Introdução à Internet das Coisas

Protocolos de Comunicação de Dados

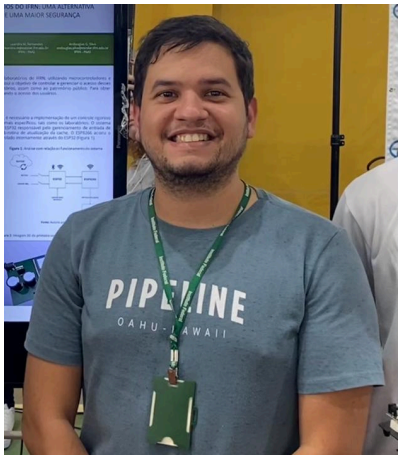
Apostila do aluno

Ph.D. Andouglas Gonçalves da Silva Júnior
Ph.D. Manoel do Bonfim Lins de Aquino
Marcos Fábio Carneiro e Silva
Autor da apostila

Ph.D. Andouglas Gonçalves da Silva Júnior
Ph.D. Manoel do Bonfim Lins de Aquino
Instrutor do curso

Larissa Jéssica Alves – Analista de Suporte Pedagógico
Revisão da apostila

Autor



Andouglas Gonçalves da Silva Júnior

Doutor em Engenharia Elétrica e da Computação - UFRN. Mestre em Engenharia Mecatrônica na área de Sistemas Mecatrônicos. Bacharel em Ciências e Tecnologia pela EC&T - Escola de Ciências e Tecnologia - UFRN. Engenheiro Mecatrônico - UFRN. Professor de Ensino Básico, Técnico e Tecnológico no Instituto Federal de Educação Tecnológica do Rio Grande do Norte (IFRN). Integrante da Rede de Laboratórios NatalNet, LAICA e colaborador ISASI-CNR-Itália. Desenvolve projetos na área de Machine Learning, Internet das Coisas e Holografia Digital. Colabora no projeto do N-Boat (Veleiro Robótico Autônomo), principalmente no desenvolvimento de sistemas para monitoramento da qualidade da água e identificação de micropartículas usando holografia digital e IA.

Manoel do Bonfim Lins de Aquino

Possui graduação (2006), mestrado (2008) e doutorado (2022) em Engenharia Elétrica e da Computação, pela Universidade Federal do Rio Grande do Norte - UFRN. Tem experiência na área de projetos de Telecomunicações, atuando na Siemens como engenheiro de Telecomunicações (2008-2010). Sou Professor (2010 - atual) do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte - IFRN, membro do NADIC, Núcleo de Análise de Dados e Inteligência Computacional, onde venho desenvolvendo projetos de P&D nas áreas de desenvolvimento de sistemas, IoT e Inteligência Artificial.

Marcos Fábio Carneiro e Silva

Estudante de informática no Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte. Participou, como bolsista, de projeto de pesquisa voltado à automação em ambiente escolar com IoT (2022 - 2023), Possui experiência em redes de computadores, eletrônica e IoT, com ênfase na utilização de microcontroladores e desenvolvimento de Hardware. Além disso, possui conhecimentos básicos em Python e C++. Atualmente é membro do NADIC (Núcleo de Análise de Dados e Inteligência Computacional) do IFRN.

APRESENTAÇÃO

Bem-vindo ao curso de **Introdução à Internet das Coisas** do *CEPEDI!*

A Internet das Coisas, ou IoT, é uma revolução tecnológica que está transformando a maneira como interagimos com o mundo ao nosso redor. Essa inovadora e crescente rede de dispositivos interconectados, que variam desde sensores e aparelhos domésticos até veículos e equipamentos industriais, está desencadeando uma mudança fundamental em como coletamos, compartilhamos e utilizamos informações.

Neste curso, introduziremos o fascinante mundo da IoT, definindo conceitos básicos, seus principais componentes e aplicações. Além disso, introduziremos os protocolos de comunicação mais usados em aplicações de Internet das Coisas, além dos principais dispositivos utilizados hoje, como ESP32, Arduino, Raspberry Pi Pico, entre outros.

Além disso, pretendemos oferecer um curso que mescle a teoria com a prática. Para isso, utilizaremos aplicações livres como Wokwi e Bipes para desenvolvimento de projetos que nos auxiliarão no aprendizado dos conceitos teóricos.

Recomendamos ao aluno que, ao final da leitura de cada seção, realize os exercícios propostos e acesse os materiais indicados nas referências bibliográficas, para aprofundar a leitura desse material e complementar o que foi lido aqui.

Desejo a você, prezado aluno, que tenha um excelente curso!!

Boa Leitura !!

Sumário

1 Protocolos de Comunicação de Dados em IoT

A comunicação de dados desempenha um papel fundamental na Internet das Coisas (IoT), conectando dispositivos e permitindo a troca de informações. Diferentes protocolos são usados para facilitar essa comunicação, cada um com suas características específicas. Nesta seção, serão abordados alguns dos principais protocolos de comunicação de dados em IoT, destacando suas arquiteturas e funcionalidades.

1.1 Arquitetura de Rede em IoT

A arquitetura de rede refere-se à estrutura organizacional e ao design de uma rede de computadores, delineando como os dispositivos e sistemas interagem e se comunicam entre si. Essa arquitetura estabelece diretrizes para a organização dos componentes de rede, incluindo hardware, software, protocolos e tecnologias, a fim de fornecer conectividade eficiente, segura e escalável.

Existem diferentes tipos de arquiteturas de rede, e a escolha depende dos requisitos específicos da aplicação. Vale destacar que cada arquitetura é empregada em diferentes tipos de protocolos de comunicação, e no caso do IoT, temos duas arquiteturas predominantes: cliente-servidor e publish/subscribe.

1.1.1 Cliente-Servidor

Na arquitetura cliente-servidor, um dispositivo cliente se conecta a um servidor para enviar ou receber dados. O servidor atua como um ponto final para os dispositivos clientes, fornecendo uma única central de controle e gerenciamento.

Na Figura 5.1 temos a representação da arquitetura cliente-servidor, onde o fluxo de dados segue uma lógica em que os clientes enviam requisições aos servidores, e estes respondem com os resultados ou recursos solicitados. No contexto do IoT, o servidor também desempenha a função de armazenar e processar os dados enviados pelos clientes. Nesse sentido, informações emitidas por sensores, por exemplo, são armazenadas em determinado servidor, que poderia ser acessado por uma aplicação – perceba que em todo caso, o servidor é sempre o destino final do processo.

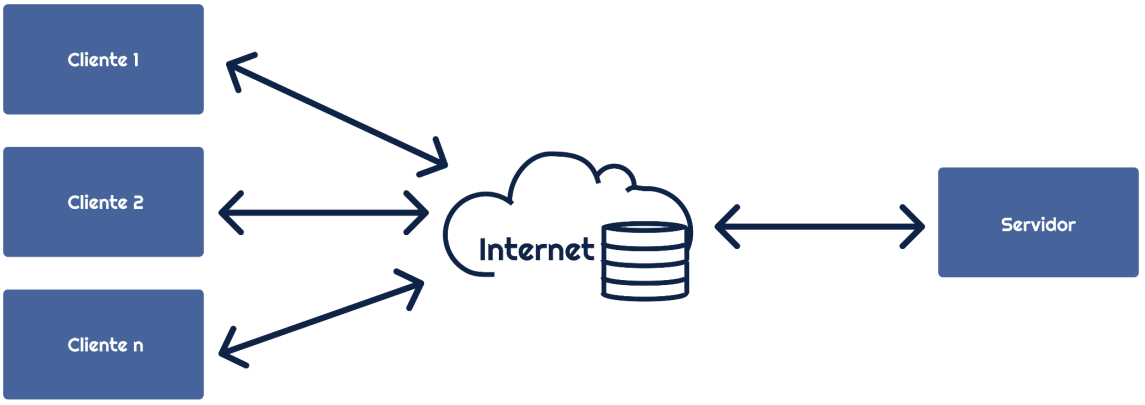


Fig. 5.1 - Arquitetura Cliente-Servidor

O modelo cliente-servidor é uma boa escolha para aplicações IoT onde um dispositivo IoT precisa se comunicar com um servidor central, como em aplicações de monitoramento ou controle. O modelo cliente-servidor também é uma boa escolha quando os dados precisam ser armazenados e processados em um servidor central.

1.1.2 Publish/Subscribe

No modelo publish/subscribe, um dispositivo publica (publicador) dados em um tópico, e outros dispositivos que estão interessados neste tópico (assinantes) podem se inscrever para receber esses dados. Os dispositivos que publicam dados não precisam saber quem está se inscrevendo para receber.

Podemos fazer uma analogia com emissoras de TV. Elas desempenham o papel de publicadoras ao disponibilizar seu canal em TV aberta; os telespectadores, por sua vez, escolhem qual canal vão assistir, desempenhando o papel de inscrito. Note que qualquer pessoa pode assistir a qualquer canal, bastando apenas sintonizar seu dispositivo no canal desejado (tópico).

Na Figura 5.2 temos a representação da arquitetura publish/subscribe, nela o servidor (broker) atua como intermediário entre os publicadores e inscritos. Os dados são enviados ao broker por meio de tópicos definidos. Os inscritos, por sua vez, têm acesso aos dados contidos no broker a partir deste tópico. Ou seja, podemos entender o broker como uma grande pasta, onde os tópicos são os endereços das subpastas que guardam os dados de maneira organizada.

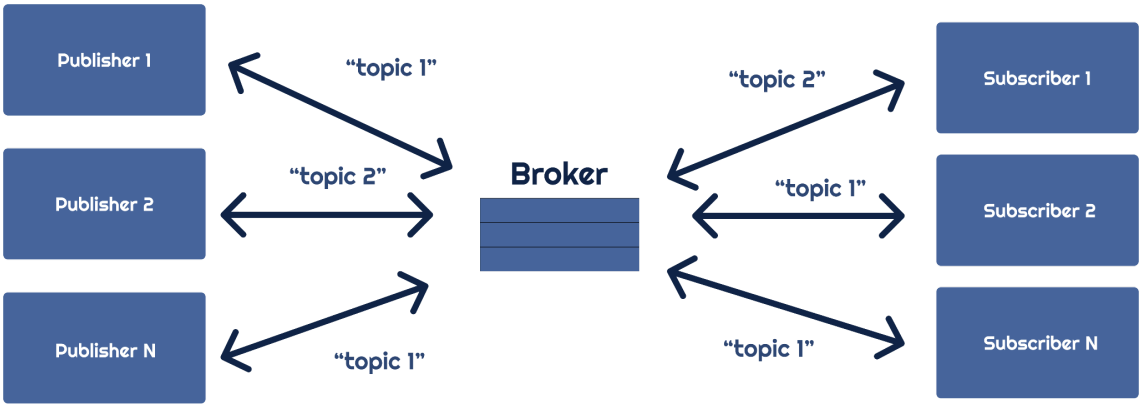


Fig. 5.2 - arquitetura publish/subscribe.

O modelo publish/subscribe é uma boa escolha para aplicações IoT onde os dispositivos IoT precisam compartilhar dados entre si, como em aplicações de sensoriamento ambiental ou de rastreamento de ativos. O modelo publish/subscribe também é uma boa escolha para aplicações IoT onde a escalabilidade é importante.

A Tabela 1 mostra um comparativo entre as duas arquiteturas.

Tabela 1 - Quadro comparativo entre as duas arquiteturas.

Fator	Modelo cliente-servidor	Modelo publish/subscribe
Relação entre dispositivos	Um dispositivo cliente se conecta a um servidor	Dois ou mais dispositivos podem se comunicar entre si
Controle e gerenciamento	O servidor fornece um ponto central de controle e gerenciamento	Não há controle ou gerenciamento central
Armazenamento e processamento de dados	Os dados podem ser armazenados e processados no servidor	Os dados podem ser armazenados e processados em qualquer dispositivo
Eficiência em termos de recursos	Pode ser mais eficiente em termos de recursos do que o modelo publish/subscribe	Pode ser mais eficiente em termos de largura de banda do que o modelo cliente-servidor
Complexidade	Pode ser mais complexo de implementar do que o modelo publish/subscribe	Relativamente simples de implementar e usar
Escalabilidade	Pode não ser escalável para aplicações IoT com um grande número de dispositivos	Escalável para aplicações IoT com um grande número de dispositivos

Nesse sentido, a escolha da arquitetura correta depende de uma série de fatores, como o tipo de aplicação IoT, os requisitos de desempenho e os recursos de rede disponíveis.

1.2 Sockets

Um conceito muito usado para estabelecer a conexão entre os dispositivos em uma comunicação é o *socket*, que atua como um ponto final para a comunicação bidirecional entre dois programas rodando em uma rede. Um socket é vinculado a um número de porta específico para que a camada de transporte possa identificar o aplicativo ao qual os dados devem ser entregues. Existem dois tipos principais de sockets utilizados em redes: o *socket* de fluxo e o de datagrama.

Os *sockets* de fluxo (Stream Sockets) utilizam o protocolo TCP (Transmission Control Protocol) para a comunicação. Eles fornecem uma comunicação confiável, orientada à conexão e baseada em byte. Isso significa que os dados chegam na ordem em que foram enviados e sem duplicatas. Os sockets de fluxo são usados em aplicações onde é crucial que os dados sejam recebidos na mesma ordem em que foram enviados, como em transferências de arquivos, servidores web e clientes de e-mail.

Já os sockets de datagrama (Datagram Sockets) utilizam o protocolo UDP (User Datagram Protocol). Eles fornecem uma comunicação não confiável, sem conexão e baseada em mensagens. Não há garantia de que as mensagens cheguem na ordem enviada ou mesmo que cheguem. Sockets de datagrama são usados em aplicações onde a velocidade é mais crítica do que a confiabilidade, como em jogos online ou streaming de vídeo.

Quando um programa deseja se comunicar com outro programa em uma rede, ele abre um socket. Esse socket pode "ouvir" as conexões de entrada se for um servidor, ou pode "conectar-se" a um endereço e porta específicos se for um cliente. Uma vez estabelecida a conexão, os dados podem ser enviados de ida e volta até que a conexão seja encerrada.

1.3 Protocolos de Comunicação de Dados

Em um ecossistema tão diverso como o da IoT, a padronização da comunicação é essencial. Diferentes dispositivos podem "falar" diferentes "idiomas". Nesta seção, apresentaremos alguns protocolos de comunicação fundamentais nesse tipo de aplicação.

1.3.1 HTTP/HTTPS

O Hypertext Transfer Protocol (HTTP) e sua contraparte segura, o Hypertext Transfer Protocol Secure (HTTPS), desempenham papéis cruciais na comunicação na internet, fornecendo a estrutura para transferência de dados entre clientes e servidores. Compreender esses protocolos é essencial para explorar a arquitetura da World Wide Web e garantir a segurança das informações transmitidas.

O HTTP opera na camada de aplicação da arquitetura TCP/IP, possibilitando a transferência de hipertexto, que inclui HTML, imagens, CSS, JavaScript e outros tipos de dados. Funciona a partir da arquitetura cliente-servidor, onde um cliente emite uma solicitação a um servidor, que, por sua vez, responde à solicitação. As mensagens de solicitação e resposta incluem informações como método HTTP, caminho do recurso solicitado, parâmetros e código de status HTTP, cabeçalho e corpo da resposta.

Funcionamento

A dinâmica padrão de funcionamento do HTTP é baseado em requisição e respostas. Para que uma informação seja enviada pelo servidor (response), um cliente precisar solicitar (request). No protocolo, existem alguns tipos de mensagens que indicam a finalidade daquela requisição. Por exemplo, o método ***GET*** é usado quando se deseja recuperar algum dado do servidor. Já o método ***POST*** envia dados para serem processados para um recurso especificado. Outros métodos mais comuns são: ***PUT, DELETE, HEAD, CONNECT,*** entre outros.

Para que os dados sejam enviados, o HTTP conta com uma arquitetura definida pela estrutura da requisição, A Figura 5.3 mostra um exemplo dessa estrutura.

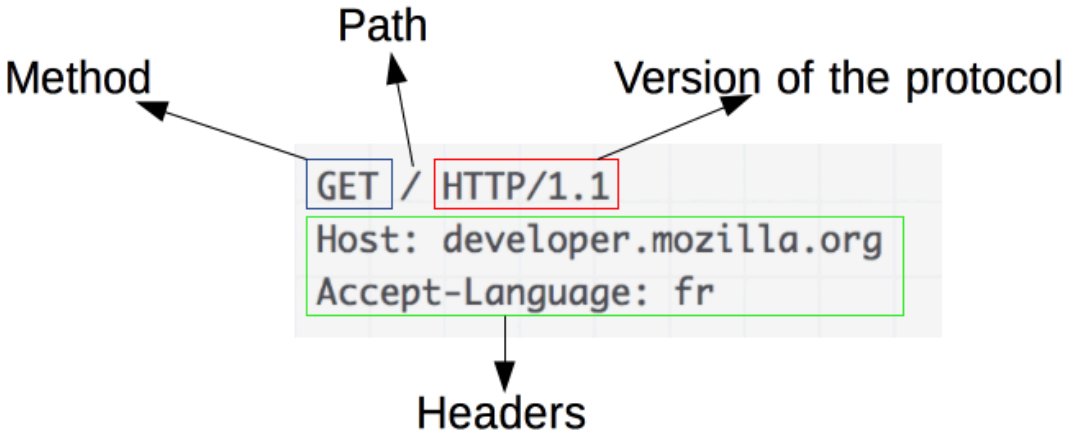


Fig. 5.3 - Exemplo de requisição HTTP. Fonte: https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview

Neste exemplo, o método da requisição é o GET, em outras palavras, está sendo solicitado alguma informação ao servidor no caminho (*path*) raiz "/". Isso quer dizer que se o seu servidor está rodando na sua máquina, de forma local, ele estará acessando a URL: "http://localhost:80", onde 80 é a porta padrão TCP e pode ser omitida. Logo em seguida é identificada a versão do protocolo, no caso HTTP/1.1. Na parte de baixo, temos o cabeçalho da solicitação (*headers*) onde estão contidas informações adicionais ou um corpo de dados, como no caso do POST, que envia dados ao servidor.

Já um exemplo de uma resposta fornecida por um servidor é visto na Figura 5.4. Essa resposta inicia com a versão do protocolo, que no caso do exemplo é o HTTP/1.1. Em seguida, o código de status (*status code*) que representa um número que identifica o estado da resposta, como, por exemplo, se foi bem-sucedida (200) , se a mensagem foi redirecionada (300), se houve um erro no lado do cliente (400) ou se um erro aconteceu do lado do servidor (500). Seguindo o exemplo de resposta, temos então uma mensagem de status, que corresponde a uma descrição informal sobre o *status code* e, por fim, os cabeçalhos (*headres)* , como os da requisição. Ainda é possível obter um corpo de dados do recurso que foi requisitado.

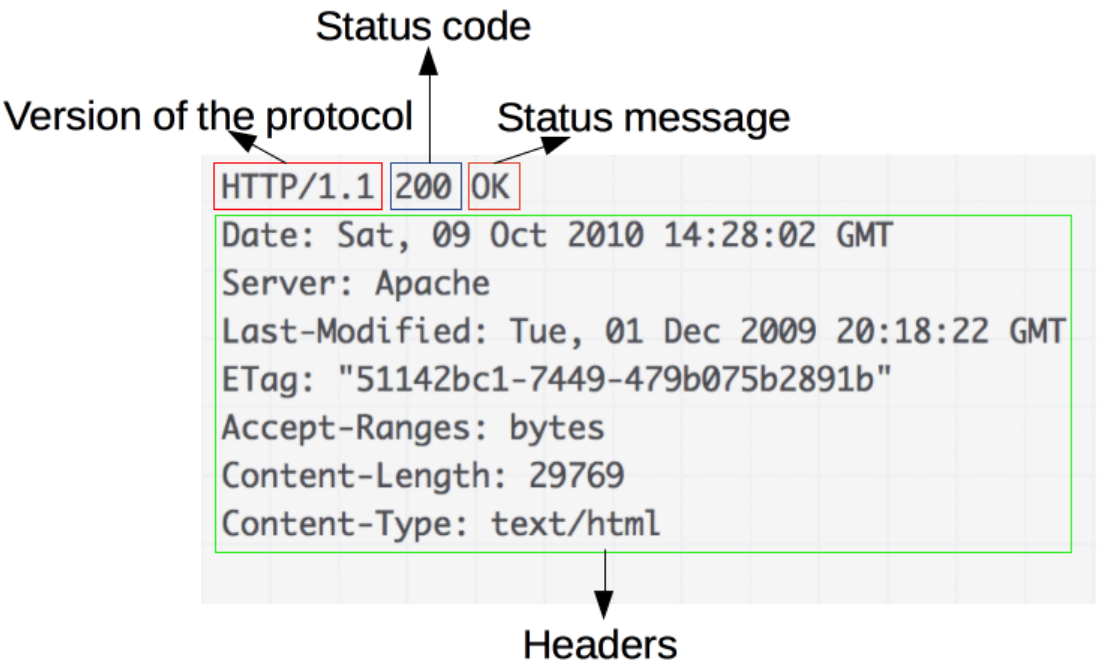


Fig. 5.3 - Exemplo de resposta HTTP. Fonte: https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview

O HTTPS, uma evolução do HTTP, introduz uma camada adicional de segurança. Utilizando o protocolo SSL/TLS, o HTTPS criptografa a comunicação entre o cliente e o servidor, protegendo os dados contra interceptação e modificações indesejadas. Essa segurança é especialmente crucial em aplicações web e em outras formas de comunicação, como transferência de arquivos e interações entre dispositivos IoT.

Ambos apresentam vantagens e desvantagens. O HTTP é bem conhecido, fácil de implementar e eficiente para pequenas quantidades de dados, mas carece da segurança robusta fornecida pelo HTTPS. Este último, por sua vez, oferece uma camada adicional de proteção, mas requer certificados SSL/TLS, aumentando os custos e podendo ser menos eficiente para grandes volumes de dados.

Exemplo

Vamos estudar dois exemplos que utilizam o protocolo HTTP em uma solução IoT. No primeiro, criaremos um método *get* para receber dados de uma página web. O segundo consiste em fazer de um ESP32 um simples servidor HTTP que podemos acessar através de um navegador externo.

Exemplo 5.1 - Método GET (Micropython)

```
def http_get(url):
    import socket
    _, _, host, path = url.split('/', 3)
    addr = socket.getaddrinfo(host, 80)[0][-1]
    s = socket.socket()
    s.connect(addr)
    s.send(bytes('GET /%s HTTP/1.0\r\nHost: %s\r\n\r\n' %
(path, host), 'utf8'))
    while True:
        data = s.recv(100)
        if data:
            print(str(data, 'utf8'), end='')
        else:
            break
    s.close()
```

Desta forma, podemos usar o método *http_get* e verificar se estamos recebendo alguma informação via uma requisição HTTP. Para isso, vamos usar o link **http://micropython.org/ks/test.html.** Rode o comando abaixo:

```
>>> http_get('http://micropython.org/ks/test.html')
```

A resposta para esse comando é apresentada a seguir. Tente identificar a versão, o *status code,* a descrição do *status,* o cabeçalho e os dados.

```
HTTP/1.1 200 OK
Server: nginx/1.10.3
Date: Mon, 05 Feb 2024 19:58:50 GMT
Content-Type: text/html
Content-Length: 180
Last-Modified: Tue, 03 Dec 2013 00:16:26 GMT
Connection: close
Vary: Accept-Encoding
ETag: "529d22da-b4"
Strict-Transport-Security: max-age=15768000
Accept-Ranges: bytes

<!DOCTYPE html>
<html lang="en">
```

```
    <head>
        <title>Test</title>
    </head>
    <body>
        <h1>Test</h1>
        It's working if you can read this!
    </body>
</html>
```

Agora, vamos para o segundo exemplo. O ESP32 será um servidor HTTP que, ao ser acessado, responderá com o valor (alto ou baixo) dos pinos 0, 2, 4, 5, 12, 13, 14 e 15. Perceba que a variável *response* recebe toda a estrutura do html que será enviado na requisição. Além disso, no momento de enviar os dados para aplicação cliente, as informações do protocolo e cabeçalho são enviadas antes.

Exemplo 5.2 - Servidor HTTP Simples (Micropython)

```
import machine
pins = [machine.Pin(i, machine.Pin.IN) for i in (0, 2, 4, 5, 12, 13,
14, 15)]

html = """<!DOCTYPE html>
<html>
    <head> <title>ESP8266 Pins</title> </head>
    <body> <h1>ESP8266 Pins</h1>
        <table border="1"> <tr><th>Pin</th><th>Value</th></tr> %s
</table>
    </body>
</html>
"""

import socket
addr = socket.getaddrinfo('0.0.0.0', 80)[0][-1]

s = socket.socket()
s.bind(addr)
s.listen(1)

print('listening on', addr)

while True:
    cl, addr = s.accept()
    print('client connected from', addr)
    cl_file = cl.makefile('rwb', 0)
    while True:
        line = cl_file.readline()
```

```
        if not line or line == b'\r\n':
            break
    rows = ['<tr><td>%s</td><td>%d</td></tr>' % (str(p), p.value())
for p in pins]
    response = html % '\n'.join(rows)
    cl.send('HTTP/1.0 200 OK\r\nContent-type: text/html\r\n\r\n')
    cl.send(response)
    cl.close()
```

1.3.2 AMQP

O Advanced Message Queuing Protocol - AMQP - ou Protocolo Avançado de Filas de Mensagens, representa um importante protocolo na arquitetura de comunicação em sistemas distribuídos, com destaque especial em ambientes de Internet das Coisas (IoT). Sua estrutura direcionada às mensagens oferece um conjunto robusto de recursos, fornecendo uma comunicação confiável e eficiente entre uma gama de dispositivos, gateways, servidores e aplicações.

Imaginemos um dispositivo de sensor IoT que envia dados de temperatura a um servidor de monitoramento. Utilizando o AMQP, o dispositivo encaminha essas mensagens para uma fila. O servidor, por sua vez, se inscreve nessa fila para receber as mensagens. Assim que uma mensagem é recebida, o servidor processa os dados e os armazenamentos, podendo também enviar uma mensagem de resposta ao dispositivo para indicar o sucesso na recepção dos dados.

Vamos entender alguns conceitos básicos utilizados no AMQP para depois compreender o funcionamento do protocolo.

Funcionamento

O AMQP utiliza a ideia da arquitetura *publisher/subscriber* apresentada anteriormente neste capítulo. Desta forma, o *broker* é usado como uma entidade que recebe mensagens dos *publishers* e as encaminham aos *consumers,* que são clientes que recebem essas mensagens. Desta forma, o AMQP é um protocolo bi-direcional onde um determinado cliente pode tanto enviar como receber informações por meio do broker.

Podemos pensar na estrutura do ***broker*** divididos em duas entidades: o ***exchange*** e a fila (***Queue***). O *exchange* recebe as mensagens de um publicador e, baseado em regras pré-programadas chamadas de ***bindings***, as

encaminham para as filas. As filas, por sua vez, podem estar sendo consumidas por outro cliente (*consumer*), baseado no seu tipo e característica.

As mensagens que são publicadas no *broker* e as mensagens que são consumidas possuem um parâmetro chamado *routing key,* que no contexto de filas ligadas a um *exchange,* são chamados de *binding key.* Basicamente, esse parâmetro é um identificador do caminho da mensagem e serve de base para armazenar corretamente as mensagens em filas.

Basicamente, esses são os conceitos básicos para entender como funciona o protocolo AMQP.

Na Figura 5.4 mostra um diagrama de funcionamento do protocolo AMQP utilizando os conceitos definidos acima.

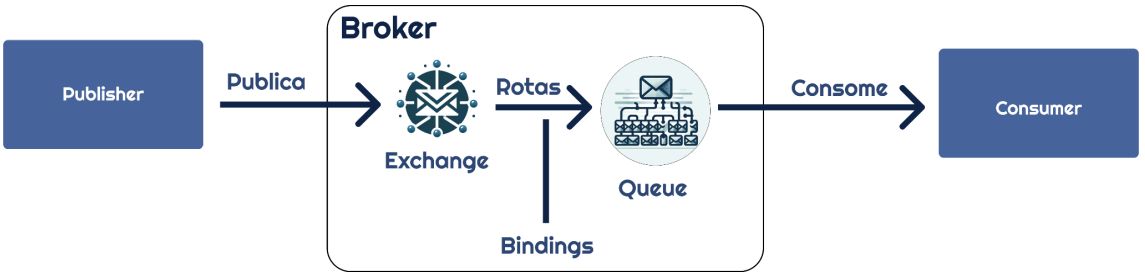


Fig. 5.4 - Diagrama de funcionamento do protocolo AMQP. Autoria Própria.

Uma característica importante que vale a pena destacar é que um cliente nunca publica uma mensagem diretamente em uma fila. Sempre será usado a entidade *exchange* que, baseado no seu tipo e nas configurações de *bindings,* encaminham as mensagens para uma determinada fila.

Vamos falar um pouco mais sobre a entidade *exchange*, já que ela representa um conceito extremamente importante dentro do protocolo.

Entidade Exchange

Vimos anteriormente que essa entidade é responsável por receber as mensagens dos clientes. É importante destacar que estes clientes não conhecem a existência das filas de forma que é o *exchange* que roteia/encaminha essas mensagens para uma ou mais filas, seguindo as regras denominadas *bindings.*

Dentro do contexto do protocolo AMQP existem vários tipos de *exchanges* que determinam como as mensagens são roteadas para as filas.

Cada tipo de *exchange* utiliza um método diferente para rotear as mensagens, baseado em regras definidas pelas *bindings* (associações) entre as *exchanges* e as filas.

Os principais tipos são:

- **Direct Exchange** - As mensagens são roteadas para as filas cujo *binding key* corresponde exatamente ao *routing key* da mensagem. Consiste em um método de roteamento ponto a ponto e muito útil quando você quer que a mensagem seja entregue a uma fila específica, baseando-se em um critério preciso.
- **Fanout Exchange** - As mensagens são roteadas para todas as filas que estão ligadas a esta *exchange*, ignorando o *routing key*. Em outras palavras, é um método de roteamento do tipo "*broadcast*". Este tipo é ideal para a distribuição de mensagens para múltiplos consumidores de forma igual, como em cenários de publicação de notificações.
- **Topic Exchange -** As mensagens são roteadas para uma ou mais filas baseadas em um padrão de correspondência entre o *routing key* da mensagem e o *pattern* (padrão) do *binding key*. Esse *pattern* pode incluir caracteres especiais, como "*" (substitui exatamente uma palavra) e "#" (substitui zero ou mais palavras). Este tipo é muito flexível e capaz de implementar diversos cenários de roteamento, como roteamento baseado em múltiplos critérios ou hierarquias.
- **Headers Exchange -** As mensagens são roteadas para as filas baseadas em cabeçalhos da mensagem (*headers*) ao invés do *routing key.* Para isso, uma mensagem é considerada correspondente se todos os *headers* correspondem aos que foram definidos nas *bindings*. Útil em cenários onde a rotação deve ser baseada em múltiplos atributos da mensagem e o *routing key* não é suficiente.

Cada tipo de *exchange* oferece uma abordagem diferente para o roteamento de mensagens, permitindo que os desenvolvedores escolham a melhor estratégia para a sua aplicação específica. A escolha do tipo de *exchange* afeta diretamente a lógica de como as mensagens são distribuídas e consumidas dentro de um sistema baseado em mensageria AMQP.

Outras Características do AMQP

O AMQP opera na camada de aplicação, acima da camada de transporte. Sua arquitetura *publish/subscribe* permite que dispositivos enviem mensagens para tópicos específicos, enquanto outros dispositivos podem se inscrever para receber essas mensagens, estabelecendo uma comunicação dinâmica e flexível.

No quesito confiabilidade, o AMQP tem a capacidade de garantir a entrega de mensagens. Isso inclui confirmação de entrega e reenvio de mensagens perdidas, garantindo que a informação crítica seja transmitida de maneira confiável.

Já em termos de eficiência de recursos, o AMQP é otimizado para ambientes IoT nos quais os dispositivos operam frequentemente com recursos limitados. Sua abordagem eficiente é crucial para manter a funcionalidade em dispositivos com restrições de energia.

Finalmente, sua flexibilidade permite adaptações para uma variedade de aplicações, garantindo sua relevância em cenários diversos de IoT.

Exemplo

Um dos softwares mais conhecidos que implementam o AMQP 0-9-1 é o **RabbitMQ**, um broker de mensagens de código aberto e amplamente utilizado. Usaremos ele para configurar um ***broker*** e executar uma aplicação básica em python. É importante destacar que o AMQP é um protocolo complexo, de forma que implementações, tanto em C++ como em Python, são raras. Geralmente, para conectar um ESP ou Arduino a um broker AMQP é implementado um gateway que converte mensagens transferidas no protocolo MQTT (que veremos mais adiante) em mensagens do protocolo AMQP. Desta forma, o exemplo que será demonstrado aqui é simples, usando um computador pessoal comum.

Usar o RabbitMQ envolve várias etapas, desde a instalação e configuração do servidor até a produção e consumo de mensagens dentro de suas aplicações. RabbitMQ é um broker de mensagens open-source que implementa o protocolo AMQP 0-9-1, oferecendo um mecanismo robusto para

sistemas de mensageria entre aplicações. Vamos seguir o passo a passo abaixo para instalar e usar o broker localmente.

1. **Instalação do RabbitMQ**

Ubuntu/Debian

```
sudo apt-get update
sudo apt-get install rabbitmq-server
```

Após a instalação, o serviço RabbitMQ será iniciado automaticamente. Você pode verificar o status com:

```
sudo systemctl status rabbitmq-server
```

Windows

Baixe o instalador do RabbitMQ do site oficial e siga as instruções de instalação. O Erlang também precisa ser instalado, pois o RabbitMQ é escrito em Erlang.

2. **Habilitar o Management Plugin**

O RabbitMQ Management Plugin fornece uma interface de usuário baseada em web e uma API HTTP para gerenciamento e monitoramento do servidor RabbitMQ. Para habilitá-lo, execute:

```
rabbitmq-plugins enable rabbitmq_management
```

Após habilitá-lo, você pode acessar a interface web através de http://localhost:15672/. O usuário e senha padrões são guest.

3. **Trabalhando com RabbitMQ**

O RabbitMQ opera principalmente com conceitos de **exchanges, queues** e **bindings**, que estudamos anteriormente. Relembrando:

- **Exchanges**: Recebem mensagens dos produtores e as encaminham para as filas com base em regras de roteamento (bindings). Existem vários tipos de exchanges (direct, topic, fanout, headers).

- **Queues**: Armazenam mensagens até que sejam consumidas pelos consumidores.
- **Bindings**: Definem a relação entre uma exchange e uma fila, determinando como as mensagens devem ser roteadas para as filas.

Para enviar mensagens, um produtor se conecta a uma **exchange** e publica mensagens. O produtor não envia mensagens diretamente para uma fila. Em vez disso, ele envia mensagens para uma **exchange** com um **routing key** especificado, e a **exchange** encaminha a mensagem para as filas apropriadas com base nos **bindings**.

Do outro lado, um consumidor se conecta a uma fila e inscreve-se para consumir mensagens. O RabbitMQ entrega mensagens da fila para os consumidores registrados.

Vamos então criar duas aplicações em Python: uma será o cliente que envia dados ao broker (publisher) e o outro é o cliente que recebe os dados da fila (consumer). Para isso, vamos usar uma biblioteca do python chamada *Pika.* Primeiro é necessário fazer a instalação da seguinte forma:

```
pip install pika
```

Os códigos das duas aplicações são apresentados nos exemplos a seguir.

Exemplo 5.3 - Publicador (publisher.py)

```
import pika

# Conecta ao RabbitMQ
connection =
pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

# Cria uma fila
channel.queue_declare(queue='hello')

# Envia uma mensagem para a fila
channel.basic_publish(exchange='',
                      routing_key='hello',
                      body='Hello World!')

print(" [x] Sent 'Hello World!'")
connection.close()
```

Exemplo 5.4 - Consumidor (consumer.py)

```
import pika

def callback(ch, method, properties, body):
    print(f" [x] Received {body}")

connection =
pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

channel.queue_declare(queue='hello')

channel.basic_consume(queue='hello',
                      auto_ack=True,
                      on_message_callback=callback)

print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

1.3.3 CoAP

O CoAP, ou Protocolo de Aplicações Restritas, destaca-se como uma peça fundamental no cenário da Internet das Coisas (IoT), oferecendo uma solução de comunicação eficiente, especialmente adaptada para dispositivos com recursos limitados. Desenvolvido com a finalidade específica de atender às demandas desses ambientes, o CoAP baseia-se no conhecido protocolo HTTP, mas com ajustes significativos para otimizar o desempenho em termos de energia, largura de banda e resistência a perdas.

A característica distintiva do CoAP é sua eficiência energética, crucial para dispositivos IoT como sensores e microcontroladores, que frequentemente operam em ambientes com restrições severas de energia. Ao empregar o protocolo UDP em vez do TCP, o CoAP minimiza o consumo de energia, proporcionando uma comunicação eficaz mesmo em dispositivos com baterias limitadas. Esta escolha estratégica torna o CoAP uma opção ideal para aplicações de monitoramento e controle, onde a eficiência energética é essencial.

A capacidade do CoAP de lidar com redes de largura de banda limitada é outra vantagem notável. Ao utilizar pacotes compactos, o protocolo reduz significativamente o tráfego na rede. tornando-o adequado para ambientes IoT

que frequentemente dependem de redes sem fio e, consequentemente, beneficia-se de uma comunicação otimizada.

Além disso, o CoAP usa o modelo de comunicação **assíncrona** baseado na arquitetura cliente-servidor, em que os clientes podem enviar mensagens sem a necessidade de aguardar uma resposta imediata do servidor. Isso é facilitado pelos métodos do CoAP, como GET para obter dados, POST para criar novos recursos, PUT para atualizar recursos existentes e DELETE para remover recursos.

Por fim, a resistência a perdas do CoAP é crucial em cenários nos quais a latência da rede ou a perda de pacotes são comuns. Essa robustez faz com que o protocolo seja altamente confiável, garantindo a entrega de mensagens mesmo em condições desafiadoras. Vamos agora entender um pouco do funcionamento desse protocolo.

Funcionamento

O protocolo CoAP é estruturado de modo que cada mensagem inicia com um cabeçalho fixo de 4 bytes, que pode ser complementado por opções e um payload. Esse formato é adotado tanto para solicitações (requests) quanto para respostas (responses), com cada mensagem incorporando um identificador (ID) único para prevenir duplicatas.

Para assegurar a confiabilidade das transmissões, o CoAP emprega mensagens de confirmação (CON), que são retransmitidas em intervalos regulares, definidos por um tempo de espera padrão, até que um reconhecimento (ACK) seja recebido do destinatário. Alternativamente, é possível optar por mensagens não confirmáveis (NON), que não requerem ACK. Adicionalmente, o protocolo prevê mensagens de reset (RST) para casos em que o destinatário não consegue processar mensagens CON ou NON.

Em contextos de comunicação multicast, é necessário adotar precauções especiais para evitar a sobrecarga da rede devido ao excesso de mensagens. Nesse cenário, recomenda-se a utilização exclusiva de mensagens NON e a omissão do envio de RST para prevenir congestionamentos. O envio de um RST em multicast replicaria o comportamento de uma mensagem unicast, mantendo o mesmo ID de mensagem, caso o remetente permaneça ativo. Em virtude dessas

especificidades, os protocolos de segurança são geralmente desativados em transmissões multicast.

O quadro abaixo mostra o formato de mensagem do protocolo. Cada linha tem 4 bytes de tamanho e o que cada espaço significa é apresentado a seguir;

Ver	Tipo	TKL	Código	Id da Mensagem
Token (opcional)				
Opções (Opcional)				
11111111			Payload (Opcional)	

- **Ver** - Identifica a versão do CoAP;
- **T** - Define o tipo de mensagem, que pode ser: Confirmável (CON), Não confirmável (NON), Reconhecimento (ACK) e Redefinir (RST);
- **TKL** - Especifica o tamanho do campo Token (0 - 8 bytes);
- **Code** - Indica o método de solicitação/resposta, dependendo do tipo de mensagem.
- **ID da Mensagem** - Detecta a duplicação de mensagens e é usado para corresponder os tipos de mensagens ACK e RST aos tipos de mensagens CON e NON.
- **Token** - Correlaciona solicitações e respostas (tamanho de acordo com o TKL).
- **Options (Opções)** - Especifica o número da opção, comprimento e valor da opção. Os recursos fornecidos pelo campo de opções incluem especificar o recurso de destino de uma solicitação e funções de proxy.
- **Payload** - Dados que serão enviados. É opcional, já que depende do método que está sendo usado, mas quando é usado, um byte de 1's (0xFF) precede o payload.

Métodos

Como discutido anteriormente, os métodos do CoAP são baseados nos do protocolo HTTP. O método GET é empregado para solicitar informações, utilizando a URI para especificar o recurso desejado. O servidor, após avaliar

as opções, realiza as validações necessárias e aplica as regras de negócio pertinentes, processa a requisição e, em caso de sucesso, retorna um código de resposta juntamente com o conteúdo solicitado.

O método POST, similar ao seu homólogo no HTTP, é utilizado para enviar dados ao servidor (encapsulados no corpo da mensagem (payload)) com o objetivo de criar ou atualizar um recurso. Após a execução bem-sucedida da operação, o servidor emite um código de resposta adequado à ação realizada, analogamente ao que ocorre com as requisições GET.

Já o método PUT é designado para a criação ou atualização de recursos identificados pela URI fornecida. Se o recurso alvo já existir, ele será atualizado; caso contrário, um novo recurso será criado. Por fim, o método DELETE apagará algum recurso no servidor.

As respostas padrão de resposta as requisições apresentadas são:

- 2.01 (CREATED)
- 2.02 (DELETED)
- 2.03 (VALID)
- 2.04 (CHANGED)
- 2.05 (CONTENT)
- 4.xx (CLIENT ERROR)
- 5.xx (SERVER ERROR)

Outras características do protocolo

O protocolo tem a capacidade de armazenar respostas específicas, seja no próprio endpoint ou através de um intermediário, com o intuito de otimizar o tempo de resposta e reutilizar informações previamente armazenadas para melhorar a performance. Essa prática é regulada pelo conceito de "freshness", que define o período de validade de uma resposta no cache antes de seu prazo de expiração.

Em redes restritas, o uso de proxy é uma estratégia eficaz para minimizar o tráfego de rede, aprimorar a performance, facilitar o acesso a recursos em dispositivos que não estão ativos e reforçar a segurança. A eficiência dessa abordagem é ampliada quando combinada com o caching, contribuindo para uma distribuição de tráfego de rede mais equilibrada e eficiente.

A adoção da porta padrão 5683, ou 5684 quando se utiliza DTLS, facilita a oferta e localização de recursos na rede CoAP, promovendo uma configuração automática dos nós e eliminando a necessidade de ajustes manuais. Essa descoberta automática também fornece aos clientes uma lista dos recursos disponíveis no servidor, incluindo seus respectivos tipos de mídia. É possível utilizar o endereço multicast 224.0.1.187 ou FF0X::FD, oferecendo alternativas para a configuração e comunicação na rede.

Exemplo

Para exemplificar o uso do protocolo, utilizaremos a biblioteca **microcoapy** para implementar um cliente usando MicroPython e um servidor em python. O primeiro, rodará em uma placa, como o ESP32 e o servidor rodará em um computador e será implementado em python usando a biblioteca **CoAPthon**. Vamos começar pelo lado do cliente.

Vamos supor que você tenha um dispositivo rodando MicroPython e que você queira enviar dados de sensor para um servidor CoAP. Primeiramente, você precisará instalar a biblioteca **microcoapy** no seu dispositivo. Isso pode variar dependendo do seu dispositivo, mas geralmente envolve copiar os arquivos da biblioteca para ele.

Abaixo, segue um código de exemplo para a utilização da biblioteca.

Exemplo 5.5 - Cliente.py

```
from microcoapy.microcoapy import Coap
import network
import time

# Conecta à rede Wi-Fi
wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect('your-ssid', 'your-password')
while not wlan.isconnected():
    pass

# Configura o cliente CoAP
coap = Coap()
coap.start()

# Define o IP do servidor CoAP e a porta
```

```
server_ip = '192.168.1.100'
server_port = 5683

# Envia um pacote CoAP POST para o servidor
uri_path = 'sensor/data'
payload = 'temperature=24.5&humidity=60'
coap.post(server_ip, server_port, uri_path, payload, None,
Coap.CONTENT_FORMAT.COAP_TEXT_PLAIN)

# Aguarda um pouco antes de fechar
time.sleep(2)

coap.stop()
```

Agora, no lado do servidor, você pode usar a biblioteca CoAPthon para criar um servidor CoAP que escute as requisições dos dispositivos. O código a seguir mostra como utilizar essa biblioteca para “conversar” com a aplicação cliente criada anteriormente.

Exemplo 5.6 - Servidor.py

```
from coapthon.server.coap import CoAP
from coapthon.resources.resource import Resource

class SensorDataResource(Resource):
    def __init__(self, name="SensorDataResource"):
        super(SensorDataResource, self).__init__(name)
        self.payload = "Sensor Data Resource"

    def render_POST(self, request):
        self.payload = request.payload
        print(f"Received data: {self.payload}")
        return self

class CoAPServer(CoAP):
    def __init__(self, host, port):
        CoAP.__init__(self, (host, port))
        self.add_resource('sensor/data/', SensorDataResource())

def main():
    server = CoAPServer("0.0.0.0", 5683)
    try:
        print("CoAP Server Started")
        server.listen(10)
    except KeyboardInterrupt:
        print("Server Shutdown")
```

```
        server.close()
        print("Exiting...")

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

Este servidor CoAP escutará na porta 5683 e responderá a requisições POST enviadas para o caminho **sensor/data/**. Quando dados são recebidos, eles são impressos no console. Certifique-se de que o IP e a porta configurados no script do cliente correspondam ao IP e à porta em que o servidor está escutando.

1.3.4 MQTT

O MQTT (Message Queuing Telemetry Transport) emerge como uma peça fundamental no vasto quebra-cabeça da Internet das Coisas (IoT), proporcionando uma comunicação de dados leve, eficiente e altamente versátil. Este protocolo foi projetado para atender dispositivos com recursos limitados, destacando-se como uma escolha preferida em uma ampla gama de aplicações IoT.

Caracterizado como um protocolo de mensagens sem estado, ele desafia a necessidade de dispositivos manterem informações persistentes sobre as conexões de comunicação. Essa característica impulsiona a eficiência, reduzindo a carga sobre os dispositivos e otimizando o uso de recursos. Além disso, o MQTT é aclamado como um protocolo de mensagens atômicas, onde a entrega de mensagens é tratada de maneira binária - ou a mensagem é entregue, ou não é. Isso proporciona simplicidade e confiabilidade ao garantir que a integridade da mensagem seja preservada.

Nesta seção, vamos estudar esse protocolo de forma detalhada, já que é muito utilizado em diversas aplicações IoT, em diferentes cenários.

História

Nos anos 90, a IBM desenvolveu o protocolo MQTT em resposta à necessidade de uma comunicação eficiente entre várias máquinas, utilizando microcontroladores para a aquisição de dados. O objetivo era estabelecer uma

comunicação leve e eficaz entre máquinas e sensores. Em outras palavras, o protocolo foi criado e adaptado para sistemas de supervisão e coleta de dados do tipo SCADA (Supervisory Control and Data Acquisition ou Sistemas de Supervisão e Aquisição de Dados, em português), muito utilizado na indústria, e ainda possui muitas características desse cenário, porém evoluiu em diversos aspectos para utilização em soluções de IoT.

Funcionamento

A arquitetura escalável do MQTT permite que suporte um grande número de dispositivos, tornando-o altamente flexível em ambientes IoT diversificados. A operação básica do MQTT envolve a arquitetura publish/subscribe, ou seja, um dispositivo ***publisher*** enviando dados para um tópico, um ***broker*** recebendo e distribuindo esses dados para todos os ***subscribers*** inscritos no tópico, e os **subscribers** processando os dados conforme necessário, de acordo com a aplicação.

Embora o MQTT não seja tão sofisticado quanto o *AMQP*, que oferece uma gama mais ampla de funcionalidades e cenários de uso, ele mantém uma simplicidade que não compromete características essenciais como segurança, qualidade de serviço e facilidade de implementação. Essas qualidades tornam o MQTT uma escolha sólida para implementações e aplicações em sistemas embarcados, apesar da concorrência acirrada.

Basicamente, o processo de recebimento de informações em uma rede MQTT envolve a subscrição por parte de um elemento da rede (*subscribers*), que faz uma requisição ao broker, o intermediário responsável por gerenciar as publicações e subscrições. Da mesma forma, elementos que desejam publicar informações enviam-nas ao broker. Esse padrão, embora não seja novo, é comum em outros protocolos de comunicação.

No caso do MQTT, o broker “apenas” organiza as mensagens que chegam (enviadas pelos clientes) baseadas em **tópicos (*topics*)**. De forma padrão, os tópicos parecem com URIs, separando níveis por barras (“/”). Os dispositivos que “querem” receber determinadas mensagens precisam se inscrever nesses tópicos. Vamos ver esse funcionamento com um exemplo básico.

Vamos supor que, em uma rede de sensores, existam vários sensores diferentes de temperatura e umidade, publicando o valor do sensor como o

dado útil (***payload***) e identificando as mensagens com tópicos nos seguintes formatos:

```
area/ID_da_area/sensor/ID_do_sensor/temperatura
area/ID_da_area/sensor/ID_do_sensor/umidade
```

Usando esse padrão, possíveis mensagens que seriam publicadas poderiam ser como os apresentados a seguir.

```
area/10/sensor/5000/temperatura
area/10/sensor/5000/umidade
area/20/sensor/5001/temperatura
area/20/sensor/5001/umidade
area/30/sensor/4000/temperatura
area/30/sensor/4000/umidade
```

Para exemplificar a leitura de uma mensagem como essa, podemos pegar a primeira mensagem publicada: `area/10/sensor/5000/temperatura.` Essa mensagem indica o valor de temperatura, do sensor que possui o id 5000 e está instalado na área 10.

Desta forma, as mensagens do quadro anterior indicam que temos 3 áreas distintas (10, 20 e 30), com 3 dispositivos sensores diferentes (5000, 50001 e 4000) que fornecem dados de temperatura e umidade.

Para que um dispositivo recebesse essas informações, bastava se inscrever em um desses tópicos. Assim como o AMQP, alguns caracteres funcionam como *curinga*, de forma a agrupar as informações que se deseja receber.

- O caractere “+” aceita qualquer valor naquele nível do tópico. Por exemplo, a inscrição no tópico **area/20/sensor/+/temperatura** indica que se deseja receber as informações de todos os sensores de temperatura da área 20.
- O caractere “**#**” indica o recebimento de mensagens de qualquer coisa abaixo de um determinado nível do tópico. Por exemplo, um cliente que se inscrever para receber as mensagens do tópico **area/10/sensor/#,** receberia todas as informações da área 10, independente do sensor ou variável medida.

Conexão

A conexão com o *broker*, tanto de publicadores como de consumidores é feita usando TCP, com opções de login com usuário e senha, e criptografia (SSL/TLS).

No processo de comunicação é indicado o nível de qualidade de serviço (QoS). Esse valor especifica como deve ser a relação entre os dispositivos que estão se comunicando e, basicamente, podem aparecer em 3 níveis distintos:

- QoS 0 (*at most once*) - Neste nível não existe garantia de entrega da mensagem, como na utilização do protocolo UDP. Além disso, o dispositivo que envia não precisa reter a mensagem armazenada para futuras retransmissões.
- QoS 1 (*at least once*) - Neste caso, temos a confirmação de entrega da mensagem, gerando várias mensagens iguais em que, pelo menos uma, será entregue. Desta forma, o dispositivo que envia mantém essa mensagem armazenada até que uma confirmação de recebimento seja recebida.
- QoS 2 (*exactly once*) - Neste nível existe a garantia que a mensagem será entregue exatamente uma vez, tendo confirmação do recebimento e confirmação da própria confirmação do recebimento. Em outras palavras, existem confirmação para todos os dados que estão trafegando na comunicação. Assim como no anterior, até que haja essa confirmação, a mensagem é mantida no dispositivo que envia.

Exemplo - Mosquitto

Primeiro precisamos definir o broker que será utilizado. Neste sentido, existem vários brokers MQTT que podem ser usados para o desenvolvimento de soluções, em diferentes linguagens de programação. O **mosquitto** é um broker muito usado no desenvolvimento de protótipos já que é *open-source* e pode ser instalado em diferentes sistemas operacionais e plataformas.

No Windows, para fazer a instalação do **mosquitto** basta acessar o link https://mosquitto.org/download/, fazer o download do arquivo e proceder com a instalação. No linux, siga os passos abaixo:

1. Abra o terminal.

2. Atualize o índice de pacotes do sistema operacional Ubuntu:

```
sudo apt update
```

3. Instale o broker Mosquitto e o cliente MQTT usando o seguinte comando

```
sudo apt install mosquitto mosquitto-clients
```

4. Após a instalação, o serviço do Mosquitto deverá iniciar automaticamente. Você pode verificar o status do serviço com o seguinte comando. Se o serviço estiver em execução corretamente, você deverá ver uma saída indicando que o serviço está ativo e em execução.

```
sudo systemctl status mosquitto
```

Importante!

Para configurar o Mosquitto para aceitar conexões externas, você precisa ajustar as configurações de segurança e rede no arquivo de configuração principal do Mosquitto (mosquitto.conf).

1. Primeiro, localize o arquivo de configuração principal do Mosquitto geralmente está localizado em **/etc/mosquitto/mosquitto.conf**.
2. Por padrão, o Mosquitto escuta apenas conexões locais. Para permitir conexões externas, você precisa especificar um endereço de escuta externo. Para isso, você pode adicionar ou modificar a seguinte linha no arquivo de configuração:

```
listener 1883 0.0.0.0
```

   Isso diz ao Mosquitto para escutar em todas as interfaces de rede (0.0.0.0) na porta 1883. Se você quiser usar uma porta diferente, basta substituir 1883 pelo número da porta desejada.
3. Se você deseja permitir que clientes externos se conectem sem autenticação, você pode precisar ajustar as configurações de segurança. Por padrão, o Mosquitto não exige autenticação para conexões locais. Para permitir conexões externas sem autenticação, você pode adicionar ou modificar a seguinte linha no arquivo de configuração:

```
allow_anonymous true
```

Isso permite que clientes se conectem sem fornecer credenciais de autenticação.

4. Depois de fazer as alterações no arquivo de configuração, você precisa reiniciar o serviço Mosquitto para que as alterações entrem em vigor:

```
sudo systemctl restart mosquitto
```

Após essas configurações, o seu servidor Mosquitto estará configurado para aceitar conexões externas na porta especificada. Certifique-se de configurar corretamente seu firewall para permitir o tráfego na porta MQTT (por padrão, a porta 1883) se você estiver usando algum firewall no seu servidor.

Uma vez instalado, podemos utilizá-lo como broker da nossa aplicação. Agora vamos implementar o código do publisher que enviará a mensagem “Hello, MQTT” para os dispositivos inscritos no tópico “teste/topico” usando o broker Mosquitto rodando na máquina de ip *192.168.0.13*. Lembre-se que esse ip deve ser substituído pelo ip da sua máquina.

Exemplo 5.7 - publisher.py

```
import time
from umqtt.simple import MQTTClient

# Configurações do MQTT
#Endereço do broker MQTT
MQTT_BROKER = "192.168.0.13"
#Porta do broker MQTT padrão
MQTT_PORT = 1883
#Tópico MQTT
MQTT_TOPIC = b"teste/topico"

# Identificação do cliente MQTT
CLIENT_ID = "umqtt_client"

# Função para lidar com mensagens recebidas
def sub_cb(topic, msg):
```

```
    print((topic, msg))

# Conexão com o broker MQTT
client = MQTTClient(CLIENT_ID, MQTT_BROKER, port=MQTT_PORT)
client.set_callback(sub_cb)
client.connect()

# Publicação de uma mensagem no tópico MQTT
client.publish(MQTT_TOPIC, b"Hello, MQTT!")

# Espera por mensagens recebidas
client.wait_msg()

# Desconecta do broker MQTT
client.disconnect()
```

Para testar nossa aplicação, faça o upload do arquivo publisher.py em um ESP32 ou RP Pi Pico W e abra um terminal na máquina onde você instalou o **mosquitto** e digite o comando a seguir:

```
mosquitto_sub -h localhost -t "teste/topico"
```

Este comando inscreve um cliente no tópico “teste/tópico” para receber as mensagens do broker. Uma vez que tudo esteja funcionando, você deve ver a mensagem enviada pelo dispositivo no terminal que foi inscrito neste tópico.

Exemplo - Broker Externo

Podemos também usar um broker externo para testar nossas aplicações. Usaremos aqui o exemplo disponibilizado na própria plataforma Wokwi, de forma que seja possível fazer alterações de configuração, como a indicação do QoS.

A plataforma HiveMQ disponibiliza um serviço de broker MQTT gratuito para pequenos testes. Neste exemplo vamos simular uma estação meteorológica que fornece dados de temperatura e umidade e envia para o broker. Esses dados serão obtidos por meio do sensor DHT22.

Inicialmente, podemos acessar a plataforma HiveMQ, usando o link http://www.hivemq.com/demos/websocket-client/ , clicar no botão ***connect***, adicionar um novo tópico chamado “wokwi-weather-cepedi” apertando no botão **“Add New Topic Subscription”.** Desta forma, configuramos um ***subscriber*** para receber mensagens enviadas neste tópico.

Depois de iniciado o broker, podemos implementar o código da aplicação como mostrado no Exemplo 5.8.

Exemplo 5.8 - Código da Estação Meteorológica

```
import network
import time
from machine import Pin
import dht
import ujson
from umqtt.simple import MQTTClient

#MQTT Server Parameters
MQTT_CLIENT_ID = "micropython-weather-demo"
MQTT_BROKER    = "broker.mqttdashboard.com"
MQTT_USER      = ""
MQTT_PASSWORD  = ""
MQTT_TOPIC     = "wokwi-weather-cepedi"

sensor = dht.DHT22(Pin(15))

print("Connectando ao WiFi", end="")
sta_if = network.WLAN(network.STA_IF)
sta_if.active(True)
sta_if.connect('Wokwi-GUEST', '')
while not sta_if.isconnected():
 print(".", end="")
 time.sleep(0.1)
print(" Conectado!")

print("Conectando ao servidor MQTT... ", end="")
client = MQTTClient(MQTT_CLIENT_ID, MQTT_BROKER,
user=MQTT_USER, password=MQTT_PASSWORD)
client.connect()

print("Conectado!")

prev_weather = ""
while True:
 print("Medindo condições climáticas... ", end="")
 sensor.measure()
 message = ujson.dumps({
   "temp": sensor.temperature(),
   "humidity": sensor.humidity(),
 })
 if message != prev_weather:
   print("Atualizando!")
   print("Enviando para tópico MQTT {}: {}".format(MQTT_TOPIC,
message))
   client.publish(MQTT_TOPIC, message)
   prev_weather = message
```

```
else:
  print("Sem mudanças!")
time.sleep(1)
```

A montagem dos componentes é mostrada na Figura 5.5.

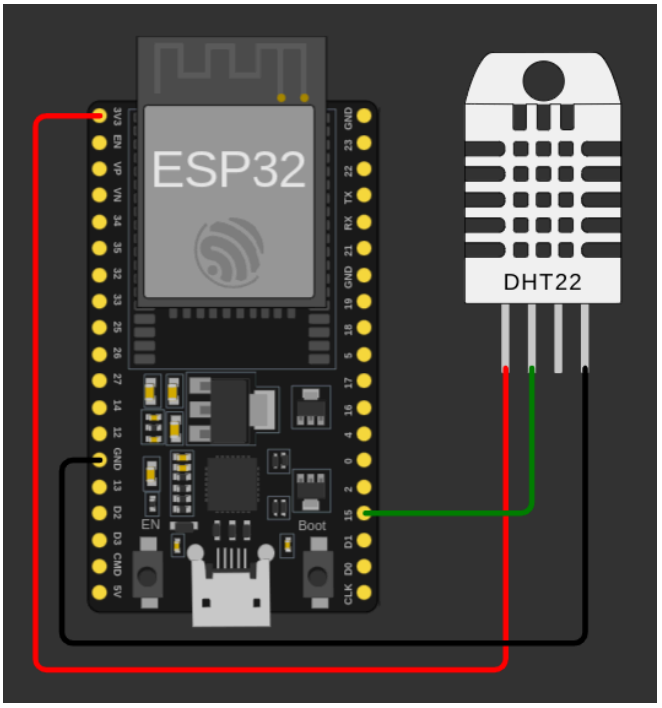


Fig. 5.5 - Montagem do sensor DHT22 ao ESP32. Fonte: Wokwi.

Exemplo - Bipes

O BIPES também fornece um broker para utilização na comunicação MQTT. Esse broker é acessado por meio da aba EasyMQTT e, para sua utilização, devemos iniciar uma sessão e utilizar as seguintes credenciais de acesso:

- server = "bipes.net.br",
- port = 1883,
- user = "bipes",
- password = "m8YLUr5uW3T"

O exemplo abaixo gera números aleatórios e os envia para o broker do Bipes. Na aba EasyMQTT podemos ver esses dados sendo plotados em um gráfico como um dashboard.

Exemplo 5.9 - Usando MQTT no BIPES

```
import umqtt.robust
import time
import random

easymqtt_session = "ng01vt";
easymqtt_client = umqtt.robust.MQTTClient("umqtt_client",
server = "bipes.net.br", port = 1883, user = "bipes", password
= "m8YLUr5uW3T");
easymqtt_client.connect()
print("EasyMQTT connected")
while(True):
  value = random.randint(0,10)
  easymqtt_client.publish(easymqtt_session + "/" + 'teste',
str(value))
  print("EasyMQTT Publish -
Session:",easymqtt_session,"Topic:",'teste',"Value:",str(value)
)
  time.sleep(1)
```

A Figura 5.6 mostra a aba EasyMQTT do Bipes recebendo os dados do broker no tópico “teste”.

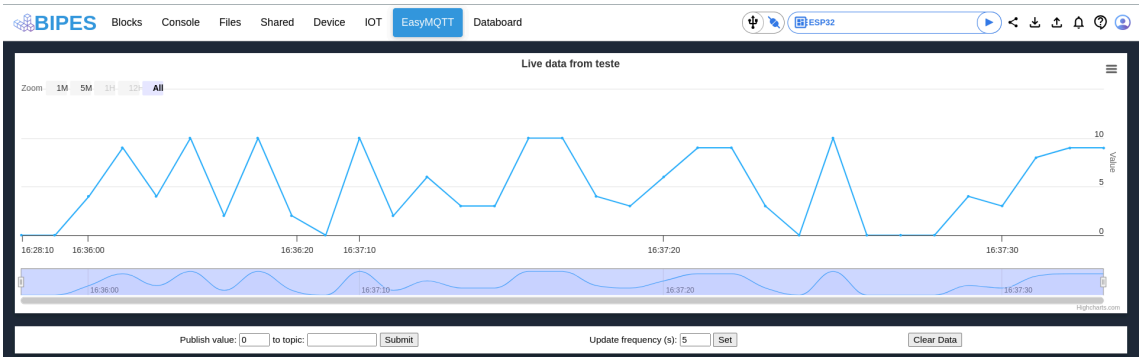


Fig. 5.6 - Aba EasyMQTT recebendo dados do broker. Fonte: BIPES.

Referências

A. G. D. S. Junior, L. M. G. Gonçalves, G. A. De Paula Caurin, G. T. B. Tamanaka, A. C. Hernandes and R. V. Aroca, "**BIPES: Block Based Integrated Platform for Embedded Systems**," in IEEE Access, vol. 8, pp. 197955-197968, 2020, doi: 10.1109/ACCESS.2020.3035083.
Full text: https://ieeexplore.ieee.org/document/9246562

SANTOS, Bruno P.; SILVA, Luiz A. M.; CELES, Carla S. F. S.; BORGES NETO, João B.; PERES, Bruno S.; VIEIRA, Marcelo A. M.; VIEIRA, Luiz F. M.; GOUSSEVSKAIA, Olga N.; LOUREIRO, Antônio A. F. (2016). **Internet das Coisas: da Teoria à Prática**. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES E SISTEMAS DISTRIBUÍDOS [SBRC], 31., 30 de maio de 2016, Salvador. Anais [...]. Salvador: UFMG, 2016. p. 1-50.

ASHTON, Kevin. **Entrevista exclusiva com o criador do termo "Internet das Coisas"**. Finep, 2015. Disponível em: <http://finep.gov.br/noticias/todas-noticias/4446-kevin-ashton-entrevista-exclusiva-com-o-criador-do-termo-internet-das-coisas>. Acesso em: 26 de outubro de 2023.

TOTVS. **Aplicações da Internet das Coisas**. Blog Totvs, 2023. Disponível em: <https://www.totvs.com/blog/inovacoes/aplicacoes-da-internet-das-coisas/>. Acesso em: 8 de novembro de 2023.

Usemobile. **IoT: 9 exemplos de aplicativos bem-sucedidos**. Usemobile, 2023. Disponível em: <https://usemobile.com.br/iot-9-exemplos-de-aplicativos/>. Acesso em: 8 de novembro de 2023.

Google. **Breaking down language barriers with augmented reality**. YouTube, 2023. Disponível em: <https://www.youtube.com/watch?v=lj0bFX9HXeE>. Acesso em: 8 de novembro de 2023.

ZUP. **A arquitetura da Internet das Coisas**. 30 de agosto de 2023. Disponível em: https://www.zup.com.br/blog/a-arquitetura-da-internet-das-coisas. Acesso em: 28 de novembro de 2023.

AMAZON WEB SERVICES. **O que é MQTT?** 20 de julho de 2023. Disponível em: https://aws.amazon.com/pt/what-is/mqtt/. Acesso em: 28 de novembro de 2023.

GTA/UFRJ. **COAP: Protocolo de comunicação para a Internet das Coisas**. 8 de março de 2019. Disponível em: https://www.gta.ufrj.br/ensino/eel878/redes1-2019-1/vf/coap/. Acesso em: 28 de novembro de 2023.

HIVEMQ. **MQTT vs HTTP: Protocolos para IoT e IIoT**. 21 de junho de 2023. Disponível em: https://www.hivemq.com/article/mqtt-vs-http-protocols-in-iot-iiot/. Acesso em: 28 de novembro de 2023.

EMBARCADOS. **AMQP: Protocolo de comunicação para IoT**. 10 de maio de 2023. Disponível em: https://embarcados.com.br/amqp-protocolo-de-comunicacao-para-iot/. Acesso em: 28 de novembro de 2023.

CONCEIÇÃO JÚNIOR, André Lisboa da. **Redes sem Fio: Protocolo Bluetooth Aplicado em Interconexão entre Dispositivos**. Disponível em: https://www.teleco.com.br/tutoriais/tutorialredespbaid/pagina_5.asp. Acesso em: 8 jan. 2024.

EMBARCADOS. **Protocolos de Rede sem Fio de IoT**. Disponível em: https://embarcados.com.br/protocolos-de-rede-sem-fio-de-iot/ Acesso em: 8 jan. 2024.

ARAÚJO, André Silva de; VASCONSELLOS, Pedro de. **Conclusão - Bluetooth Low Energy (BLE)**. Disponível em: https://www.gta.ufrj.br/ensino/eel879/trabalhos_vf_2012_2/bluetooth/conclusao.htm. Acesso em: 8 jan. 2024.

ELEMENT14 COMMUNITY. **Tech Spotlight: SigFox - A Wide Area Network Protocol for IoT.** Disponível em: https://community.element14.com/learn/learning-center/the-tech-connection/w/documents/3897/tech-spotlight-sigfox----a-wide-area-network-protocol-for-iot. Acesso em: 8 jan. 2024.

EMBARCADOS. **Uma Visão Técnica da Rede Sigfox**. Disponível em: https://embarcados.com.br/uma-visao-tecnica-da-rede-sigfox/. Acesso em: 8 jan. 2024.

GTA UFRJ. **Protocolos de Rede para Redes de Sensores Sem Fio**. Disponível em: https://www.gta.ufrj.br/grad/10_1/rssf/protocolos.html. Acesso em: 8 jan. 2024.

EMBARCADOS. **NFC (Near Field Communication) – Aplicações e uso**. Disponível em: https://embarcados.com.br/nfc-near-field-communication/ Acesso em: 23 jan. 2024.

GTA UFRN. **Protocolo Zigbee.** Disponível em: https://www.gta.ufrj.br/ensino/eel879/trabalhos_vf_2017_2/802154/zigbee.html. Acesso em: 24 jan. 2024.

BOM CURSO!

Instituto de Tecnologia