 usando o MicroPython. Esse cliente se comunicaria com o servidor WoT para ler ou escrever valores conforme definido na TD.

Como o ESP32 está rodando o MicroPython, você terá que usar bibliotecas que sejam compatíveis com o MicroPython. Para se comunicar com o servidor WoT, vou usar o HTTP, que é suportados pelo MicroPython.

No entanto, essa abordagem significa que a funcionalidade do ESP32 será limitada àquela definida pelo servidor WoT. O ESP32 basicamente atuaria como um dispositivo de borda ou um nó de sensor em uma rede de IoT, com o servidor WoT fornecendo a principal funcionalidade de aplicativo.

Aula 8: Processo de Comunicação

05 Junho 2023

2.8 Semana 08

A situação envolve um dispositivo ESP32 equipado com um sensor ML8511 (sensor UV) atuando como cliente, enquanto um computador atua como servidor utilizando a biblioteca WoTPy.

- Configuração do ESP32 e ML8511: O primeiro passo envolve configurar o ESP32 para coletar dados do sensor ML8511. Isso geralmente envolve escrever um script em MicroPython (ou outro firmware suportado pelo ESP32) para ler dados do sensor ML8511.
- Exposição de Dados via HTTP: Uma vez que os dados estejam sendo coletados corretamente, o ESP32 deve então ser programado para enviar esses dados para o servidor. Isso pode ser realizado através do protocolo HTTP, com o ESP32 atuando como cliente. A comunicação HTTP envolve a criação de um POST request contendo os dados do sensor, que é então enviado para o servidor.
- Configuração do Servidor WoTPy: O próximo passo envolve configurar o servidor no computador utilizando a biblioteca WoTPy. Isso inclui definir

uma Thing Description (TD) que descreve os recursos do ESP32 e do sensor ML8511, bem como configurar os servidores HTTP para aceitar as solicitações do cliente ESP32.

- Recepção e Processamento de Dados: Uma vez que o servidor WoTPy está configurado e funcionando, ele pode começar a receber dados do ESP32. Quando uma solicitação POST é recebida, o servidor extrai os dados do sensor do corpo da solicitação, processa esses dados conforme necessário, e atualiza a Thing Description (TD) com as leituras atuais do sensor.
- Interação com a Web of Things (WoT): Com a TD atualizada, outros dispositivos e serviços na Web of Things (WoT) podem agora interagir com o sensor ML8511 por meio do servidor WoTPy. Eles podem ler as últimas leituras do sensor, solicitar atualizações de dados, ou até mesmo enviar comandos para o ESP32, dependendo de como a TD foi configurada.

2.8.1 Gateway

Um gateway na Internet das Coisas (IoT) atua como um ponto de conexão entre a nuvem (ou servidor) e os dispositivos, sensores e atuadores no campo. Em relação ao WoT e WoTPy, o gateway desempenha várias funções essenciais:

Protocol Translation: O gateway IoT pode servir como um tradutor de protocolo, facilitando a comunicação entre dispositivos que usam diferentes protocolos de comunicação. Por exemplo, seu dispositivo ESP32 com sensor ML8511 pode usar HTTP ou MQTT para se comunicar, enquanto outros dispositivos em sua rede podem usar CoAP, WebSocket ou outros protocolos. O gateway WoT pode traduzir entre esses protocolos conforme necessário.

Data Aggregation and Preprocessing: O gateway pode agregar e pré-processar dados de múltiplos dispositivos antes de enviá-los para a nuvem. Isso pode envolver a combinação de dados de vários sensores, a realização de cálculos de nível básico nos dados ou a redução da quantidade de dados enviados para a nuvem.

Device Management: O gateway pode fornecer funcionalidades de gerenciamento de dispositivos, como a configuração de dispositivos, atualizações de firmware ou monitoramento do estado do dispositivo.

Security: O gateway é um ponto crucial para a segurança em uma rede IoT. Ele pode fornecer funções como autenticação e autorização de dispositivos, criptografia de dados e proteção contra ameaças de segurança.

Edge Computing: Alguns gateways podem realizar computação de borda, processando dados localmente em vez de enviá-los para a nuvem. Isso pode melhorar a latência, a privacidade dos dados e a eficiência do uso da largura de banda.

No contexto do sistema ESP32 e ML8511 com WoTPy, o gateway (neste caso, o computador) estará recebendo dados do sensor ML8511 via ESP32, exporá

esses dados na Web of Things e fornecerá um ponto de acesso para que outros clientes WoT acessem esses dados.

2.8.2 Thing Description

A "Thing Description"(TD) é um dos elementos fundamentais na arquitetura da Web of Things (WoT). Essencialmente, a TD é uma representação de alto nível do dispositivo ou "Thing"na Web of Things. Ela fornece um conjunto de metadados sobre o dispositivo e descreve suas capacidades em termos de propriedades, ações e eventos.

Metadata: Isso inclui informações gerais sobre o dispositivo, como seu nome, tipo de dispositivo e qualquer outra informação que possa ser útil para os clientes ou para outros dispositivos na rede.

Properties: As propriedades representam o estado atual do dispositivo. Por exemplo, para um sensor de luz, uma propriedade pode ser o valor atual da luz detectada.

Actions: As ações representam as funcionalidades que podem ser executadas no dispositivo. Por exemplo, um dispositivo de luz inteligente pode ter ações como 'ligar' e 'desligar'.

Events: Os eventos representam as notificações ou os alertas que o dispositivo pode enviar. Por exemplo, um sensor de temperatura pode enviar um evento quando a temperatura ultrapassa um determinado limite.

A Thing Description é formatada como um documento JSON-LD, o que significa que ela pode ser facilmente lida por humanos e máquinas, e pode ser incorporada em uma variedade de sistemas. A TD permite que os dispositivos IoT se comuniquem e interajam entre si, independentemente do protocolo de rede ou da tecnologia subjacente que eles utilizam.

No contexto do sistema usando o ESP32 e o sensor ML8511, seria criado uma Thing Description para representar o sensor ML8511, incluindo metadados sobre o sensor e descrevendo suas propriedades (como o valor atual do UV), possíveis ações (se houver) e quaisquer eventos que o sensor possa emitir.

Aula 8: Processo de Comunicação

05 Junho 2023

2.9 Semana 08

A situação envolve um dispositivo ESP32 equipado com um sensor ML8511 (sensor UV) atuando como cliente, enquanto um computador atua como servidor utilizando a biblioteca WoTPy.

- Configuração do ESP32 e ML8511: O primeiro passo envolve configurar o ESP32 para coletar dados do sensor ML8511. Isso geralmente envolve

escrever um script em MicroPython (ou outro firmware suportado pelo ESP32) para ler dados do sensor ML8511.

- Exposição de Dados via HTTP: Uma vez que os dados estejam sendo coletados corretamente, o ESP32 deve então ser programado para enviar esses dados para o servidor. Isso pode ser realizado através do protocolo HTTP, com o ESP32 atuando como cliente. A comunicação HTTP envolve a criação de um POST request contendo os dados do sensor, que é então enviado para o servidor.
- Configuração do Servidor WoTPy: O próximo passo envolve configurar o servidor no computador utilizando a biblioteca WoTPy. Isso inclui definir uma Thing Description (TD) que descreve os recursos do ESP32 e do sensor ML8511, bem como configurar os servidores HTTP para aceitar as solicitações do cliente ESP32.
- Recepção e Processamento de Dados: Uma vez que o servidor WoTPy está configurado e funcionando, ele pode começar a receber dados do ESP32. Quando uma solicitação POST é recebida, o servidor extrai os dados do sensor do corpo da solicitação, processa esses dados conforme necessário, e atualiza a Thing Description (TD) com as leituras atuais do sensor.
- Interação com a Web of Things (WoT): Com a TD atualizada, outros dispositivos e serviços na Web of Things (WoT) podem agora interagir com o sensor ML8511 por meio do servidor WoTPy. Eles podem ler as últimas leituras do sensor, solicitar atualizações de dados, ou até mesmo enviar comandos para o ESP32, dependendo de como a TD foi configurada.

2.9.1 Gateway

Um gateway na Internet das Coisas (IoT) atua como um ponto de conexão entre a nuvem (ou servidor) e os dispositivos, sensores e atuadores no campo. Em relação ao WoT e WoTPy, o gateway desempenha várias funções essenciais:

Protocol Translation: O gateway IoT pode servir como um tradutor de protocolo, facilitando a comunicação entre dispositivos que usam diferentes protocolos de comunicação. Por exemplo, seu dispositivo ESP32 com sensor ML8511 pode usar HTTP ou MQTT para se comunicar, enquanto outros dispositivos em sua rede podem usar CoAP, WebSocket ou outros protocolos. O gateway WoT pode traduzir entre esses protocolos conforme necessário.

Data Aggregation and Preprocessing: O gateway pode agregar e pré-processar dados de múltiplos dispositivos antes de enviá-los para a nuvem. Isso pode envolver a combinação de dados de vários sensores, a realização de cálculos de nível básico nos dados ou a redução da quantidade de dados enviados para a nuvem.

Device Management: O gateway pode fornecer funcionalidades de gerenciamento de dispositivos, como a configuração de dispositivos, atualizações de

firmware ou monitoramento do estado do dispositivo.

Security: O gateway é um ponto crucial para a segurança em uma rede IoT. Ele pode fornecer funções como autenticação e autorização de dispositivos, criptografia de dados e proteção contra ameaças de segurança.

Edge Computing: Alguns gateways podem realizar computação de borda, processando dados localmente em vez de enviá-los para a nuvem. Isso pode melhorar a latência, a privacidade dos dados e a eficiência do uso da largura de banda.

No contexto do sistema ESP32 e ML8511 com WoTPy, o gateway (neste caso, o computador) estará recebendo dados do sensor ML8511 via ESP32, exporá esses dados na Web of Things e fornecerá um ponto de acesso para que outros clientes WoT acessem esses dados.

2.9.2 Thing Description

A "Thing Description"(TD) é um dos elementos fundamentais na arquitetura da Web of Things (WoT). Essencialmente, a TD é uma representação de alto nível do dispositivo ou "Thing"na Web of Things. Ela fornece um conjunto de metadados sobre o dispositivo e descreve suas capacidades em termos de propriedades, ações e eventos.

Metadata: Isso inclui informações gerais sobre o dispositivo, como seu nome, tipo de dispositivo e qualquer outra informação que possa ser útil para os clientes ou para outros dispositivos na rede.

Properties: As propriedades representam o estado atual do dispositivo. Por exemplo, para um sensor de luz, uma propriedade pode ser o valor atual da luz detectada.

Actions: As ações representam as funcionalidades que podem ser executadas no dispositivo. Por exemplo, um dispositivo de luz inteligente pode ter ações como 'ligar' e 'desligar'.

Events: Os eventos representam as notificações ou os alertas que o dispositivo pode enviar. Por exemplo, um sensor de temperatura pode enviar um evento quando a temperatura ultrapassa um determinado limite.

A Thing Description é formatada como um documento JSON-LD, o que significa que ela pode ser facilmente lida por humanos e máquinas, e pode ser incorporada em uma variedade de sistemas. A TD permite que os dispositivos IoT se comuniquem e interajam entre si, independentemente do protocolo de rede ou da tecnologia subjacente que eles utilizam.

No contexto do sistema usando o ESP32 e o sensor ML8511, seria criado uma Thing Description para representar o sensor ML8511, incluindo metadados sobre o sensor e descrevendo suas propriedades (como o valor atual do UV), possíveis ações (se houver) e quaisquer eventos que o sensor possa emitir.

Aula 9 12 Junho 2023

2.10 Semana 09

O objetivo é estabelecer uma comunicação básica entre o ESP32 e o servidor, utilizando HTTP para transmitir os dados do sensor UV.

- O script boot.py no ESP32 será responsável por estabelecer a conexão com a rede WiFi.
- O script main.py no ESP32 será encarregado de ler o valor do sensor UV e transmitir esse valor para o servidor por meio de um pedido HTTP POST.
- O servidor, que executa o script server.py, receberá e processará os valores do sensor UV que são enviados pelo ESP32.
- https://github.com/T16K/wot-py/commit/34afc51099e2bec61f61d9fc7daf971a54dbc856

2.10.1 Servidor

Encontrei problemas na abertura das portas do servidor. Para solucionar isso, testei o programa server.py a partir do exemplo *temperature*, com o objetivo de acessar o servidor WoT (Web of Things) criado. O servidor escuta nas seguintes portas especificadas:

- Porta 9393 para conexões WebSocket.
- Porta 9494 para conexões HTTP.

Contudo, ao tentar realizar uma solicitação GET para http://localhost:9494/, a conexão com o localhost foi recusada.

Essa situação não havia ocorrido antes de começar a executar os scripts dentro da imagem do Docker, então deduzi que este poderia ser o problema. Após pesquisar, encontrei a solução no link https://docs.docker.com/engine/reference/run/#expose-incoming-ports.

Portanto, modifiquei o comando original para:

```
docker container run -p 9090:9090 -p 9393:9393 -p
    9494:9494 -ti --rm -v $PWD:/asdf 7af5d5d07d93 sh
```

No entanto, percebi outro problema: a configuração de rede do Docker não estava utilizando a mesma rede local do ESP32. Para diagnosticar esse problema, utilizei o comando "ip a". Para resolver, bastou utilizar a opção --network host"e excluir a opção -p".

2.10.2 Função POST

Com relação à função POST, estou enfrentando dois problemas principais. No terminal do servidor, estou recebendo as seguintes mensagens:

```
404 GET /uv (::1) 0.64ms
404 GET /favicon.ico (::1) 0.64ms
```

E, no rshell, ao usar o comando "repl"para comunicar com o ESP32, obtenho:

```
[Errno 113] ECONNABORTED
[Errno 118] EHOSTUNREACH
```

Aula 10: Atualização nos programas

19 Junho 2023

2.11 Semana 10

https://github.com/agmangas/wot-py/commit/a7cf9b8aa81d871dbe9264730f5ad88dfcfc60b9

2.11.1 boot.py

Removi a inicialização do webrepl, pois não incluí a biblioteca correspondente no ESP32.

2.11.2 main.py

Adicionei uma verificação para o status da resposta HTTP após o envio dos dados. Se o status estiver entre 200 e 299, o sistema imprimirá uma mensagem de sucesso. Além disso, o intervalo de sleep está configurado para 10 segundos.

2.11.3 server.py

Incorporei mais dois endpoints, /uvValue e /favicon.ico. O endpoint /uvValue devolve a última leitura UV armazenada no formato JSON e o /favicon.ico é usado para gerir solicitações de favicon feitas pelos navegadores. Adicionalmente, a página HTML gerada pelo GET no endpoint /uv agora inclui um script JavaScript para atualizar a leitura UV automaticamente a cada 2 segundos.

Aula 11: Implementação WoTPy

26 Junho 2023

2.12 Semana 11

Plano para a implementação do WoTPy no arquivo ”server.py”:

- Iniciar o Servient. Adicionar servidores HTTP e WebSocket ao Servient, se necessário.
- Produzir um Thing com base na descrição TD.
- Definir manipuladores personalizados para as propriedades e eventos do Thing.
- Expor o Thing.

- https://github.com/agmangas/wot-py/commit/3dd4f7e428bd565970adf96f8f4482cf986496ea

Anteriormente eu criei um servidor web com Tornado para ler e registrar leituras de UV. Para utilizar o WoTPy, penso em transformar essa funcionalidade em uma "Thing"do WoT.

A implementação WoTPy é usada apenas no servidor (neste caso, no seu servidor Python onde as leituras UV são recebidas e processadas). O ESP32, ou qualquer outro dispositivo IoT, não precisa suportar WoTPy ou mesmo Python. O dispositivo só precisa ser capaz de enviar suas leituras para o servidor de uma maneira que o servidor possa interpretar.

Neste caso, o ESP32 está enviando as leituras UV para o servidor como um POST HTTP com o corpo da solicitação contendo um JSON. O método POST geralmente não é usado para interagir diretamente com uma propriedade em WoT. Em vez disso, é frequentemente usado para interagir com ações em um Thing, que são operações mais complexas que podem envolver vários parâmetros de entrada e produzir um resultado.

No padrão Web of Things (WoT) da W3C, a forma padrão de interagir com uma propriedade de um Thing é através de métodos HTTP GET (para ler o valor da propriedade) e PUT (para atualizar o valor da propriedade). Estes são os métodos recomendados pela especificação WoT e são os que a biblioteca WoTPy implementa para expor propriedades de Things.

Ajustar o código no ESP32 para seguir as recomendações da W3C envolve usar o método HTTP PUT para atualizar a propriedade 'uv' do Thing.

https://github.com/agmangas/wot-py/commit/0613417f2ffbae386e45576d9bf48721b37c8483

2.12.1 server.py

No servidor, a biblioteca wotpy é usada para criar um servidor WoT que expõe uma "Coisa"com uma propriedade de UV. Esta "Coisa"possui um manipulador de leitura e de gravação para a propriedade de UV, que são utilizados para ler e atualizar os valores do sensor. O servidor utiliza a biblioteca tornado para gerenciar operações assíncronas.

```
import json
import logging
import tornado.gen
from tornado.ioloop import IOLoop
from wotpy.protocols.http.server import HTTPServer
from wotpy.wot.servient import Servient
```

Primeiro as declarações de importação. Elas carregam diferentes módulos e pacotes que o código precisa para funcionar corretamente. Aqui está o que cada um deles faz:

import json: O módulo json é uma biblioteca padrão do Python para trabalhar

com dados JSON (JavaScript Object Notation). Ele fornece métodos para analisar dados JSON para converter em um objeto Python (como um dicionário ou lista) e para converter objetos Python em strings JSON. Usada na função uv_read_handler para decodificar a string de dados UV em formato JSON para um objeto Python (uv_data_dict = json.loads(GLOBAL_UV_DATA.decode("utf-8"))). E também usada na função main para criar um Thing WoT a partir da descrição fornecida (wot.produce(json.dumps(DESCRIPTION))).

import logging: O módulo logging é uma biblioteca padrão do Python para configuração de registros de eventos em seu aplicativo. Ele pode registrar eventos em diferentes níveis de severidade e direcionar esses registros para uma variedade de destinos de saída. Usada em todo o código para registrar informações importantes e úteis para fins de depuração e rastreamento. Por exemplo, LOGGER.info("Reading UV data.").

import tornado.gen: tornado.gen é uma parte do framework Tornado para Python. É um módulo que contém utilitários para facilitar o uso de coroutines e outras funções que retornam Futures. Usada para definir uv_read_handler, uv_write_handler e main como funções coroutine que são especiais no Python, elas podem ser pausadas e retomadas, permitindo comportamento assíncrono.

from tornado.ioloop import IOLoop: IOLoop é a principal classe do Tornado para gerenciar eventos de E/S. Cada thread normalmente tem exatamente um IOLoop. IOLoop.current().add_callback(main) e IOLoop.current().start() são chamadas na parte principal do script para iniciar o loop de eventos I/O do Tornado, que permite o comportamento assíncrono.

from wotpy.protocols.http.server import HTTPServer: Esta linha importa a classe HTTPServer do pacote wotpy.protocols.http.server. Esta classe é usada para criar um servidor HTTP. HTTPServer é usado para criar um servidor HTTP na função main (http_server = HTTPServer(port=HTTP_PORT)).

from wotpy.wot.servient import Servient: Esta linha importa a classe Servient do pacote wotpy.wot.servient. Uma instância da classe Servient atua como um cliente e servidor em uma rede Web of Things (WoT). Servient é usado para criar um serviente WoT na função main (servient = Servient()).

```
logging.basicConfig()
LOGGER = logging.getLogger()
LOGGER.setLevel(logging.INFO)
```

Em seguida configurar o registro (logging) em Python. Incluir configurações de registro (logging) no código é fundamental para o monitoramento, depuração e rastreamento de atividades dentro de um aplicativo. Aqui está uma explicação detalhada do que cada linha faz:

logging.basicConfig(): Esta função configura o sistema de registro para seu aplicativo. Sem nenhuma configuração específica passada como argumentos para basicConfig(), ela irá criar um registro padrão que escreve as mensagens de log no console (ou seja, a saída padrão). Esta função só deve ser chamada uma vez,

e se for chamada mais de uma vez, a primeira chamada determinará o formato e a configuração do registro.

LOGGER = logging.getLogger(): Esta função retorna uma referência para um objeto logger. Um logger é o objeto que as aplicações usam diretamente para fazer chamadas de log. Se nenhuma configuração de nome for fornecida para getLogger(), como no seu caso, ela retornará o logger de nível raiz.

LOGGER.setLevel(logging.INFO): Isso configura o nível de severidade do logger para INFO. Os níveis de severidade do log, de menor para maior, são: DEBUG, INFO, WARNING, ERROR e CRITICAL. Ao definir o nível do logger para INFO, todas as mensagens de log com nível INFO e acima (ou seja, WARNING, ERROR e CRITICAL) serão registradas. As mensagens de log com nível DEBUG serão ignoradas.

```
ID_THING = "urn:uvthing"
NAME_PROP_UV = "uv"

DESCRIPTION = {
    "id": ID_THING,
    "name": ID_THING,
    "properties": {
        NAME_PROP_UV: {
            "type": "number",
            "readOnly": False,
            "observable": True
        }
    }
}
```

Neste trecho de código, estamos definindo alguns identificadores e uma descrição para a "Coisa"(Thing) WoT que estamos expondo. Vamos analisar cada linha:

ID_THING = "urn:uvthing": Aqui, um identificador único (URN - Uniform Resource Name) para a "Coisa"WoT está sendo definido. Este identificador será usado para se referir a esta "Coisa"em particular na rede WoT.

NAME_PROP_UV = "uv": Este é o nome da propriedade que estamos definindo para a "Coisa"WoT. Neste caso, a propriedade é "uv", o que provavelmente se refere ao valor de ultravioleta que a "Coisa"fornece.

DESCRIPTION = ...: Aqui, uma descrição completa da "Coisa"WoT está sendo definida. Esta descrição inclui o id e o name da "Coisa", bem como uma lista de suas properties (propriedades).

Dentro das properties, estamos definindo uma propriedade chamada uv (o valor que definimos anteriormente para NAME_PROP_UV). Esta propriedade tem um type de "number", o que significa que o valor que ela fornece será numérico. Ela também tem uma configuração readOnly definida como False, o que significa que essa propriedade pode ser alterada, e uma configuração observable definida

como True, o que significa que os clientes podem se inscrever para receber notificações sobre alterações nesta propriedade.

Em resumo, estas linhas de código definem a "Coisa"WoT que será exposta pela nossa aplicação, incluindo o seu identificador, o nome e as características da propriedade que ela fornece.

```
@tornado.gen.coroutine
def uv_read_handler():
    LOGGER.info("Reading UV data.")
    if GLOBAL_UV_DATA is None:
        return
    uv_data_dict = json.loads(GLOBAL_UV_DATA.decode("utf-8"))
    uv_data = float(uv_data_dict['uv'])
    raise tornado.gen.Return(uv_data)
```

A função uv_read_handler é um manipulador personalizado para a propriedade 'UV'. Ela é definida como uma 'coroutine' com a ajuda do decorator @tornado.gen.coroutine, o que permite que a função seja assíncrona, ou seja, pode ser pausada e retomada, permitindo a execução de outras tarefas no meio tempo.

Vamos olhar mais de perto para a função:

LOGGER.info("Reading UV data."): Isto é apenas um registro de informação para sinalizar que a função começou a ler os dados UV.

if GLOBAL_UV_DATA is None: return: Este é um controle de fluxo para verificar se GLOBAL_UV_DATA é None. Se for None, a função retorna imediatamente e termina. Esta é provavelmente uma medida de segurança para garantir que o programa não tente acessar ou trabalhar com dados não existentes.

uv_data_dict = json.loads(GLOBAL_UV_DATA.decode("utf-8")): Aqui, os dados UV globais são decodificados de uma string binária para uma string normal usando o método decode("utf-8") e, em seguida, são convertidos de uma string JSON para um dicionário Python usando json.loads().

uv_data = float(uv_data_dict['uv']): Aqui, o valor UV real é extraído do dicionário uv_data_dict e convertido para um float.

raise tornado.gen.Return(uv_data): Finalmente, o valor UV é retornado como o resultado da função coroutine. A instrução raise tornado.gen.Return(uv_data) é equivalente a return uv_data em uma função normal, mas é a maneira recomendada de retornar valores de coroutines no Tornado.

Então, basicamente, essa função é chamada quando algum cliente quer ler o valor atual da propriedade UV da "Coisa"WoT. A função verifica se os dados UV globais existem e, se existirem, extrai o valor UV, converte-o em um float e o retorna.

```
@tornado.gen.coroutine
def uv\_write\_handler(value):
    global GLOBAL\_UV\_DATA
    LOGGER.info("Writing UV data.")
    GLOBAL\_UV\_DATA = value
    LOGGER.info("UV data updated to: {}".format(GLOBAL\_UV\_DATA))
```

A função uv_write_handler é um manipulador personalizado para escrever os dados UV. É definida como uma coroutine com a ajuda do decorador @tornado.gen.coroutine, o que significa que pode ser uma função assíncrona (embora neste caso específico não haja operações assíncronas na função).

Aqui está o que acontece nesta função:

global GLOBAL_UV_DATA: Esta linha permite que a função acesse a variável global GLOBAL_UV_DATA. Sem esta linha, qualquer referência a GLOBAL_UV_DATA na função seria tratada como uma referência a uma nova variável local com o mesmo nome, e não à variável global.

LOGGER.info("Writing UV data."): Esta é apenas uma mensagem de log para sinalizar que a função começou a escrever os dados UV.

GLOBAL_UV_DATA = value: Aqui, a função atualiza a variável global GLOBAL_UV_DATA com o novo valor de UV fornecido à função. Este valor provavelmente é uma string JSON que representa os dados UV.

LOGGER.info("UV data updated to: ".format(GLOBAL_UV_DATA)): Esta é uma outra mensagem de log para confirmar que os dados UV foram atualizados.

Então, basicamente, essa função é chamada quando algum cliente quer atualizar o valor da propriedade UV da "Coisa"WoT. A função recebe o novo valor, atualiza a variável global com ele, e registra que os dados UV foram atualizados.

```
@tornado.gen.coroutine
def main():
    LOGGER.info("Creating HTTP server on: {}".format(HTTP_PORT))
    http_server = HTTPServer(port=HTTP_PORT)
    LOGGER.info("Creating servient")
    servient = Servient()
    servient.add_server(http_server)
    LOGGER.info("Starting servient")
    wot = yield servient.start()
    LOGGER.info("Exposing and configuring Thing")
    exposed_thing = wot.produce(json.dumps(DESCRIPTION))
    exposed_thing.set_property_read_handler(NAME_PROP_UV, uv_read_handler)
    exposed_thing.set_property_write_handler(NAME_PROP_UV, uv_write_handler)
    exposed_thing.expose()
```

A função main é o ponto de entrada da aplicação. Ela é responsável por ini-

cializar e configurar o servidor e a "Coisa"WoT. Como a função main usa a função yield, ela precisa ser uma coroutine, e é por isso que ela é definida com o decorador @tornado.gen.coroutine.

Vamos analisar cada parte da função:

LOGGER.info("Creating HTTP server on: ".format(HTTP_PORT)): Isso registra uma mensagem indicando que a criação do servidor HTTP está prestes a começar.

http_server = HTTPServer(port=HTTP_PORT): Esta linha cria uma nova instância do servidor HTTP no porto especificado pela constante HTTP_PORT.

LOGGER.info("Creating servient"): Isso registra uma mensagem indicando que a criação do Servient (uma entidade que pode consumir e expor "Coisas"WoT) está prestes a começar.

servient = Servient(): Esta linha cria uma nova instância de um Servient.

servient.add_server(http_server): Aqui, o servidor HTTP é adicionado ao Servient, permitindo que ele exponha e consuma "Coisas"WoT via HTTP.

LOGGER.info("Starting servient"): Isso registra uma mensagem indicando que o Servient está prestes a ser iniciado.

wot = yield servient.start(): Esta linha inicia o Servient. Como a função start é assíncrona (retorna uma Future), a palavra-chave yield é usada para pausar a execução da função main até que o Servient seja iniciado. O valor retornado (um objeto WoT) é então atribuído à variável wot.

LOGGER.info("Exposing and configuring Thing"): Isso registra uma mensagem indicando que a exposição e configuração da "Coisa"WoT estão prestes a começar.

exposed_thing = wot.produce(json.dumps(DESCRIPTION)): Esta linha cria a "Coisa"WoT usando a descrição definida anteriormente e a expõe.

exposed_thing.set_property_read_handler(NAME_PROP_UV, uv_read_handler): Aqui, o manipulador de leitura para a propriedade UV é configurado para ser a função uv_read_handler.

exposed_thing.set_property_write_handler(NAME_PROP_UV, uv_write_handler): Aqui, o manipulador de gravação para a propriedade UV é configurado para ser a função uv_write_handler.

exposed_thing.expose(): Finalmente, a "Coisa"WoT é exposta, tornando-a acessível para outros dispositivos na rede WoT.

Então, em resumo, a função main é responsável por configurar e iniciar o servidor e a "Coisa"WoT.

```
if __name__ == "__main__":
    LOGGER.info("Starting loop")
```

```
IOLoop.current().add_callback(main)
IOLoop.current().start()
```

Essa parte do código é responsável por iniciar o loop de eventos principal do Tornado e colocar a função main na fila para ser executada assim que o loop começar.

if __name__ == "__main__": é uma estrutura comum em scripts Python. __name__ é uma variável especial que o Python cria para cada módulo. Quando um módulo é importado, __name__ é definido como o nome do módulo. No entanto, quando o arquivo é executado como um script (em vez de ser importado), __name__ é definido como "__main__". Portanto, essa estrutura é usada para garantir que o código dentro do bloco seja executado apenas quando o script é executado diretamente, e não quando é importado como um módulo.

LOGGER.info("Starting loop") é uma chamada para registrar uma mensagem indicando que o loop de eventos está prestes a começar.

IOLoop.current().add_callback(main) usa o método add_callback da classe IOLoop para adicionar a função main à fila de tarefas a serem executadas. Essa função será chamada assim que o loop de eventos começar.

IOLoop.current().start() inicia o loop de eventos. Esta chamada bloqueia e só retorna quando o loop de eventos é interrompido (o que normalmente acontece quando o programa é encerrado). Quando o loop começa, ele começa a executar todas as funções na fila de tarefas, que neste caso inclui a função main.

Então, essencialmente, essas linhas estão inicializando o loop de eventos do Tornado e configurando-o para chamar a função main assim que começar.

2.12.2 main.py

No dispositivo ESP32, o valor de UV é lido periodicamente do sensor UV e enviado ao servidor WoT. Para isso, o script utiliza a biblioteca urequests para enviar uma requisição HTTP PUT ao servidor, incluindo o valor lido do sensor no corpo da requisição.

```
import machine
import ujson
import urequests
import time
```

machine: Esta biblioteca fornece funções para interagir com o hardware do dispositivo. Neste caso, é usada para interagir com o sensor UV através do ADC (Conversor Analógico-Digital).

ujson: É uma versão otimizada para microcontroladores da biblioteca json do Python padrão. É usada para serializar os dados do sensor em um formato JSON para serem enviados ao servidor.

urequests: Semelhante à biblioteca requests no Python padrão, urequests é uma biblioteca para fazer solicitações HTTP. Aqui, é utilizada para enviar os dados do sensor ao servidor WoT.

time: Esta biblioteca fornece funções para trabalhar com o tempo. Neste código, é usada para fazer o dispositivo ESP32 aguardar um certo período de tempo entre as leituras do sensor.

Então, essas quatro bibliotecas são usadas para realizar as principais tarefas no ESP32: ler os dados do sensor UV, converter esses dados para JSON, enviá-los para o servidor WoT, e fazer uma pausa entre as leituras do sensor.

```
TD = {
    "links": [
        {"href": "http://000.000.0.00:9494/urn:uvthing/property/uv"}
    ]
}
```

Este é um exemplo de um documento de Descrição de Coisa (TD - Thing Description) simplificado. Em uma aplicação WoT (Web of Things), a Descrição da Coisa é uma representação padronizada da interface de uma coisa na Web of Things, fornecendo detalhes sobre suas capacidades e como interagir com ela.

Neste caso específico, o documento TD contém apenas um único link, que é o endpoint do servidor onde o valor do sensor UV é enviado. Este link é definido como uma URL, que é composta da seguinte forma:

```
"href": "http://<host>:<port>/<thing_name>/property/<property_name>"
```

Portanto, o script do ESP32 utiliza este documento TD para saber para onde enviar os dados do sensor UV. Quando ele lê um valor do sensor, ele envia esse valor para a URL especificada no documento TD.

```
while True:
uv_value = adc.read()
data = {"uv": uv_value}
url = TD['links'][0]['href']
headers = {'content-type': 'application/json'}

try:
    r = urequests.put(url, data=ujson.dumps(data), headers=headers)
    if r.status_code >= 200 and r.status_code < 300:
        print('Successfully sent data to WoT server')
    else:
        print('Failed to send data to WoT server: received status code {}'.format(r.stat
    r.close()
except Exception as e:
    print('Could not send data to WoT server: ', e)

time.sleep(10)
```

Este bloco de código é o loop principal que o dispositivo ESP32 executa continuamente. Ele realiza as seguintes ações:

Lê o valor do sensor UV usando a função adc.read(). A variável adc é uma instância de machine.ADC, que foi inicializada para ler o valor do sensor UV.

Cria um dicionário data que contém o valor lido do sensor UV. Este dicionário é então serializado em uma string JSON usando ujson.dumps(data). O resultado é o corpo da solicitação HTTP que será enviada ao servidor WoT.

Recupera a URL do servidor WoT do documento de Descrição da Coisa (TD).

Define os cabeçalhos da solicitação HTTP para indicar que o conteúdo da solicitação é JSON.

Envia uma solicitação HTTP PUT para a URL do servidor WoT com os dados do sensor UV no corpo da solicitação. Se a solicitação for bem-sucedida (indicado por um código de status HTTP na faixa de 200 a 299), ele imprime uma mensagem de sucesso. Se a solicitação falhar (indicado por um código de status HTTP fora da faixa de 200 a 299), ele imprime uma mensagem de erro.

Se ocorrer um erro ao tentar enviar a solicitação (por exemplo, se o dispositivo não conseguir se conectar ao servidor), ele imprime uma mensagem de erro.

Por fim, o dispositivo pausa por 10 segundos antes de começar a próxima iteração do loop. Isto é para garantir que o dispositivo não sobrecarregue o servidor WoT com solicitações.

Em resumo, este loop principal está lendo continuamente o valor do sensor UV, enviando esse valor ao servidor WoT e pausando por 10 segundos entre cada leitura/envio. Se ocorrer um erro em qualquer parte deste processo, ele imprime uma mensagem de erro.

2.12.3 Refatoração

Principais mudanças feitas durante a refatoração:

server.py:

Funções menores: start_server foi introduzido. Esta função incorpora a criação e inicialização do servidor WoT e a configuração da "Coisa"que o servidor está expondo. Isso torna o código mais modular e mais fácil de entender.

Nomes de Variáveis: Alterei o nome da variável GLOBAL_UV_DATA para uv_data. Em Python, é uma convenção que nomes de variáveis em maiúsculas são constantes e não devem ser alteradas. Além disso, o nome uv_data é mais simples e claro.

Nomes de Funções: Alterei o nome das funções de manipulação de UV de uv_read_handler e uv_write_handler para read_uv e write_uv, respectivamente. Isso foi feito para aderir à convenção de que os nomes das funções devem ser verbos e descrever a ação que a função realiza.

main.py

Funções menores: Introduzi a função send_uv_data, que encapsula a lógica de envio de dados para o servidor WoT. Esta função é então chamada dentro do loop principal do programa.

Função Main: Introduzi uma função main que contém o loop principal do programa. Isso torna o código mais organizado e permite que a lógica principal do programa seja facilmente identificada.

2.12.4 Aprimorar o Escalonamento

No código aprimorado para escalonamento com suporte a múltiplos sensores, foram feitas as seguintes alterações:

Uso de Dicionário para Sensores e Descrição de Coisas:

Os sensores agora são representados por um dicionário chamado sensors. Cada sensor é identificado por uma chave exclusiva (por exemplo, 'uv'), que também é usada como identificador no Thing Description correspondente. O dicionário sensors contém informações sobre cada sensor, como o tipo do sensor (por exemplo, 'uv') e a configuração específica do sensor (neste caso, a configuração do pino do ADC). O Thing Description é representado por um dicionário chamado TD. Cada sensor tem seu próprio conjunto de links e outros metadados. Essa abordagem permite adicionar facilmente mais sensores no futuro, especificando suas informações no dicionário sensors e no dicionário TD. Função send_sensor_data:

Introduzi uma função chamada send_sensor_data, que é responsável por enviar os dados de um sensor específico para o servidor WoT. A função recebe o sensor_id (a chave do sensor no dicionário sensors), o sensor_type, a URL do servidor e os dados a serem enviados. O sensor_id é usado para obter a URL específica do sensor no Thing Description (TD) e também é usado para imprimir mensagens de log mais informativas. Agora, você pode adicionar mais sensores ao dicionário sensors e eles serão tratados de forma genérica pela função send_sensor_data. Loop Principal:

O loop principal no main agora itera sobre os sensores no dicionário sensors. Para cada sensor, é lido o valor do sensor e criado um dicionário de dados correspondente. A função send_sensor_data é chamada para enviar os dados do sensor para o servidor WoT. Isso permite que você adicione mais sensores no dicionário sensors e eles serão processados de forma iterativa. Essas alterações permitem adicionar e dimensionar facilmente mais sensores no seu sistema, tornando-o mais flexível e expansível no futuro.

2.12.5 Recomendações W3C

Os códigos fornecidos não seguem completamente as recomendações da W3C (World Wide Web Consortium) para a criação de um servidor WoT. No entanto,

eles implementam alguns conceitos fundamentais do WoT, como a exposição de um Thing (coisa) com propriedades e a interação por meio do protocolo HTTP.

As recomendações da W3C para o WoT incluem padrões e especificações técnicas para descrever, conectar e interagir com as coisas na Web. Algumas das principais especificações do WoT são o WoT Thing Description (TD) e o WoT Binding Templates. Essas especificações fornecem uma estrutura para descrever as coisas, suas propriedades, ações e eventos, bem como definir como elas podem ser acessadas e interagidas.

Os códigos fornecidos estão parcialmente alinhados com as recomendações da W3C para o Web of Things (WoT), mas também apresentam algumas áreas em que não estão completamente alinhados. Vamos analisar esses pontos:

Alinhados com as recomendações da W3C:

Exposição do Thing: O código server.py implementa a exposição de um Thing (coisa) por meio da biblioteca WotPy. Ele define uma descrição básica do Thing e configura um servidor HTTP para receber solicitações.

Propriedades observáveis: A descrição do Thing no server.py inclui a definição de uma propriedade chamada "uv"que é observável. Isso está alinhado com as recomendações da W3C para permitir que os clientes recebam atualizações quando o valor da propriedade é alterado.

Handlers de leitura e escrita: O código server.py implementa handlers de leitura e escrita para a propriedade UV. Esses handlers são responsáveis por fornecer o valor atual da propriedade para leitura e atualizar o valor quando ocorre uma escrita. Isso é consistente com as recomendações da W3C para interações com as propriedades dos Things.

Não totalmente alinhados com as recomendações da W3C:

Descrição do Thing: A descrição do Thing no server.py é definida como um dicionário Python em vez de seguir o formato do WoT Thing Description (TD), que é uma especificação da W3C. O uso do dicionário é uma simplificação e não segue a estrutura e os detalhes específicos recomendados pelo WoT TD.

Protocolo HTTP: Embora o protocolo HTTP seja comumente usado em implementações WoT, as recomendações da W3C também incluem outros protocolos adequados para dispositivos IoT com recursos limitados, como o CoAP e o MQTT. O código fornecido utiliza apenas o protocolo HTTP e não considera outras opções.

Configuração de segurança: Os códigos fornecidos não abordam a configuração de segurança, como autenticação e criptografia, que são importantes na implementação de um servidor WoT seguro. As recomendações da W3C enfatizam a importância de considerar a segurança em todas as etapas da implementação.

2.12.6 Versão Final

https://github.com/agmangas/wot-py/commit/d1151f9e2c0ba9e176b18755d83ebf8c0b98014c

Transmissão de Dados do Sensor UV com ESP32 e WoT

Este projeto tem como objetivo demonstrar a transmissão de dados do sensor UV de um microcontrolador ESP32 para um servidor Web of Things (WoT) utilizando o protocolo HTTP. O projeto utiliza a biblioteca WotPy para criar o servidor e lidar com as interações do WoT, além do microcontrolador ESP32 com o sensor UV ML8511.

O código consiste em dois arquivos principais: server.py e main.py. O arquivo server.py configura o servidor WoT, expõe um Thing (coisa) que representa o sensor UV e define manipuladores personalizados para a leitura e escrita dos dados do sensor UV. Por outro lado, o arquivo main.py é executado no ESP32, lê os dados do sensor UV periodicamente e os envia para o servidor WoT usando requisições HTTP.

Para utilizar este projeto, você precisa executar o arquivo server.py em seu servidor Python e carregar o arquivo main.py no seu ESP32. O ESP32 lerá continuamente os dados do sensor UV e os enviará para o servidor, onde podem ser acessados e observados através do Thing exposto.

Este projeto serve como um exemplo básico de integração de dispositivos IoT com os princípios do WoT, possibilitando a comunicação e interoperabilidade entre dispositivos e aplicativos em um ecossistema IoT descentralizado.

Sinta-se à vontade para personalizar e expandir este projeto de acordo com seus requisitos e configurações específicas de sensores.

`server.py`

Importação das bibliotecas e módulos necessários:

```
import json
import logging
import tornado.gen
from tornado.ioloop import IOLoop
from wotpy.protocols.http.server import HTTPServer
from wotpy.wot.servient import Servient
```

Definição das constantes:

```
HTTP_PORT = 9494
ID_THING = "urn:esp32"
UV_SENSOR = "uv"
```

A constante HTTP_PORT especifica a porta na qual o servidor irá escutar. ID_THING representa o identificador para o Thing (coisa), e UV_SENSOR é o nome da propriedade que representa o sensor UV.

Armazenamento global dos dados:

```
uv_data = None
```

Esta variável irá armazenar os dados mais recentes do sensor UV recebidos do ESP32.

Configuração do registro (logging):

```
logging.basicConfig()
logger = logging.getLogger()
logger.setLevel(logging.INFO)
```

Configurando o registro para exibir mensagens de log com o nível definido como INFO.

Descrição do Thing:

```
description = {
    "id": ID_THING,
    "name": ID_THING,
    "properties": {
        UV_SENSOR: {
            "type": "number",
            "readOnly": False,
            "observable": True
        }
    }
}
```

Este dicionário representa a descrição do Thing. Ele especifica o ID, nome e propriedades do Thing. Neste caso, define a propriedade UV_SENSOR como um tipo numérico que pode ser lido e observado.

Manipuladores personalizados para leitura e escrita dos dados UV:

```
@tornado.gen.coroutine
def read_uv():
    """Manipulador personalizado para a propriedade 'UV'."""
    if uv_data is None:
        return
    uv_data_dict = json.loads(uv_data.decode("utf-8"))
    return float(uv_data_dict['uv'])

@tornado.gen.coroutine
def write_uv(value):
    """Manipulador personalizado para escrever dados UV."""
    global uv_data
    uv_data = value
```

Essas duas funções servem como manipuladores personalizados para ler e escrever os dados do sensor UV. A função read_uv analisa os dados do UV armazenados e os retorna como um valor float. A função write_uv atualiza a variável uv_data com o valor recebido.

Iniciando o servidor:

```
@tornado.gen.coroutine
def start_server():
    http_server = HTTPServer(port=HTTP_PORT)
    servient = Servient()
    servient.add_server(http_server)
    wot = yield servient.start()
    exposed_thing = wot.produce(json.dumps(description))
    exposed_thing.set_property_read_handler(UV_SENSOR, read_uv)
    exposed_thing.set_property_write_handler(UV_SENSOR, write_uv)
    exposed_thing.expose()

if __name__ == "__main__":
    IOLoop.current().add_callback(start_server)
    IOLoop.current().start()
```

A função start_server cria um servidor HTTP utilizando a porta especificada e inicializa o objeto Servient do WotPy. Em seguida, ele inicia o servient, produz o Thing com base na descrição, define os manipuladores personalizados de leitura e escrita para a propriedade UV e expõe o Thing. Por fim, o servidor é iniciado adicionando a função start_server ao IOLoop.

`main.py`

Importação dos módulos necessários:

```
import machine
import ujson
import urequests
import time
```

Configuração do sensor:

```
sensors = {
    'uv_sensor': {'type': 'uv', 'sensor': machine.ADC(machine.Pin(34))},
}
```

Este dicionário contém as configurações do sensor. Neste caso, ele define um sensor uv_sensor com seu tipo como 'uv' e especifica o pino correspondente no ESP32.

Descrição do Thing:

```
TD = {
    'uv_sensor': {"links": [{"href": "http://192.168.0.39:9494/urn:esp32/property/uv"}]},
```

```
}
```

O dicionário TD armazena as descrições dos Things. Ele contém um link para a propriedade do sensor UV no servidor.

Função para enviar os dados do sensor para o servidor:

```
def send_sensor_data(sensor_id, sensor_type, url, data):
    headers = {'content-type': 'application/json'}
    try:
        r = urequests.put(url.format(sensor_id), data=ujson.dumps(data), headers=headers)
        if r.status_code >= 200 and r.status_code < 300:
            print('Dados do {} enviados com sucesso do {} para o servidor WoT'.format(sensor
        else:
            print('Falha ao enviar os dados do {} do {} para o servidor WoT: código de statu
        r.close()
    except Exception as e:
        print('Não foi possível enviar os dados do {} do {} para o servidor WoT: '.format(se
```

Esta função recebe o ID do sensor, o tipo, a URL e os dados como entrada. Ela envia uma requisição HTTP PUT para o servidor com a carga útil de dados no formato JSON. Em seguida, ela imprime uma mensagem de sucesso ou falha com base no código de status da resposta.

Função principal:

```
def main():
    while True:
        for sensor_id, sensor_info in sensors.items():
            sensor_value = sensor_info['sensor'].read()
            data = {sensor_info['type']: sensor_value}
            url = TD[sensor_id]['links'][0]['href']
            send_sensor_data(sensor_id, sensor_info['type'], url, data)
        time.sleep(5)

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

A função main é executada quando o script é executado. Ela entra em um loop infinito e itera por cada sensor definido no dicionário sensors. Ela lê o valor do sensor, cria um objeto de dados com o tipo correspondente, recupera a URL da Descrição do Thing (TD) e chama a função send_sensor_data para enviar os dados para o servidor. O loop aguarda 5 segundos antes de repetir.

`boot.py`

O arquivo boot.py é um script que é executado automaticamente quando o ESP32 é inicializado. Seu objetivo é estabelecer uma conexão Wi-Fi com a rede especificada, permitindo que o ESP32 se conecte à internet e se comunique com o servidor WoT.

O código define uma função chamada do_connect que recebe o SSID (nome da rede) e a senha como argumentos. Dentro da função, ele importa o módulo network para gerenciar a conexão Wi-Fi.

A função verifica se o ESP32 já está conectado a uma rede usando o método isconnected da interface STA_IF (interface de estação). Se o ESP32 não estiver conectado, ele prossegue com a conexão, ativando a interface de estação (active(True)) e chamando o método connect com o SSID e senha fornecidos.

Após iniciar a conexão, o código entra em um loop que espera até que o ESP32 se conecte com sucesso à rede (isconnected() retorna True).

Uma vez que a conexão seja estabelecida, ele imprime os detalhes de configuração da rede (endereço IP, máscara de sub-rede, gateway, etc.) usando o método ifconfig da interface de estação.

Para usar este script, substitua 'YourSSID' pelo SSID da sua rede Wi-Fi e 'YourPassword' pela senha correspondente.

Ao incluir este script boot.py no seu ESP32, ele se conectará automaticamente à rede Wi-Fi especificada durante a inicialização, garantindo uma conexão com a internet confiável para o seu aplicativo.

Lembre-se de carregar o arquivo boot.py modificado para o ESP32 e verificar se as credenciais de rede estão corretas antes de implantar o seu projeto.