Capítulo 3

Protocolos de Aplicação para a Internet das Coisas: conceitos e aspectos práticos

Alexandre Sztajnberg, Roberto da Silva Macedo e Matheus Stutzel

Abstract

Regardless of the use of frameworks, middleware and communication technologies, the interaction between applications, services and devices in the Internet of Things is mediated by Application Protocols. The use of these protocols is facilitated by libraries, and coding is usually similar to known mechanisms, such as sockets. In this chapter we present an overview of the main Application Protocols currently available, HTTP/REST, CoAP, MQTT, AMQP and XMPP. Also, the differences between the protocols and applications for which they are most indicated are discussed. The WebSocket mechanism is also introduced to show how the protocols can be used from browsers. Examples and their execution environments illustrate the presented protocols. A more structured example completes the chapter.

Resumo

Independentemente do uso de frameworks, middleware e tecnologias de comunicação, a interação entre aplicações, serviços e dispositivos na Internet das Coisas é mediada por Protocolos de Aplicação. O uso destes protocolos é facilitado por bibliotecas, e a codificação geralmente semelhante à mecanismos conhecidos, como sockets. Neste capítulo é apresentada uma visão dos principais Protocolos de Aplicação atualmente disponíveis, HTTP/REST, CoAP, MQTT, AMQP e XMPP. Também, são discutidas as diferenças entre os protocolos e aplicações para os quais são mais indicados. O mecanismo de WebSocket é também introduzido para mostrar como os protocolos podem ser usados a partir de navegadores. Exemplos e seus ambientes de execução ilustram os protocolos apresentados. Um exemplo mais estruturado completa o capítulo.

3.1. Introdução

A Internet das Coisas (*Internet of Things*, IoT) promete facilitar a interação automática e o acesso a dispositivos industriais, de automação e domésticos, como câmeras

de vigilância, dispositivos sensores, atuadores ou monitores. Isso permitirá o desenvolvimento de aplicações que podem explorar uma quantidade enorme de dados gerados por esses dispositivos [1], [2]. Estas aplicações podem surgir de muitos domínios, como automação industrial e residencial, gestão inteligente de ambientes, saúde e cidades inteligentes ([3], [1], [4], [5], [6]).

Um dos desafios para o desenvolvimento da IoT é o número crescente e a heterogeneidade de dispositivos. Em 2010, o número de dispositivos conectados à Internet já estava na ordem de 10^9 a e espera-se que ele alcance 10^{11} até 2020 [7].

A diversidade de tecnologias usadas na IoT requer o suporte de serviços de *middleware* e estruturas de desenvolvimento com abstrações bem definidas. O *middleware* pode abstrair a complexidade tecnológica e um *framework* pode agilizar o desenvolvimento das aplicações.

Por exemplo, usar uma abordagem orientada a serviços, manipulando dispositivos como serviços de software, pode facilitar a integração e fazer uso de padrões de interação conhecidos, como *register-lookup-use* [8]. No entanto, as técnicas usadas para encapsular dispositivos em artefatos de software precisam lidar com elementos típicos de dispositivos, como sensores/atuadores e atributos específicos, como localização, estado, frequência ou intervalo de medição [8]. Além disso, mais importante, alguns dispositivos podem requerer o uso de estilos de interação diferentes, *request-response* ou *publish-subscribe*, o que pode adicionar complexidade [9], [10] [11], [12].

Iniciativas e soluções como FIWARE [13], SOFIA [14], OpenIoT [15], Particle [16], KNOT [17] e Eclipse IoT [18] de um modo geral oferecem plataformas e tecnologias que permitem o desenvolvimento de aplicações para a IoT em alto nível de abstração. *Frameworks* propostos para IoT geralmente contemplam serviços de suporte, formas de integração e interação destes serviços, e de dispositivos, com módulos-cliente das aplicações [8], [9], [19]. Também são geralmente oferecidas opções de persistência de dados entre aplicações, serviços e dispositivos, em repositórios de dados locais ou na nuvem [2]. Com isso podem ser utilizadas técnicas inteligentes atuando sobre grandes massas de dados coletados.

Órgãos de padronização como ITU-T, IEEE e consórcios como o oneM2M e OASIS trabalham na padronização de uma arquitetura para a Internet das Coisas, em suas camadas de infraestrutura, segurança e privacidade, rede, comunicação máquia-máquina (*machine-to-machine*) e serviços para as aplicações [20], [21], [22], [23], [24], [25], [26]. Um dos objetivos é orientar a concepção de produtos e sistemas, para que sejam interoperáveis.

Aplicações desenvolvidas para utilizar sistemas de suporte para IoT podem integrar e fazer uso de diversos dispositivos, de fabricantes diferentes com recursos heterogêneos e limitações específicas. Estes dispositivos podem oferecer serviços através de interfaces de software específicas e se conectar por diferentes interfaces físicas, como USB, Bluetooth, ZigBee ou Ethernet. Mesmo em um nível mais alto de abstração, cada dispositivo pode utilizar protocolos de comunicação específicos e formatos de dados diferentes [27].

A Figura 3.1 apresenta uma organização geral de elementos para a Internet das Coisas. Uma aplicação pode utilizar serviços de *middleware*, quando conveniente, para

acessar serviços e dispositivos em nível alto de abstração. A comunicação entre os serviços de *middleware* com os dispositivos e serviços pode ser feita diretamente ou através de um *gateway*[1]. Neste contexto, as arquiteturas propostas para IoT incluem Protocolos de Aplicação, orientados à interação entre processos e dispositivos ou entre dispositivos. Estes protocolos podem ser utilizados tanto pelos serviços de *middleware* — que encapsulam o código necessário — como diretamente pelas aplicações — que podem fazer um uso específico dos mesmos.

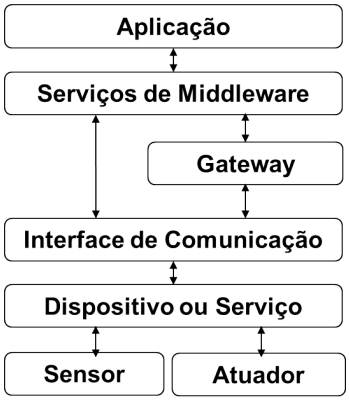


Figura 3.1. Organização de elementos na IoT.

Vários Protocolos de Aplicação estão estabelecidos e alguns padronizados, com destaque para o MQTT [29], CoAP [30], AMQP [31] e XMPP [32]. Cada um tem características indicadas para determinadas aplicações em IoT, dependendo dos recursos e modos de operação dos dispositivos ou serviços a serem integrados (Figura 3.2).

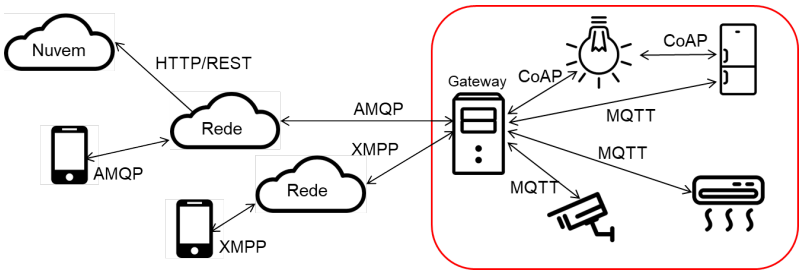


Figura 3.2. Aplicação com uso de múltiplos protocolos de aplicação.

Alguns destes protocolos contemplam aspectos não-funcionais como segurança, autenticação, QoS e escalabilidade (por exemplo, [10], [33], [11]), que podem não ser integralmente aproveitados quando um *framework* é empregado. Além disso, fabricantes podem fornecer dispositivos com algum Protocolo de Aplicação embarcado ou, ainda, um novo dispositivo pode requerer justamente o provisionamento de um Protocolo de Aplicação para realizar a comunicação com o ambiente de IoT não contemplado em um *framework*. Assim, mesmo utilizando (ou até desenvolvendo) um *framework* para IoT é importante dominar os conceitos e mecanismos destes protocolos.

[1] *Gateways* para IoT poderiam ser discutidos num capítulo à parte. Soluções em hardware e software podem ser consultadas em [28].

O objetivo deste capítulo é apresentar os principais Protocolos de Aplicação para IoT atualmente disponíveis — MQTT, CoAP, AMQP e XMPP — mencionados anteriormente, através de uma abordagem prática. Ainda, a tecnologia WebSocket [34] e a abordagem REST sobre HTTP também serão apresentadas como alternativa de suporte para a interação entre aplicações, dispositivos e serviços. Serão discutidos os conceitos principais, as diferenças entre os protocolos — relacionando estas diferenças com as aplicações e cenários para os quais são mais indicados — e apresentados exemplos práticos de programação e seus ambientes de execução. Com esta base será apresentado um estudo de caso mais estruturado com o emprego de um dispositivo NodeMCU.

As seções deste capítulo estão estruturadas da seguinte forma. Na Seção 3.2 são resumidos os estilos de interação *request-response* e *publish-subscribe*, base para os protocolos discutidos. Antes de apresentar os protocolos, a Seção 3.3 apresenta o mecanismo de WebSockets, que viabiliza o uso de vários protocolos de aplicação a partir de um navegador Web. Em seguida, a Seção 3.4 apresenta cada um dos protocolos, destacando suas principais características. Para ilustrar de forma prática o uso de cada protocolo, a Seção 3.5 apresenta um exemplo simples, junto com as bibliotecas e ambientes utilizados para o desenvolvimento e operação do mesmo. A Seção 3.6 traz um estudo de caso mais estruturado, onde se explora a conciliação de dois protocolos. Por último, a Seção 3.7 conclui o capítulo. No Apêndice 3.8 listamos referências complementares para ampliar as opções do leitor nos seus primeiros passos para o uso dos protocolos apresentados.

3.2. Estilos de interação

Sistemas de acionamento e monitoramento normalmente utilizam dois estilos de interação: requisição-resposta (*request-response*) e publica-subscreve (*publish-subscribe*) [35]. Aplicações na Internet das Coisas têm este perfil: ou a aplicação tem interesse em monitorar algum sensor — ou conjunto de sensores, ou interesse em acionar algum dispositivo atuador, ou os dois. Estas aplicações geralmente incorporam uma estrutura de laço com os seguintes passos: monitoramento, avaliação, acionamento, reconfiguração e novo monitoramento. Neste laço, ou de forma concorrente, podem estar incluídas nas atividades da aplicação a interação com o usuário, exibição de informações, etc.

Considerando os passos de monitoramento e acionamento, de uma forma geral, aplicações de IoT podem precisar interagir de forma síncrona com um processo, agente ou dispositivo remoto. Isto é, a aplicação envia uma mensagem com uma requisição e só pode continuar suas atividades depois de receber uma resposta, ou uma recusa.

Outra possibilidade é a aplicação IoT não poder, ou não querer, aguardar uma resposta, mas receber uma notificação de forma assíncrona. Isto é, a aplicação registra sua requisição junto ao seu alvo, e continua sua execução logo em seguida. Num momento posterior, uma notificação pode chegar, e a aplicação deve estar preparada para tratar a mesma. Este estilo de interação é útil quando um evento é gerado, um dado estado é atingido ou uma computação concluída por uma entidade remota e, tão logo ocorra, isso deva ser informado à aplicação.

3.2.1. Direta - Requisição-Resposta (*Request-Response*)

Protocolos de Aplicação que utilizam o estilo *request-response* são estruturados para oferecer interação síncrona e direta entre dois processos (Figura 3.3).

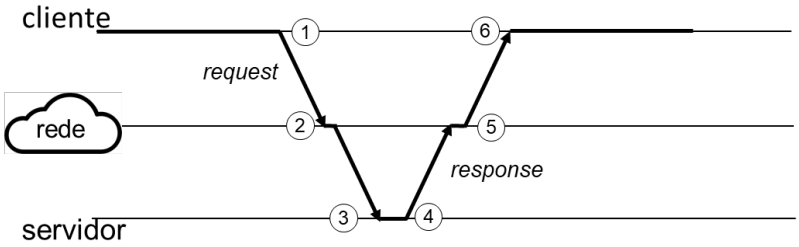


Figura 3.3. Estilo de interação *request-response*.

1. O processo cliente, ou enviante, envia uma mensagem contendo uma requisição para o processo servidor, ou recebedor e em seguida é bloqueado para aguardar a resposta;

2. A mensagem contendo a requisição precisar ser transportada pela rede, passando pelas barreiras de segurança, até que seja entregue ao servidor;

3. O processo servidor, que estava bloqueado, aguardando uma requisição, recebe a mensagem, interpreta o seu conteúdo e realiza o serviço solicitado na requisição;

4. A requisição pode, por exemplo, solicitar algum cálculo/processamento do servidor, a leitura de uma variável de um sensor ou o acionamento de um atuador. Se uma resposta é necessária, uma nova mensagem é preparada contendo esta resposta e retornada para o cliente;

5. A mensagem de resposta também precisa ser transportada pela rede, passando pelas barreiras de segurança no caminho de volta;

6. Finalmente, é recebida pelo processo cliente, requisitante, que se desbloqueia, avalia resposta e prossegue sua execução.

É importante observar que as duas trocas de mensagens ocorrem de forma direta: os dois elementos cliente e servidor precisam estar envolvidos diretamente. A requisição só é recebida pelo servidor depois que o cliente envia a mensagem, e esta é recebida com algum atraso devido à latência da rede e dos sistemas de suporte à troca de mensagens. Se o servidor se prepara para receber a requisição antes da mensagem ser transmitida, este é bloqueado até que a mesma esteja disponível.

Por outro lado, se uma resposta é necessária, os papéis são invertidos. Logo após o envio da requisição o cliente se coloca pronto para receber uma mensagem contendo a resposta e, por isso, é bloqueado. O servidor prepara uma mensagem com a resposta e a envia para o cliente. Este é desbloqueado apenas quando a mensagem é efetivamente entregue.

Uma característica inerente à comunicação direta, o processo cliente precisa conhecer a referência ou endereço do processo servidor, geralmente o número IP, a porta e o protocolo de transporte utilizado (no caso do presente capítulo, UDP ou TCP). O servidor, por sua vez, deve expor uma referência ou endereço conhecido e possível de ser obtido pelo processo cliente. Além disso, elementos de segurança, como *firewall* e NAT, precisam ser configurados para o endereço do servidor.

3.2.2. Indireta - Publica-Subscreve (*Publish-Subscribe*)

O estilo de interação indireta, publica-subscreve (*publish-subscribe*), também chamado de interação baseada em eventos (*event-based*), é apropriado para comunicação assíncrona, quando os elementos que precisam interagir, não podem ou não querem estabelecer uma comunicação direta para trocar mensagens, ou não podem aguardar bloqueados a interação ocorrer. Ainda, também é a opção adequada quando a informação a ser comunicada pode ser produzida à qualquer momento, como por exemplo, uma variável medida por um sensor atinge um valor ou limiar.

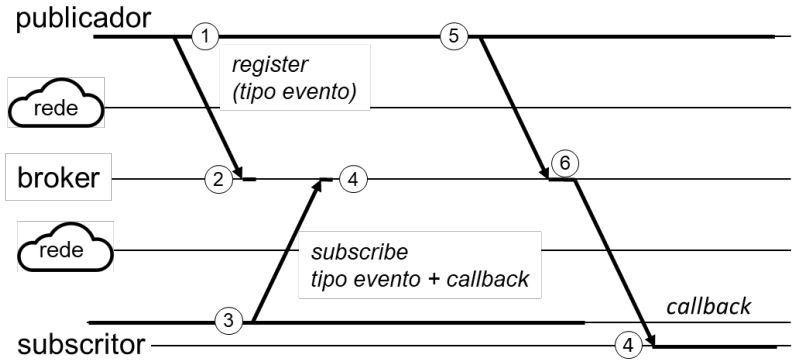


Figura 3.4. Estilo de interação *Publish-Subscribe*.

O suporte para este estilo de interação é mais complexo, se comparado com o estilo *request-response*. Seguindo a Figura 3.4, observamos os seguintes elementos:

- Publicador (*publisher*) ou fonte de evento (*event source*), é o elemento que pode gerar eventos. A aplicação pode interagir com vários publicadores.
- Evento, é uma abstração que pode ser representada por uma estrutura de dados ou objeto. Os eventos podem ter um tipo, identificação ou atributos, que indicam a sua natureza ou origem.
- Subscritor (*subscriber*) ou sorvedor de evento (*event sinc*), é o elemento que está interessado em receber notificações quando eventos de determinado tipo ocorrem. A aplicação pode trabalhar com vários subscritores concorrentemente.
- O subscritor interessado precisa informar uma referência de *callback* ou tratador (*handler*), para onde a notificação deve ser enviada e o evento entregue. Essa referência pode ser padronizada, pode ser um endereço, ou um nome de procedimento/método.

- Sistema de eventos (*event engine*) ou *event broker* ou simplesmente *broker* (termo mais comum nos sistemas de IoT), intermedeia as interações entre publicadores e subscritores, incluindo os pedidos de registro e interesse em eventos, e o encaminhamento das notificações.

Como existem mais elementos e mais trocas de mensagens entre os elementos, os atarsos de comunicação poderiam interferir mais, se comparado ao *request-response*. Por outro lado, observe-se também na Figura 3.4 que as linhas de interação entre os elementos ocorrem de forma independente ao longo do tempo, reforçando a ideia de uma interação assíncrona. Assim, os atrasos da rede podem ter importância menor.

1. O publicador primeiro envia uma mensagem para o *broker* pedindo o registro de um tipo de evento;
2. O *broker* recebe a mensagem e registra o tipo de evento.

   Em alguns casos, esta sequência pode ser feita por configuração administrativa.
3. O processo subscritor envia uma mensagem para o *broker* registrando interesse em um tipo de evento, e passa a referência de *callback*;
4. O *broker* verifica os aspectos de segurança – quando isso fizer parte de sua operação – e, então, adiciona a referência informada pelo subscritor na lista de interessados pelo evento;

   Depois de registrar o interesse em um evento, o processo subscritor pode continuar seu processamento.
5. Em algum momento qualquer, o publicador gera um evento e envia uma mensagem ao *broker* solicitando sua publicação;
6. Recebendo a publicação, o *broker* percorre a lista relacionada ao tipo de evento e envia uma mensagem de notificação para cada subscritor interessado, em sua referência da *callback*;
7. A notificação é recebida na referência de *callback* pela rotina de tratamento, *handler*, sem que a rotina original seja interrompida. A arquitetura de software necessária para o tratamento concorrente das atividades “normais” e de notificações não é aprofundada aqui.

A necessidade de um elemento intermediário, como o *broker*, introduz um ponto de complexidade: temos mais um elemento, que precisa executar como um serviço contínuo, que por sua vez precisa ser mantido/administrado. Entretanto, como ressaltado nas próximas seções, este elemento pode ser responsável por procedimentos de segurança e controle de acesso, o que é positivo. Pode também oferecer opções de enfileiramento, filas com prioridade diferenciada e persistência de eventos, permitindo que um subscritor que estava *offline* receba notificações passadas. Um elemento central como o *broker* também pode facilitar a comunicação de elementos atraz de redes com NAT e *firewall*, sem a necessidade de esforços de configuração nos elementos de borda da rede.

Por último, uma dica que pode ajudar o entendimento do uso dos protocolos. Os mecanismos relacionados ao estilo de interação *publish-subcribe* podem ser usados em uma aplicação de uma maneira um pouco "invertida" no caso de atuadores. Por exemplo, digamos que a aplicação precise acionar um relé controlado por um dispositivo Arduino. Digamos também, que uma aplicação Android permita ao usuário controlar este relé remotamente, enviando um "comando" de liga-desliga. O Arduino controlando o relé poderia ter o papel de subscritor, registrando o interesse em "eventos de liga-desliga". A aplicação Android, por sua vez poderia fazer o papel de fonte de evento do tipo "liga-desliga", enviando um evento quando o usuário assim acionasse, que por sua vez levaria ao *broker* notificar o dispositivo Arduino, que então atuaria ligando ou desligado o relé.

3.3. WebSockets

O WebSocket [34] é um mecanismo similar ao *socket* "tradicional", proposto para oferecer suporte para troca de mensagens bidirecionais, *full-duplex*, sobre TCP, entre um navegador Web e um servidor HTTP remoto, que suportem HTML5 (Figura 3.5). Navegadores como, por exemplo, o Internet Explorer, Mozilla Firefox, Google Chrome, Safari e Opera já possuem ou estão viabilizando suporte ao WebSocket.

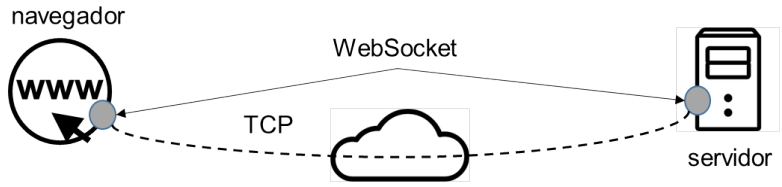


Figura 3.5. Comunicação com WebSocket.

O mecanismo WebSocket cria uma conexão que pode ser mantida aberta por um longo tempo, com pouco impacto no desempenho do navegador. É possível abrir e fechar conexões a qualquer momento. Com isso passa a ser também possível receber notificações, sem a necessidade de uso de "truques" como o *polling*, com vantagens na estrutura dos programas, melhor desempenho e latências menores.

O WebSocket tem características específicas em relação ao "socket" tradicional, dado que o contexto de uso é, em princípio, específico ao ambiente Web. Existe a preocupação de compatibilidade com o HTTP. Por isso, existe um protocolo WebSocket, padronizado pelo IETF [36] na RFC 6455, que gerencia o ciclo de vida de uma conexão WebSocket e permite, também, que uma conexão iniciada por HTTP chaveie para uma conexão WebSocket. Isso é feito através de uma mensagem HTTP *UPGRADE*.

Para o desenvolvimento das aplicações é proposta uma API de programação, padronizada pelo W3C [37]. A API para WebSockets abre a possibilidade para que aplicações para IoT possam também utilizar os navegadores como parte da solução para acionar diretamente um atuador ou obter uma informação de algum sensor. Este é o ponto explorado aqui.

O programador deve utilizar uma linguagem de programação Web, como JavaScript, e bibliotecas de suporte ao mecanismo WebSocket para desenvolver uma aplicação

cliente executando em um navegador. Após abrir uma conexão TCP utilizando WebSocket, o programa pode enviar ou receber mensagens contendo qualquer tipo de informação. É possível criar um protocolo customizado para uma aplicação específica, mas também é possível transportar protocolos padronizados, como os Protocolos de Aplicação discutidos neste capítulo como o MQTT, AMQP e XMPP[2].

A API WebSocket requer que o programador descreva a referência, ou endereço, do servidor remoto através de uma URL, e informe um esquema específico *ws* ou *wss* (para a versão segura), por exemplo: `ws://lcc.uerj.br:8000/`. A versão segura utiliza a camada SSL/TLS como no HTTPS.

O fluxo de interação entre o navegador e o servidor Web segue àquele utilizado por *sockets* TCP (Figura 3.6).

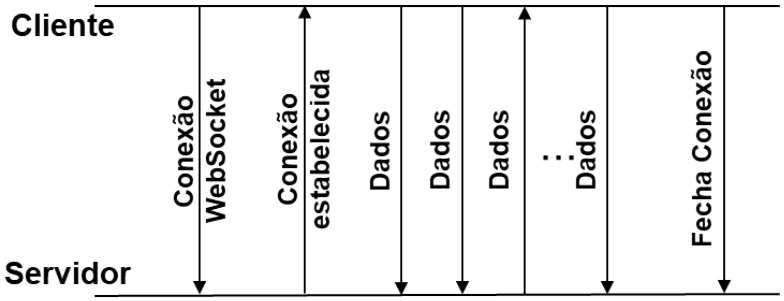


Figura 3.6. Conexão, troca de mensagens, desconexão com WebSocket.

A API do mecanismo WebSockets oferece primitivas para a criação, conexão, envio-recebimento de mensagens e término da conexão. Também oferece alternativas para recebimento de eventos e integração com o HTTP e mecanismos do navegador. Vamos utilizar um exemplo de código para introduzir a API e discutir o fluxo de interação (Código 3.1 e Código 3.2). Tanto o cliente quando o servidor utilizam o Node.js [38], e fazem uso dds mecanismos de notificação disponíveis neste tipo de ambiente.

```
1 const WebSocket = require('ws');
2 const cliWS = new WebSocket('ws://lcc.uerj.br:8080/');
3 cliWS.addEventListener('open', function sndRequest() {
4   var obj = new Object();
5   obj.a = 15;
6   obj.b = 16
7   cliWS.send(JSON.stringify(obj));
8 });
9 cliWS.addEventListener('message', function recResponse(data) {
10  console.log(data);
11  cliWS.close();
12 });
```

Código 3.1. Exemplo de programa cliente Websocket.

[2]O CoAP é baseado em UDP e, em princípio não pode ser transportado por TCP (Seção 3.4.2)

Inicialmente o cliente carrega a biblioteca de suporte (linha 1) e, em seguida, cria o Websocket, passando como parâmetro a URL do servidor (linha 2). O cliente precisa aguardar a conexão ser estabelecida e para isso registra a função *sndRequest()*, que é executada quando a conexão estiver pronta (evento *open* do *cliWS*), linha 3. Quando isso ocorre, um objeto é criado e enviado em uma estrutura *JSON*, linha 7. Por último, a função *recResponse(data)* é registrada para ser executada quando uma mensagem de retorno chegar (linha 9). Quando esta estiver disponível, a informação de resposta é logada e a conexão fechada (linha 11).

A execução do cliente em um navegador pode ser feita diretamente pelo ambiente de Node.js ou simplesmente carregando uma página HTML contendo uma *tag* para executar o código a partir de um arquivo.

```
1 const WebSocket = require('ws');
2 const wsServer = new WebSocket.Server({ port: 8080 });
3 wsSocket.on('connection', function execServ(activeWS) {
4   activeWS.on('message', function recvRequest(msg) {
5     console.log('received: %s', msg);
6     obj = JSON.parse(msg);
7     var resp = new Object(); resp.value = obj.a+obj.b;
8     activeWS.send(JSON.stringify(resp));
9   });
10 });
```

Código 3.2. Exemplo de programa servidor WebSocket.

O programa do servidor segue a mesma linha (Código 3.2). Inicialmente, é instanciado um WebSocket "servidor", passivo, especializado em receber pedidos de conexão, associado à porta 8080 (linha 2). Isso resolve em parte o problema do cliente conhecer a referência do servidor, pois a porta é fixa. Em seguida a função *execServ(activeWS)* é registrada para execução quando um pedido de conexão for aceito (linha 3). Como resultado de uma nova conexão, um Websocket ativo é criado (*activeWS*), semelhante ao que ocorre com o *socket* TCP tradicional. Cada conexão estabelecida recebe uma instância diferente da função *execServ*, responsável por tratar a conexão com o cliente de forma concorrente (detalhes de como isso acontece fogem do escopo do capítulo). A execução do serviço no exemplo é simples: a função *recvRequest* é registrada para ser executada quando a mensagem *msg* for recebida (linha 4). Esta função loga a mensagem, obtém as informações transmitidas (esta seria a requisição), executa a soma (esta seria a resposta) e envia a mesma como uma estrutura *JSON* pelo WebSocket ativo (linha 8). Observa-se que no servidor não fechamos a conexão, pois ele está preparado para receber novas conexões.

O código apresentado para o servidor precisa de algumas adaptações para executar no contexto de um servidor Web, como o Apache, ou de um Servidor de Aplicação, como o GlassFish. Mas o uso da API de WebSocket é essencialmente o mesmo.

Na próxima seção os Protocolos de Aplicação serão apresentados. O WebSocket será utilizado depois, na seção de exemplos, e também como mecanismo de suporte para o uso destes protocolos em navegadores Web.

3.4. Protocolos

Nesta seção apresentamos os Protocolos de Aplicação para IoT. Para cada protocolo serão discutidos os conceitos básicos, mecanismos/estruturas principais e os estilos de interação suportados. Também serão abordados aspectos específicos, que distinguam os protocolos como, por exemplo aspectos de QoS, diferentes esquemas de filas, facilidade para passar por NAT e *firewall*, registro, autenticação e mecanismos de segurança disponíveis. Também importante, serão apresentadas as soluções para identificação de referências, endereços, eventos e filas, quando pertinente.

3.4.1. HTTP/REST

O *Hypertext Transfer Protocol*, HTTP [39], é um protocolo com estilo *request-response*, descrito na RFC 2616, base para a comunicação de dados para a *World Wide Web*, WWW, desde 1990. Requisições do navegador Web e a resposta do servidor são transportadas por HTTP. Como outros protocolos de aplicação da Internet, o HTTP tem as informações de controle transmitidas como cadeias de caractere.

O HTTP é **media independent**, ou seja, qualquer tipo de dado pode ser enviado, desde que o cliente e o servidor saibam como lidar com os conteúdos. A informação efetivamente sendo transportada pode ter um dos tipos previstos no padrão, usando o tipo *MIME*, descrito em um campo chamado *content-type*, como por exemplo, imagens, arquivos compactados, textos em HTML ou XML.

O TCP é utilizado como protocolo de transporte. Nas primeiras versões do HTTP a conexão era desfeita logo após uma resposta ser retornada para o cliente. Na versão 1.1, descrita na RFC 2616 [39], o protocolo passa a manter a conexão aberta entre cliente e servidor, permitindo que sequências de requisições e respostas sejam tratadas, melhorando o desempenho.

Também é possível prover segurança na comunicação entre um cliente e servidor HTTP utilizando a versão segura, HTTP *Secure*, que executa sobre uma camada SSL/TLS.

O HTTP faz uso do Identificador Uniforme de Recursos, *Uniform Resource Identifier*, URI [40], como referência para o servidor com o qual a conexão será estabelecida, e para determinado recurso hospedado neste servidor. No seguinte exemplo, (i) *http* é chamado de esquema, e indica o protocolo ou tecnologia utilizada; (ii) *www.lcc.uerj.br* é chamado de domínio, e será convertido para um número IP após consulta ao DNS; (iii) *8080* é a especificação da porta onde o servidor espera receber pedidos de conexão. Caso não seja informado, o esquema vincula a porta a um padrão (80, no caso do HTTP); (iv) *laboratorio* indica a rota ou caminho para o diretório onde o recurso está disponível; e (v) *recurso* é a referência ao recurso desejado.

```
http://www.lcc.uerj.br:8080/laboratorio/recurso
```

Uma vez estabelecida a conexão, mensagens HTTP de requisição e reposta são transmitidas. São padronizados alguns tipos de mensagem, chamados de métodos no HTTP, que podem provocar reações diferentes do servidor. Nos exemplos deste capítulo utilizaremos:

GET. Recupera um *recurso* do *servidor*, dados fornecidos na URI.

POST. Envia dados para o servidor no corpo da mensagem HTTP, como por exemplo, informações de texto, *upload* de arquivos, ou informações contidas em formulários HTML. O servidor precisa estar configurado e preparado para realizar alguma ação com as informações enviadas. O exemplo "clássico" são os programas acionados pelo mecanismo de CGI, disparados quando o servidor identifica o parâmetro *action* dentro de um formulário HTML.

Outros métodos disponíveis são: *HEAD*, *PUT*, *DELETE*, *CONNECT*, *OPTIONS* e *TRACE*.

Por padrão, as mensagens de resposta contém um código de retorno, entre os quais: (i) 200, indica sucesso; (ii) 404, indica recurso não encontrado e (iii) 500, erro interno do servidor.

O Código 3.3 apresenta trechos das mensagens de requisição e resposta utilizando o método *GET*. Observa-se que as informações de controle aparecem em texto pleno.

```
1 GET / HTTP/1.1
2 Host: www.lcc.uerj.br
3 Connection: keep-alive
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) [...]
5 Accept: text/html,application/xhtml+xml,application/xml; [...]
6 Accept-Encoding: gzip, deflate
7 Accept-Language: pt-BR,pt;q=0.9,en-US;q=0.8,en;q=0.7

8 HTTP/1.1 200 OK
9 Date: Wed, 11 Apr 2018 23:22:40 GMT
10 Server: Apache/2.2.15 (CentOS)
11 Connection: close
12 Content-Type: text/html
```

Código 3.3. Mensagens de requisição e resposta HTTP com método *GET*.

A razão central para considerar o HTTP como suporte à interação entre processos, dispositivos e serviços em uma aplicação para a IoT é utilizar um protocolo de aplicação provadamente funcional, disseminado, que executa sobre uma infraestrutura de comunicação já estabelecida, com segurança geralmente configurada.

Os métodos disponíveis *PUT*, *GET*, *POST* e *DELETE*, e a abordagem de campos e meta-informações do HTTP podem ser relacionados a uma interface CRUD (*Create* (Criar), *Retrieve* (Consultar), *Update* (Atualizar) e *Delete* (Apagar)), amplamente usada em sistemas de informação. Estes poderiam ser utilizados para acionar atuadores ou obter informações de sensores, por exemplo. O problema, em princípio, seria: (i) mudar a interpretação padrão dos servidores Web para estas operações e (ii) prover um mecanismo para executar ações fora do contexto do processo do servidor Web. Por exemplo, o método GET, por padrão, implica em se percorrer o *caminho* até o diretório onde o *recurso* se encontra e em seguida recuperar o arquivo cujo nome é "recurso" e enviar o mesmo como resposta para o cliente. Outro aspecto desta interpretação é que o conteúdo de *recurso* é estático, à menos que seja alterado manualmente. No caso de IoT queremos chamar um

procedimento para acionar um atuador, ler um sensor ou solicitar um serviço, por exemplo. Neste caso, a execução e a resposta podem ser diferentes a cada chamada.

Ao longo dos anos de operação da Web, alguns mecanismos foram propostos para utilizar a sua infraestrutura para executar procedimentos remotos. O *Common Gateway Interface*, CGI, foi um dos primeiros mecanismos propostos [41]. A cada solicitação, um ambiente de execução é criado e um programa carregado e executado. Parâmetros de entrada podem ser passados como variáveis de ambiente para o programa a ser executado. A resposta é, geralmente, um texto em HTML criado dinamicamente pelo programa. Várias evoluções foram propostas.

Os Serviços Web (*Web Services*) são uma outra abordagem, mais eficiente que o CGI, que permite que um processo cliente envie uma requisição para execução remota de procedimento. Para isso, é enviando para o servidor o nome do procedimento e uma lista de parâmetros de entrada. O serviço é realizado e uma resposta é retornada para o cliente. Conceitualmente, funciona como o mecanismo *Remote Procedure Call*, RPC, utilizando também o conceito de interface, descritas em WSDL, e é aderente ao padrão SOA (*Service Oriented Architecture*). Requisições e respostas são representadas em XML. Um esquema XML, chamado SOAP, regulamenta como os elementos das requisições e respostas devem ser estruturados em arranjos e *tags* XML específicos. *Web Services* são amplamente suportados por bibliotecas, servidores HTTP e servidores de aplicação especializados. A abordagem é bem completa e tem um padrão estabelecido [42].

Sistemas e aplicações para IoT podem ser baseadas em *Web Services*. Entretanto, a abordagem tem custo computacional notável. Este custo pode ser atribuído às rotinas de *parsing* das estruturas XML, pela necessidade de ser manter os elementos de suporte para descrição de interfaces em WSDL e pelo custo dos mecanismos que encaminham as informações de uma chamada para a entidade que vai executar a mesma.

REST sobre HTTP. A abordagem *REpresentational State Transfer*, REST, tem se mostrado um alternativa interessante para IoT [43], [44]. Independente de aspectos conceituais discutidos em [45], REST permite o uso da infraestrutura do HTTP para acionar ou obter recursos, apenas dando uma interpretação diferente para os métodos *GET*, *PUT*, *POST* e *DELETE*, e valendo-se da possibilidade de enviar informações adicionais numa mensagem HTTP (que podem ser interpretados como parâmetros de entrada). O conceito em torno de REST discute que a abordagem não é um RPC, que o melhor uso não é fazer chamadas remotas de procedimentos com uma possível resposta como retorno. Entretanto, algumas infraestruturas para IoT utilizam REST para ler e acionar dispositivos transferindo estruturas JSON nos dois sentidos, como por exemplo [13], com o objetivo de se ter um mecanismo de interação distribuído, com custo aceitável executando sobre HTTP.

Todo mecanismo que estende a interpretação padrão dos métodos do HTTP precisa configurar o servidor Web com este objetivo. Isso vale para servidores bem estruturados como o Apache [46] ou servidores customizados, como veremos na seção de exemplos. Sem entrar em detalhes, e apenas considerando o REST, esta configuração pode ser feita confinando as aplicações ou serviços REST em um diretório específico, que por sua vez terá configurado o comportamento desejado[3].

[3]Uma tática simples é utilizar um arquivo como o *.httaccess* com as configurações e transformações

Neste capítulo considera-se, então, a estratégia de combinar o protocolo HTTP e a abordagem REST como opção de Protocolo de Aplicação para integrar processos, dispositivos e serviços em aplicações para IoT. Os serviços ou dispositivos oferecidos como recursos para aplicações IoT precisam aderir ao padrão arquitetural do REST, passando a ser "considerado" RESTful. Como qualquer outro recurso referenciado e recuperado por HTTP, um serviço RESTful, precisa estar localizado num rota alcançável pelo servidor Web no sistema de arquivos local. O servidor Web utiliza, então, as informações enviadas pelo cliente na mensagem HTTP: (i) a *rota*, para localizar o recurso e (ii) o *recurso*, utilizado como nome do serviço RESTful a ser acionado. Por exemplo, o trecho código HTTP seguinte, extraído do exemplo na Seção 3.5.2 mostra o método *PUT* indicando o acionamento do recurso *vermelho*, localizado em */servicos/LEDServer/* no host *teste.lcc.uerj.br*. Como resposta uma mensagem HTTP com código *200 OK* é devolvida.

```
PUT /servicos/LEDServer/vermelho HTTP/1.1
Host: teste.lcc.uerj.br
...
HTTP/1.1 200 OK
```

A URI utilizada seria, por exemplo:

```
http://teste.lcc.uerj.br:8080/servicos/LEDServer/vermelho
```

O mecanismo REST não precisa considerar detalhes da implementação do serviço RESTful. Por outro lado, o mecanismo pode usar as informações de controle do HTTP para complementar a definição de como o serviço será executado. Campos customizados podem ser usados para transferir parâmetros para o serviço. Mesmo não sendo recomendado, o próprio campo com a informação útil sendo transportada pode transferir dados em notação XML ou JSON. A configuração definida no servidor para tratar a requisição REST e a própria implementação do serviço é que vão definir como estas informações são utilizadas e como o serviço será acionado [47].

Em resumo, para desenvolver aplicações HTTP/REST o programador precisa:

- desenvolver o serviço e implantá-lo no servidor, na localização adequada;
- configurar o servidor para acionar corretamente o serviço;
- desenvolver a aplicação cliente montando a URI e informações adicionais a serem inseridas na mensagem HTTP. Bibliotecas e IDEs amplamente disponíveis facilitam a programação (Seção 3.5.2).

Como última observação, o uso de HTTP não é obrigatório para acionamento de serviços com abordagem REST. Um exemplo é a adoção desta abordagem no protocolo CoAP descrito na próxima seção. Na Seção 3.5.2 apresentamos uma aplicação HTTP/REST, utilizando alguns dos aspectos discutidos aqui.

necessárias, no caso de um servidor Apache.

3.4.2. CoAP

O *Constrained Application Protocol*, CoAP [30], é projetado para permitir a troca direta de mensagens entre dispositivos com poucos recursos computacionais, interconectados por redes com baixas taxas de transferência, com a 6LowPAN [48], por exemplo. Os dispositivos considerados geralmente são construídos com microcontroladores de baixo custo, como por exemplo o Arduino, NodeMCU e outros. O CoAP é descrito na RFC 7252 do IETF CoRE (*Constrained RESTful Enviroments*) Working Group.

O estilo de interação oferecido pelo CoAP é o *request-response*, com características semelhantes ao HTTP/REST. Como na *Web*, os dispositivos são referenciados usando o endereço IP e o número da porta. Entretanto, por estratégia de projeto, o CoAP é executado sobre UDP e não TCP. Isso pode dificultar a implantação de uma aplicação cujos elementos estejam distribuídos em redes atrás de NAT e firewall. A configuração, neste caso, deve considerar tanto os clientes como os servidores dado que não existe uma conexão. Assim, o CoAP é mais facilmente usado em uma rede local.

O CoAP não foi desenhado para substituir o HTTP, mas ele implementa um pequeno subconjunto de práticas HTTP amplamente aceitas, e as otimiza para a troca de mensagens entre dispositivos (*machine-to-machine*, M2M) [23].

O acesso aos serviços expostos pelo dispositivo se dá por URIs REST. Também de forma semelhante ao HTTP, são previstos tipos de requisição, por exemplo *GET*, *PUT*, *POST* e *DELETE*, com códigos de resposta, como por exemplo, 404, 500 e a especificação de um tipo de conteúdo usados para transmitir informações.

As mensagens transportadas pelo CoAP tem formato binário, permitindo o transporte de qualquer formato de dados, como por exemplo XML, JSON ou qualquer outro desejado. Possui um cabeçalho fixo de 4 bytes e um *payload* máximo que deve caber em um único datagrama UDP, ou no *frame* IEEE 802.15.4 [48].

O CoAP pode também usar o DTLS como protocolo de transporte para comunicação segura. Além disso, para contornar a falta de garantia de entrega de pacotes do UDP, o CoAP oferece dois níveis de qualidade de serviço: *Confirmado*, que adiciona um pacote *ACK* de confirmação de recebimento, e *Não-Confirmado*, sem mensagem de confirmação.

As opções na troca de mensagens dão flexibilidade ao CoAP. Por isso, discutimos algumas opções a seguir.

Confirmado com *piggbacking*. O cliente envia uma mensagem *CON* para o servidor solicitando a temperatura (Figura 3.7). O tipo da requisição é *GET*, a URL do caminho é "sensores/temperatura". O *ID* da mensagem é um número de 16 bits usado para identificar exclusivamente uma mensagem e ajudar o servidor na detecção de mensagens duplicadas. Assim que o servidor recebe a mensagem *CON*, a temperatura é obtida e uma mensagem de confirmação *ACK* é retornada, com o mesmo *ID*. Junto com a confirmação, a mensagem também leva “de carona” os dados de temperatura solicitada, no caso *30C*. Por fim, a resposta também tem um código de mensagem, neste caso, "2.05 Content". Estes são semelhantes aos códigos de status HTTP.

Se uma confirmação não for necessária, ou se for aceitável para a aplicação eventualmente não receber uma resposta, uma mensagem do tipo *NON* pode ser utilizada.

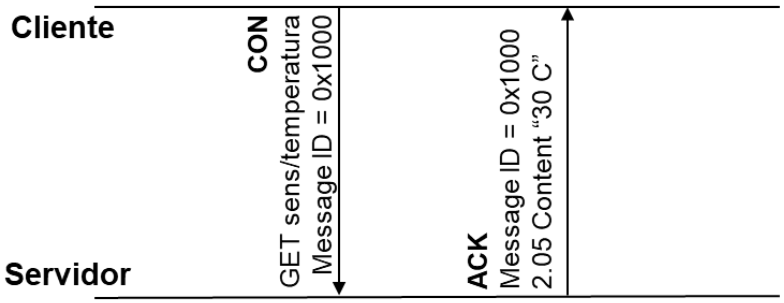


Figura 3.7. Mensagem CoAP confirmada com resposta *piggybacked*.

Confirmado sem *piggybacking* (correlação de mensagens). No caso em que a resposta para uma mensagem confirmada não possa ser levada de " carona", *piggybacked*, o ACK é enviado rapidamente, por exemplo, no caso do servidor precisar de um tempo maior para processar a mensagem. Posteriormente, uma resposta separada é enviada de volta ao cliente.

Como o CoAP é transportado por UDP, não existe uma conexão estabelecida: cada mensagem é um datagrama independente (por isso não pode ser usado com WebSockets). Assim, mensagens de requisição e resposta precisam ser correlacionadas pelo protocolo, dado que a interação deve ser coerente. A solução é o uso de um *token* para identificar o par de mensagens requisição-resposta (Figura 3.8).

Depois que o servidor tiver concluído o serviço, ele enviará uma outra mensagem *CON* contendo a resposta. Esta resposta possui um novo *ID* de mensagem e deverá ser confirmada posteriormente por um ACK correspondente. Entretanto, o valor do *token* desta resposta é o mesmo da requisição, permitindo ao cliente correlacionar as mensagens.

De uma forma geral, o *token* é usado para correlacionar mensagens, simulando uma conexão, mesmo que a mensagem seja não-confirmada e os IDs diferentes.

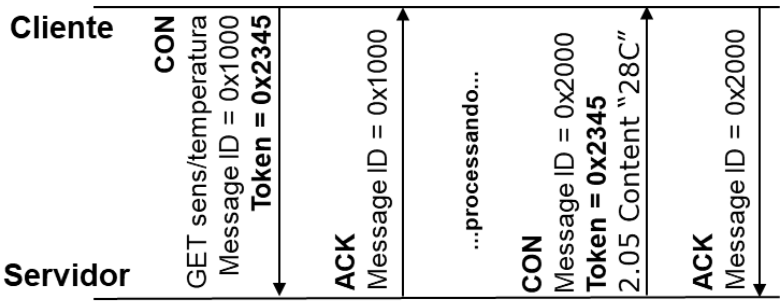


Figura 3.8. Mensagem CoAP confirmada, resposta separada (correlação).

Além da troca de mensagens *request-response*, o CoAP prevê duas outras funcionalidades e opções:

Observador. Com esta opção, a mensagem de requisição do cliente pode conter o registro do interesse por um recurso. Qualquer mudança de estado no recurso, por exemplo, pode ser notificada com o envio de uma mensagem. Neste caso, o *token* é usado para identificar o

evento ou origem da notificação. Com isso simula-se o estilo *publish-subscribe*, chamado de *resource-observe* no CoAP [49].

Descoberta de Recursos. O CoAP também padroniza uma forma para um cliente descobrir os recursos disponíveis em um servidor. Para isso, uma mensagem *GET* deve ser enviada com a URL do servidor finalizada com `/.well-known/core`. Neste caso, o servidor retorna os recursos disponíveis no formato CoRE Link [50].

O CoAP oferece outras opções de configuração e interação, além das discutidas nesta seção. Por exemplo, o conceito *block-wise* que facilita a transmissão de informações longas. A RFC 7252, já mencionada, apresenta estas outras opções.

3.4.3. MQTT

O *Message Queuing Telemetry Transport*, MQTT [29], foi concebido por Andy Stanford-Clark (IBM) e Arlen Nipper (Arcom, agora Cirrus Link) em 1999, como um protocolo leve para troca de mensagens entre dispositivos, que tivesse baixo consumo de bateria e pouco uso de banda de rede. Atualmente o MQTT é um padrão ISO/IEC, 20922:2016 e OASIS [51].

Embora o seu nome inclua "enfileiramento de mensagens" o MQTT não é orientado à fila de mensagens como o AMQP e o XMPP. Essencialmente o MQTT é um sistema gerenciador de eventos, que suporta o estilo de interação *publish-subscribe*, e utiliza o TCP para todas as trocas de mensagens entre publicadores e subscritores com o *broker*. A Figura 3.9 apresenta os elementos principais do MQTT.

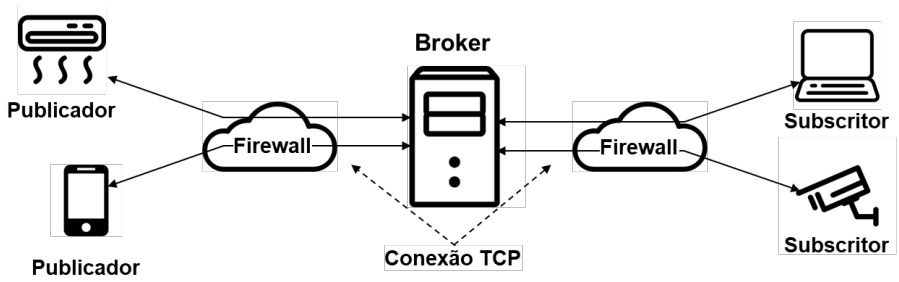


Figura 3.9. Elementos do MQTT.

A arquitetura centralizada do *broker*, facilita a introdução de mecanismos de autenticação e a implementação de características de confiabilidade ou garantia de entrega (chamadas de QoS no MQTT). As credenciais do usuário devem ser enviadas ao se estabelecer uma conexão. Além disso, a comunicação pode fazer uso da camada de segurança TLS/SSL.

No MQTT um tipo de evento é chamado de *tópico*. Um tópico é criado quando uma mensagem *PUBLISH* ou *SUBSCRIBE* é enviada (detalhes adiante). A estratégia para nomear ou referenciar um tópico segue a da formação de nomes de diretórios, que pode ter níveis separados pelo caractere "/" e usar *wildcards*.

Exemplo 1: `minhacasa/terreo/sala/temperatura`. O subscritor recebe o tópico *temperatura* de *sala*.

Exemplo 2: `minhacasa/terreo/$+$/temperatura`. O subscritor recebe o tópico *temperatura* de qualquer "parte" do *terreo*.

Exemplo 3: `minhacasa/terreo/\#`. O subscritor recebe qualquer tópico de qualquer "parte" do *terreo*.

Os clientes MQTT — publicadores e subscritores — e o *broker* comunicam-se através de mensagens de tipos específicos. As principais: *CONNECT*, *SUBSCRIBE*, *UNSUBSCRIBE* e *PUBLISH*, oferecem suporte para a interação no estilo *publish-subscribe*. Existem também mensagens do tipo *ACK* para cada ação, dependendo do nível de QoS selecionado. Para utilizar o MQTT o programador deve enviar e receber mensagens destes tipos, contendo alguns dados inerentes à informação de interesse, como, por exemplo, o tópico, e informações de controle, como, por exemplo, a senha do usuário. Na sequência discutimos as principais.

CONNECT. Enviada pelo cliente quando este deseja conectar-se ao *broker*. Como o MQTT utiliza o TCP, uma conexão é efetivamente estabelecida entre cada cliente e o *broker*. Isso facilita a administração da rede pois, a configuração de *firewall* e NAT precisa apenas ser realizada no nó onde o *broker* executa, uma vez que a conexão TCP permite troca de mensagens bi-direcional. Além disso, a conexão é também reconhecida como sendo o endereço de *callback* para todos os *tópicos* subscritos por cada cliente.

- **clientId**. Identificador único de cada cliente. A aplicação deve gerenciar esta informação.
- **cleanSession**. Especifica se uma conexão vai iniciar uma nova sessão ou vai reestabelecer uma sessão prévia. Se um dado cliente subscrever um tópico e indicar o uso de QoS 1 ou 2, isso fica registrado no *broker*. Caso seja necessário reestabelecer uma conexão e *cleanSession false*, a sessão recupera as subscrições e parâmetros daquele *clientID* e as mensagens possivelmente perdidas são recebidas.
- **username**. Credenciais de autenticação e autorização junto ao broker. Observa-se que o usuário deve registrar-se previamente no *broker*.
- **password**. Credenciais de autenticação e autorização junto ao broker.
- **lastWillTopic**. Quando a conexão for encerrada inesperadamente, o *broker* publicará automaticamente uma mensagem de "último desejo" em um tópico.
- **lastWillMessage**. A própria mensagem de "último desejo"enviada para o *lastWillTopic*.
- **keepAlive**. Intervalo de tempo em que o cliente precisa efetuar *ping* no *broker* para manter a conexão ativa.

SUBSCRIBE. Enviada para informar ao *broker* os tópicos de interesses para recebimento de notificações.

- **tópico**. Referência do tópico para subscrever.

- **qos**. Indica a semântica de entrega e confirmação com que as mensagens neste tópico precisam ser entregues aos clientes, segundo a Tabela 3.1.

Tabela 3.1. Níveis de QoS no MQTT.

Nível	Semântica	Descrição
0	no máximo uma vez (*at most once*)	A mensagem de confirmação não é enviada pelo receptor. É conhecido como "Fire and Forget". Pode ser mais eficiente.
1	pelo menos uma vez (*at least once*)	A mensagem será entregue ao receptor pelo menos uma vez, porém, a tentativa de entrega pode ocorrer mais de uma vez. O publicador precisa guardar a mensagem em um *buffer* local até o recebimento da confirmação pelo receptor.
2	exatamente uma vez (*exactly once*)	A mensagem será recebida, e apenas uma vez. É o nível mais seguro, mas também pode ser menos eficiente. Neste caso, tanto o publicador quanto o receptor, tem a certeza do recebimento da mensagem pelo destinatário.

PUBLISH. Este tipo de mensagem é usado em dois momentos. Primeiro, pelo cliente ao *broker* para a publicação de algum conteúdo em algum tópico. Em seguida, o *broker* realiza as verificações necessárias e retransmite esta mensagens a cada um dos clientes subscritos.

- **topicName**. Referência do tópico no qual a mensagem é publicada. Se o tópico ainda não existe, ele é criado.
- **qos**. Nível de qualidade de serviço da entrega da mensagem. Observe-se que o nível que QoS configurados nas mensagens *PUBLISH* e *SUBSCRIBE* são independentes, pois dizem respeito à comunicação entre o cliente o *broker*.
- **topicMessage**. Informação que dever ser transportada na mensagem. Pode ser uma sequência de caracteres ou um *blob* binário de dados.

Na Seção 3.5.4 um exemplo prático ilustra um ambiente de operação MQTT o uso de biblioteca com API para montar as mensagens do protocolo.

3.4.4. AMQP

O *Advanced Message Queuing Protocol*, AMQP [31], é um sistema de comunicação orientado à filas de mensagens, projetado e desenvolvido pela JPMorgan Chase em 2003. O AMQP é, atualmente, um padrão ISO/IEC (versão 1.0), OASIS. A norma está descrita no documento ISO/IEC 19464:2014 [52].

Como todo sistema orientado a filas, o AMQP tem em sua arquitetura um gerenciador de filas e um roteador de mensagens, além do protocolo de envio, recebimento e enfileiramento de mensagens. Este conjunto de elementos é geralmente chamado de *middleware* orientado a mensagens, MOM (*message oriented middleware*).

Para situar este tipo de infraestrutura, temos a analogia com sistemas de correio eletrônico que utiliza elementos chamados Agentes de Troca de Mensagens (*Message Transfer Agents*, MTA), que poderiam ser baseados no AMQP.

A arquitetura do AMQP versão 0-9-1, utilizada neste capítulo, apresenta os seguintes elementos, coletivamente chamados de *entidades* do *broker* AMQP (Figura 3.10)[4]:

- **Produtores** ou **Publicadores**. Clientes que enviam mensagens a um ou mais destinatários. Ao publicar uma mensagem o produtor pode especificar atributos de mensagens (meta-dados), que podem ser usados pelo *broker*.
- **Consumidores** ou **Subscritores**. Clientes que recebem mensagens de uma fila.
- ***Exchage***. Recebe mensagens dos produtores e as encaminha para uma fila de mensagem. Este encaminhamento pode utilizar políticas parametrizadas por propriedades extraídas da própria mensagem ou de seu conteúdo, ou segundo regras chamadas *Bindings*.
- ***Binding* (ligação)**. Regra que define o relacionamento entre um *Exchage* e uma e Fila de Mensagens, incluindo critérios de roteamento/encaminhamento.
- **Fila de mensagens**. Armazena mensagens até que elas possam ser processadas com segurança por um consumidor ou múltiplos consumidores. Cada fila é criada com um nome, que pode ser usado como chave de roteamento pelo *Exchange*.

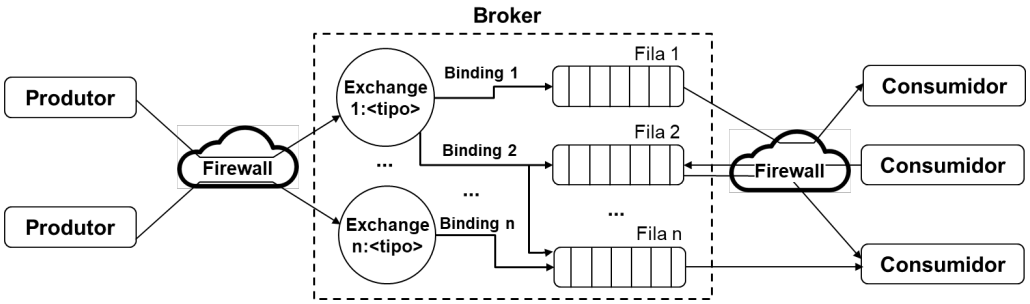


Figura 3.10. Elementos e entidades do *Broker* AMQP.

As mensagens são transportadas de forma binária, e o TCP é utilizado como protocolo de transporte. A camada SSL/TLS pode ser usada para comunicação segura entre os elementos. Para ter acesso ao servidor, aplicações produtoras e consumidoras precisam estabelecer uma conexão e se autenticar no *broker*. A conexão é mantida pela aplicação enquanto houver interesse pelos serviços do AMQP. Esta característica também facilita o acesso ao serviço mesmo que o cliente esteja em redes com NAT e protegida por *firewall*.

O padrão inclui características configuráveis de confiabilidade, segurança e interoperabilidade. O AMQP pode emular o estilo de comunicação *request-response* e também o *publish-subscribe*. Isso é facilitado pelo uso de um elemento central, o *broker* AMQP,

[4]A versão 1.0 apresenta um arquitetura e nomenclatura de elementos um pouco diferentes.

para intermediar a troca de mensagens e, também, pela flexibilidade dos elementos como o *Exchange* e o gerenciamento das Filas de Mensagens. Também é possível oferecer combinações de características não-funcionais como prioridade de filas ou filtros, com o uso de *Bindings*. A confiabilidade da entrega de mensagens, baseada em mensagens de controle *ACK* é um outro ponto positivo do AMQP.

O protocolo para troca de mensagens do AMQP é neutro e aberto, facilitando o envio de qualquer informação e a interoperabilidade entre diferentes plataformas. Além disso, o protocolo é programável, contribuindo para a facilidade de configuração de características oferecidas pelo *broker*. A configuração das entidades, regas e políticas é realizada pela própria aplicação, através de mensagens específicas, e não pelo administrador do servidor. Assim, uma aplicação pode declarar as entidades necessárias, esquemas de roteamento e o comportamento de filas. Uma das configurações possíveis para a aplicação é selecionar o tipo *Exchange* a ser utilizado:

- **Direto, *amq.direct* ou sem nome**, tipo mais simples, padrão para toda fila criada, utilizado para comunicação ponto a ponto ou *unicasting*. Roteia as mensagens segundo a chave de roteamento (na verdade, um nome associado ao *Exchange* e a uma *Fila*).
- **Fanout, *amq.fanout***, utilizado para comunicação *1*-para-*N* ou comunicação *broadcast*. A chave de roteamento é ignorada e uma cópia de mensagem é colocada em todas as filas associadas ao *Exchange*. Este tipo de *Exchange* pode ser usado para aumentar a escalabilidade.
- **Tópico, *amq.topic***, utilizando emular o estilo *publish-subscribe*, ou comunicação *1*-para-*N* ou *N*-para-*N*, ou comunicação *multicasting*. As mensagens são roteadas para uma ou mais filas, com base na chave de roteamento da mensagem e em um padrão usado para associar a *Fila* ao *Exchange*.

Além do tipo, é possível selecionar atributos do *Exchange* como: o nome específico (uma cadeia de caracteres), se serão duráveis ou não duráveis (são restaurados ou não após uma reinicialização do servidor), e se têm metadados associados a eles na criação (detalhes em [53]).

Depois que as mensagens são recebidas pelo *Exchange* e roteadas de acordo com os *Bindings*, estas seguem para suas respectivas filas. As mensagens são armazenadas em uma base de dados interna e gerenciada pelo *broker* para posterior recuperação de clientes interessados. As Filas de Mensagens no AMQP também têm atributos como o nome e durabilidade, pode ser exclusiva ou compartilhada, apagada automaticamente, além de metadados associados na declaração. Selecionando propriedades adequadas, as Filas de Mensagens podem oferecer suporte para a interação entre produtores e consumidores com características específicas:

- **Armazena e encaminha (*Store and forward*)**. Armazena temporariamente as mensagens encaminhadas pelos produtores e as distribui para consumidores registrados de forma circular. Geralmente são criadas para longos períodos e compartilhadas por vários consumidores.

- **Resposta privada.** Recebe as mensagens dos produtores, e as retém até encaminhá-las para um único consumidor. Geralmente são temporárias, com nome atribuído pelo servidor e privativas para um único consumidor.

- **Subscrição privada.** Coleta mensagens de vários produtores, subscritos previamente, e as encaminha para um único consumidor, responsável pelas subscrições.

As possibilidades de configuração e a API do AMQP são amplas. Conforme mencionado, a criação de *Filas*, seleção de *Exchanges* e declaração de *Bindings*, além de várias outras ações, têm suporte da API. O leitor é orientado a consultar as referências para avaliar as opções. Combinações de *Exchanges*, Filas de Mensagens e configurações de roteamento e mensagens de confirmação *ACK* também são possíveis. Na documentação do *broker* RabbitMQ [53], signatário da versão 0-9-1 do AMQP, são apresentadas algumas destas combinações.

Clientes consumidores utilizam a família de métodos *consume* e *deliver* da API do AMQP. Podem também utilizar métodos do tipo *push*. Para usar esta API o consumidor deve se registrar na fila de interesse. Quando uma mensagem for colocada na fila, o *broker* entrega a mesma diretamente para o consumidor. Uma fila pode ter mais de um consumidor interessado. Cada registro ou subscrição tem uma identificação chamada *consumer tag*. Também é possível buscar uma mensagem usando métodos do tipo *pull*. Clientes produtores utilizam a família de métodos *put* e *publish*. Variações de operação para estas primitivas são previstas para que os clientes usufruam das diferentes configurações do *broker*.

A Seção 3.5.5 apresenta uma aplicação simples que ilustra as possibilidades básicas do AMQP e utiliza o RabbitMQ como *broker*.

Para os objetivos deste capítulo, um ponto ainda deve ser discutido. O AMQP, como acontece vários outros MOMs, é flexível e robusto. Para isso, é necessária a execução de muitos elementos de suporte no servidor, e o consumo de recursos computacionais é inevitável. O AMQP pode ser usado de forma federada para oferecer escalabilidade, adicionando mais complexidade ao suporte. Este tipo de de suporte é geralmente necessário em grandes sistemas de informação, orientado à negócios, *enterprise*. O IBM MQ, utilizado em sistemas corporativos e o Java Message Service, JMS, que pode ser integrado a sistemas com suporte do Enterprise Java Beans, são exemplos de MOM.

Então, qual seria a utilidade do AMQP em um contexto de IoT? Provavelmente o AMQP não vai ser utilizado diretamente no acesso à dispositivos com poucos recursos ou com interfaces de comunicação de baixa capacidade, como ocorre com o CoAP ou o MQTT. Bibliotecas AMQP ainda não estão amplamente disponíveis para o Arduino, por exemplo. Por outro lado, a confiabilidade e robustez do AMQP pode ser empregada com vantagens para trazer ou levar requisições, respostas e informações para dispositivos com mais recursos, como *smartphones*, ou ainda para organizar o registro de eventos e demais informações coletadas de sensores em um serviço na nuvem. O cenário da Figura 3.2 ilustra este ponto, e o estudo de caso apresentado na Seção 3.6 apresenta um caso prático.

3.4.5. XMPP

O *Extensible Messanging and Presence Protocol*, XMPP [32] é também um sistema orientado a fila de mensagens. Ele oferece um protocolo para troca de mensagens que usa XML para comunicação, o que inclui mensagens instantâneas, como o AMQP, e também suporte para os conceitos de presença e colaboração.

O XMPP utiliza o TCP como protocolo de transporte e suporta vários estilos de comunicação, sendo os principais o *request-response* e o *publish-subscribe*. A camada SSL/TLS pode ser usada para comunicação segura entre os elementos. Para ter acesso ao servidor, os clientes precisam estabelecer uma conexão e se autenticar.

O projeto original do XMPP era chamado de *Jabber*, com sua primeira versão lançada em 1999. O *Jabber* foi originalmente projetado para uso em aplicativos de mensagens instantâneas, sendo adaptado aos poucos, para utilização em outros cenários. O protocolo original teve seu nome modificado para XMPP. Atualmente, o XMPP tem sido utilizado para atender a diferentes requisitos não funcionais, como funcionamento em navegadores, utilizando extensões WebSocket 3.3, além de aplicações para IoT. É um protocolo aberto e padronizado desde 2011 pela Internet Engineering Task Force [54].

O XMPP é utilizado por aplicativos de Mensagens Instantâneas, *Chat* em Grupo, Jogos, Geolocalização e voz sobre IP. O WhatsApp, por exemplo, utilizava o XMPP em suas primeiras versões. O Google Talk também utiliza o suporte do XMPP.

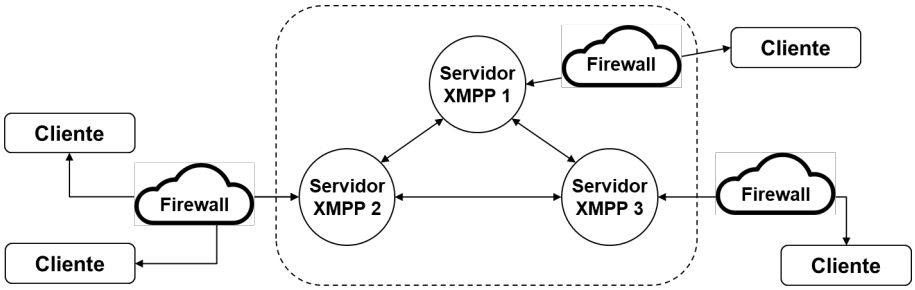


Figura 3.11. Servidores XMPP formando uma federação.

Sendo um sistema de mensagens, o XMPP também requer elementos gerenciadores de filas, chamados servidores de mensagens XMPP. O XMPP pressupõe uma rede federada de servidores, como mediadores no encaminhamento das mensagens. Isso permite que clientes distribuídos em redes protegidos por *firewalls* separados comuniquem-se uns com os outros. Cada servidor controla seu próprio domínio e autentica usuários nesse domínio. Os clientes de um domínio podem se comunicar com clientes em outros domínios através da federação. Os servidores XMPP estabelecem uma malha de conexões, de maneira segura, para trocar mensagens entre seus domínios. A Figura 3.11 ilustra este aspecto.

A autenticação de um cliente é feita usando uma arquitetura baseada em autenticação simples e camada de segurança (SASL) ou TLS/SSL.

A identidade de cada cliente é chamada de endereço XMPP ou *Jabber ID* (JID). Este identificador é semelhante à combinação de um endereço de e-mail e uma URL. Por

exemplo, um usuário com o nome “aluno” executando a aplicação “protocolosIoT” em “lcc.uerj.br” terá o seguinte JID:

`aluno@lcc.uerj.br/protocolosiot`.

Da mesma forma, recursos compartilhados, como salas de discussão, também são representados por JIDs. Por exemplo, “reuniao@lcc.uerj.br” representa uma discussão “reuniao” que está disponível no endereço “lcc.uerj.br”.

A comunicação XMPP consiste de fluxos bidirecionais de fragmentos XML chamados de *stream* e *stanza*, respectivamente. O *stream* é um contêiner para o troca de elementos, ou *stanzas* XML entre duas entidades quaisquer em uma rede. São especificadas três tipos de *stanza*:

- ***Presence***. Utilizado para enviar informações sobre o próprio usuário para outras partes autorizadas e interessadas. Um exemplo é o anúncio para estas entidades sobre sua disponibilidade ou presença (conectado ou desconectado). Este é um diferencial em relação ao modelo de serviços do AMQP.
- ***Message***. Utilizado para enviar mensagens assíncronas (sem aguardar resposta) para um determinado receptor.
- ***IQ* (*info-query*)**. Utilizada para enviar uma requisição que requer uma resposta da outra entidade, interação *request-response*. Existem tipos de *stanza* IQ pré-definidos: *get*, *set*, *result* e *error*. É obrigatório o uso de um campo *id*, para permitir a correlação de requisições e respostas, como ocorre com o *token* do CoAP.

Um fluxo típico de *stanzas* é apresentado na Figura 3.12. Um *stanza* é enviado para um destinatário e este a responde de forma apropriada. O processo de comunicação se dá da seguinte forma: (i) um cliente abre uma conexão com o servidor XMPP e inicia um *stream* XML; (ii) em seguida, existe a negociação de segurança e a autenticação; (iii) *stanzas* do tipo *Message* são, então, trocados e (iv) por último, o *stream* XML e a conexão são fechadas.

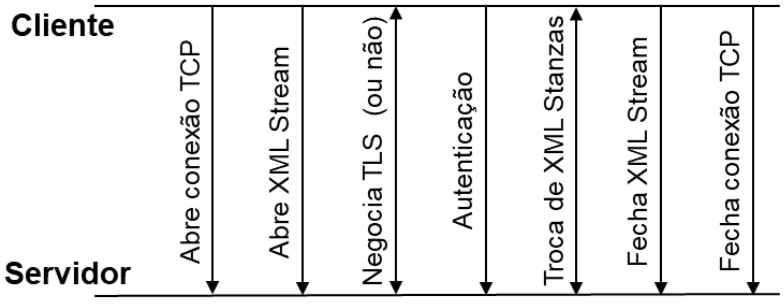


Figura 3.12. Troca de mensagens em um fluxo XMPP.

As bibliotecas de desenvolvimento XMPP, de forma geral, encapsulam a criação e manipulação dos *stanzas* XML, necessários para comunicação entre as entidades. Os *stanzas* gerados são simples. O Código 3.4 apresenta um *stanza* do tipo *Message*, com os elementos necessários.

```
<message xml:lang='en'
  to='aluno@lcc.uerj.br'
  from='alexszt@ime.uerj.br/dicc'
  type='chat'>
  <body>Podemos trabalhar no texto hoje?</body>
</message>
```

Código 3.4. *Stanza* XMPPdo tipo *Message*.

O exemplo no Código 3.5, mostra um *stanza IQ* com uma requisição, e um *stanza IQ* com a resposta retornada pelo servidor XMPP. A requisição solicita ao servidor uma lista de recursos suportados. Esta funcionalidade é útil para o cliente realizar verificações antes de tentar utilizar algum serviço do servidor XMPP.

O *stanza* de requisição tem tipo *get* (linha 5) e precisa de um identificador único (linha 3). O identificador é usado para correlacionar a resposta/confirmação com a requisição. A solicitação específica é descrita em uma *query* qualificada por um *namespace*, neste caso, referenciado por *disco#info*, indicando um pedido de informações ao serviço de descoberta (linha 6). Outros tipos de solicitação e extensões estão disponíveis.

```
1  <!--Requisição-->
2  <iq from="roberto@lcc.uerj.br"
3    id="123456"
4    to="lcc.uerj.br"
5    type="get">
6    <query xmlns="http://jabber.org/protocol/disco#info"/>
7  </iq>
8  <!--Resposta-->
9  <iq from="lcc.uerj.br"
10   id="123456"
11   to="roberto@lcc.uerj.br/6bcfui1lpj"
12   type="result">
13   <query xmlns="http://jabber.org/protocol/disco#info">
14     <identity name="Openfire Server" category="server" type="im"/>
15     <identity category="pubsub" type="pep"/>
16     [...]
17     <feature var="http://jabber.org/protocol/pubsub#subscribe"/>
18   </query>
19 </iq>
```

Código 3.5. *Stanza* XMPP *IQ* dos tipos *get* e *result*.

A resposta é retornada pelo servidor XMPP em um *stanza IQ* do tipo *result* (linha 12), com informações de suas capacidades. O mesmo identificador deve ser usado, linha 10, para permitir que o cliente correlacione a resposta com a requisição, dado que as mensagens em um fluxo XML são independentes. Observamos na resposta que a extensão *publish-subscribe* está disponível no servidor (categoria *pubsub*, linha 15), permitindo ao cliente tomar a decisão de usar este estilo de interação.

O XMPP fornece um protocolo aberto, fácil de usar, flexível e extensível. Isso faz com que a comunidade que utiliza o protocolo, crie uma série de extensões. Existe um fórum chamado XMPP Standards Foundation (XSF), que publica um conjunto de extensões, que são revisadas e discutidas, garantindo a interoperabilidade. Essas extensões são chamadas de XMPP Extension Protocols (XEPs) [55]. O XMPP incentiva e reúne as extensões relacionadas à IoT em `http://www.xmpp-iot.org/`.

O exemplo da Seção 3.5.6 ilustra o uso do XMPP através de chamadas de biblioteca, que tornam mais simples a montagem dos *stanzas*.

3.4.6. Comparação

Para concluir esta Seção, comparamos os protocolos apresentados de forma consolidada e complementamos o conjunto com algumas informações adicionais.

Na Tabela 3.2 reunimos características gerais. Como curiosidade, temos o ano em que cada protocolo foi proposto. Ainda que o uso no contexto de IoT seja recente, as propostas não são tão novas. Todas os protocolos estão atualmente padronizados por algum órgão ou entidade ligada à computação, telecomunicações ou sistemas distribuídos. Também listamos os principais sistemas de suporte para os protocolos. Na seção seguinte utilizaremos alguns deles para construir os exemplos. Destaque para o RabbitMQ, que oferece suporte ao MQTT e ao AMQP. Na última coluna consolidamos os principais estilos de interação suportados por cada protocolo.

Tabela 3.2. Características Gerais.

Protocolo	Ano	Padrão	Bibliotecas	Interação
HTTP	1997	IETF, W3C	Disponíveis para Java, CSharp, Javascript e outras	request-response
CoAP	2010	IETF, Eclipse Foundation	Esp-CoAP, Californium	request-response
MQTT	1999	OASIS, Eclipse Foundation	PubSubClient, RabbitMQ, Eclipse MQTT JS	publish-subscribe
AMQP	2003	OASIS, ISO/IEC	RabbitMQ	request-response, publish-subscribe
XMPP	1997	IETF, W3C	Jaxmpp2, OpenFire, Outras	request-response, publish-subscribe

A Tabela 3.3 reune algumas características ligadas à comunicação. Observa-se que o único protocolo que não utiliza o TCP é o CoAP. O CoAP foi projetado para ter pouco *overhead*, ou seja o menor impacto possível na comunicação. Entretanto por utilizar o UDP, é necessário um esforço maior para a configuração de *firewall* e *NAT*, por isso, a indicação direta do seu uso é em redes locais. Listamos, para rápida referência, as portas usadas por padrão em cada protocolo.

A coluna *Confiabilidade* indica como cada protocolo trata o requisito não-funcional de garantia de entrega das mensagens. Em alguns casos este requisito é tratado como QoS

(o MQTT coloca explicitamente desta forma). Preferimos usar o termo *confiabilidade*, pois o termo *QoS* é geralmente usado para várias características não-funcionais ligadas à garantias de tempo de resposta, latência, prioridade e variação de atraso. O AMQP, por exemplo, pode oferecer tratamento prioritário para filas específicas — e isso poderia ser considerado suporte à QoS. Mas, isso não está incluído na coluna. A confiabilidade de entrega é apoiada em dois elementos: o uso do TCP, que tem características de garantia de entrega, controle de erro e controle de congestionamento, por exemplo, e na transmissão de mensagens de confirmação (tipo *ACK*), que devem ser verificadas pela aplicação.

Tabela 3.3. Características de Comunicação.

Protocolo	Transp.	Porta	Confiabilidade	Autent.	Segur.
HTTP	TCP	80, 443 (TLS/SSL)	Confb. TCP	Sim	TLS/SSL
CoAP	UDP	5683, 5684 (DTLS)	Confirmado, Non-confirmable	Não	DTLS
MQTT	TCP [5]	1883, 8883 (TLS/SSL)	QoS 0 - At most once, QoS 1 - At least once, QoS 2 - Exactly once	Sim	TLS/SSL
AMQP	TCP	5672, 5671 (TLS/SSL)	Confb. TCP, mensagens ACK (versão 0.9.1)	Sim	TLS/SSL
XMPP	TCP	5222 (cliente), 5269 (servidor), 5223 (TLS/SSL)	Confb. TCP, mensagens ACK	Sim	TLS/SSL

Por último, características relacionadas ao controle de acesso e à privacidade. Com exceção do CoAP, os protocolos oferecem alguma forma de autenticação, restringindo o acesso à usuários (ou aplicações) que são autenticados. Os protocolos baseados em um elemento central como o *broker* permitem que serviços de terceiros para autenticação sejam utilizados. Para privacidade, opções de criptografia das mensagens são utilizadas.

3.5. Suporte, Bibliotecas e Exemplos

Nesta seção é apresentado um exemplo prático para cada protocolo discutido na Seção 3.4, considerando também o mecanismo WebSockets apresentado na Seção 3.3. A Figura 3.13 ajuda a entender o cenário usado como base. Nos exemplos, exploramos os usos mais indicados de cada protocolo, *request-response* ou *publish-subscribe*. Um dispositivo NodeMCU é utilizado na maioria dos exemplos [56]. O ambiente de programação é compatível com o Arduino.

Nosso NodeMCU hora atua como "servidor", que pode ligar ou desligar os LEDs vermelho e verde, e hora como publicador de eventos, que periodicamente publica informações. Um dispositivo Java/JavaScript, que pode ser um *smartphone* Android ou um nó Linux (ou Windows) que atuam como clientes e servidores completam o conjunto de elementos.

[5] UDP com MQTT-SN

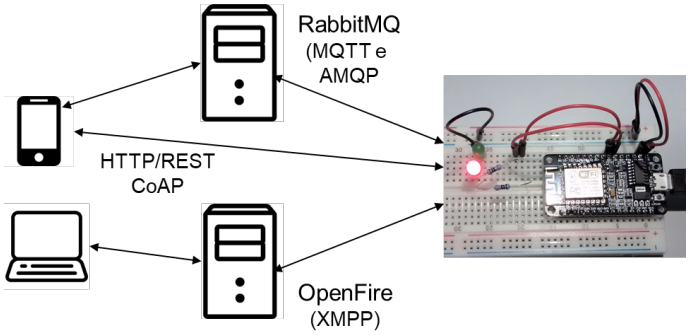


Figura 3.13. Cenário dos exemplos.

Java e o "dialeto" C para Arduino são utilizadas como linguagens de programação no desenvolvimento dos exemplos. Nos exemplos será possível verificar que, devido à limitações do Arduino, são necessárias estruturas de controle para os protocolos, que são opcionais, ou mais flexíveis, em outros ambientes. Por exemplo, é recomendando, por precaução, inserir no *loop* do Arduino instruções para reconexão. Para interação via MQTT sobre WebSockets, a linguagem de programação JavaScript é utilizada.

As ferramentas de desenvolvimento utilizados são comuns à vários ambientes de projeto: Android Studio para aplicações Android, NetBeans para desenvolvimento Java e JavaScript e ArduinoIDE para o desenvolvimento de aplicações para o NodeMCU. Entretanto, todos códigos apresentados nas próximas subseções podem ser editados em qualquer IDE compatível com a linguagem utilizada.

Completando o exemplo de cada protocolo, a biblioteca ou sistema de suporte utilizados são também apresentados. Antecipando, o *broker* RabbitMQ é usado como suporte para o MQTT e para o AMQP. Ele executa em um dispositivo RaspberryPi. Não existe razão específica para esta configuração além de mostrar que é possível fazer isso. O servidor OpenFire, que suporta o XMPP executa em um nó Windows 10. Outras bibliotecas e serviços de suporte são listados complementarmente no Apêndice 3.8.

3.5.1. WebSockets

O exemplo com o WebSocket mostra como esta tecnologia pode ser utilizada por navegadores, em uma comunicação bidirecional, com outros servidores ou clientes. Os estilos de interação apresentados anteriormente, bem como os Protocolos de Aplicação que utilizam TCP, podem utilizar-se do WebSocket como um mecanismo de transporte, similar ao uso do *socket* tradicional.

O exemplo apresentado possui um cliente que envia ao servidor um relatório a cada X unidades de tempo. Além disso, o cliente está preparado para receber, a qualquer momento, uma solicitação do servidor para que esse intervalo X seja alterado. O servidor é preparado para tratar a informação enviada pelo cliente e, a qualquer momento, através de uma mensagem, pode alterar a frequência com que essa informação é enviada.

Para a implementação deste exemplo foi utilizado um NodeMCU como o cliente, que gera relatórios periodicamente. Para implementar o servidor WebSocket foi utilizado a linguagem de programação Java.

Este cenário poderia ser utilizado em um ambiente de monitoramento, por exemplo, onde existem diversos sensores gerando relatórios para um único servidor. É possível ainda que esses papéis sejam invertidos, como por exemplo, em um cenário de Cidade Inteligente esses sensores poderiam ser disponibilizados como *beacons*, fazendo com que o nodeMCU seja o servidor WebSocket, e cada cliente interessado nas informações deveria estabelecer a comunicação.

O Código 3.6 apresenta o servidor Java desenvolvido para o cenário descrito acima. Utilizamos a classe *Server*, da biblioteca *Tyrus* que provê uma infraestrutura geral de um servidor WebSocket. Entretanto, com pequenas modificações é possível integrar este mesmo código com ambientes Web estruturados como o Apache Web Server [46] ou o Glassfish [57].

```
1 @ServerEndpoint(value = "/")
2 public class WebSocketServer {
3   public static void main(String[] args) {
4      Server server = new Server(
5         "0.0.0.0", 8080, "/", null,
6               WebSocketServer.class);
7      try {
8        server.start();
9        while(true){
10         Thread.sleep(10000);
11         for(Session s:list){
12           if(s.isOpen())
13           s.getBasicRemote().sendText(getJson());
14     } } }
15     catch (Exception e) { e.printStackTrace();}
16     finally { server.stop();}
17   }
18   @OnOpen
19   public void onOpen(Session session) throws IOException {
20       list.add(session);
21   }
22   @OnClose
23   public void onClose(Session session) throws IOException {
24       list.remove(session);
25   }
26   @OnMessage
27   public void onMessage(String message, Session sessao){
28     usaInformacao(message);
29 }}
```

Código 3.6. Servidor WebSocket escrito em Java.

O programa começa criando uma variável referente ao servidor WebSocket, indicando também as configurações desejadas. Neste exemplo o serviço deve ser disponibilizado para todas as interfaces disponíveis (0.0.0.0) na porta 8080. Além disso, é indicado também que a própria classe `WebSocketServer` irá responder aos eventos da conexão (linhas 4 a 6). Só então o servidor é iniciado na linha 8.

Conforme discutido na apresentação deste cenário, o servidor pode eventualmente enviar mensagens para os clientes para que eles alterem a periodicidade do envio de relatórios. Para fins de demonstração, o código apresentado envia estas mensagens a cada 10s para todos os clientes atualmente conectados (linha 11). Então na linha 13 é preparado um *JSON* e enviado como texto para os clientes. Para isso, uma lista concorrente com todos os clientes é mantida.

A biblioteca Tyrus [58], utilizada na implementação do servidor, permite que, através de anotações no código, determinados métodos funcionem como *callbacks* para eventos do WebSocket. Neste caso, a linha 20 será executada sempre que um cliente se conecta ao servidor pela primeira vez, então ele é adicionado à lista de clientes atual. De forma análoga a linha 16 é executada sempre que um cliente se desconecta. Por último, a linha 28 é executada sempre que o servidor recebe uma mensagem do cliente.

```
1 #include <WebSocketsClient.h>
2 WebSocketsClient webSocket;
3 void setup(){
4   configuraWifi(ssid, senha);
5   webSocket.begin(IP, PORTA, "/");
6   webSocket.onEvent(webSocketEvent);
7   webSocket.setReconnectInterval(5000);
8 }
9 void loop() {
10   webSocket.loop();
11   webSocket.sendTXT(getReport());
12   delay(intervalo);
13 }
14 void webSocketEvent(WStype_t type, uint8_t * payload, size_t length) {
15   switch(type){
16   case WStype_CONNECTED:
17     webSocket.sendTXT(thisDeviceInfo());
18     break;
19   case WStype_TEXT:
20     Json msg = Json.parseObject(toString(payload,length));
21     if(msg.success())
22       intervalo = msg["intervalo"];
23     break;
24 }}
```

Código 3.7. Código para o nodeMCU operando como cliente WebSocket.

O código 3.7 apresenta o programa do cliente, utilizando a biblioteca Arduino WebSocket [59]. Inicialmente, são configurados os elementos específicos, como por exemplo o Wi-Fi, sensores, atuadores, etc. Na linha 5 é iniciado o cliente WebSocket que deve se conectar ao *IP*, *PORTA* e caminho do servidor. Então na linha 6 é registrada a função responsável por responder aos eventos da conexão. A biblioteca, permite que outros parâmetros da conexão sejam editados, como por exemplo um tempo para se restabelecer a conexão (linha 7) automaticamente.

Com todas as configurações realizadas, o cliente passa a executar a função *loop*,

que realiza tarefas específicas da biblioteca (linha 10), gera um relatório e envia para o servidor (linha 11) e então, dorme por um determinado intervalo de tempo (linha 12).

Por ultimo, é definida a função que trata os eventos gerados pela conexão. Para manter a simplicidade do exemplo iremos apresentar os eventos: `WStype_CONNECTED` e `WStype_TEXT`, que representam, respectivamente, a confirmação de uma nova conexão estabelecida e o recebimento de uma mensagem de texto. A linha 17 é executada no momento que a conexão é estabelecida, neste caso, o dispositivo irá enviar um texto contendo informações sobre o próprio dispositivo para o servidor. Quando o dispositivo recebe uma mensagem de texto do servidor ele tenta interpretar como um JSON (linha 20) e então verifica se a mensagem é válida (linha 21). Caso positivo, o valor recebido será atribuído à variável intervalo (linha 22).

3.5.2. HTTP/REST

Como primeiro exemplo utilizando o estilo *request-response*, será usado o protocolo HTTP sobre REST. No cenário apresentado, um nodeMCU tem o papel de servidor e permite que um LED seja controlado através de um serviço REST.

O Código 3.8 apresenta de forma simplificada este servidor, enquanto o Código 3.9 apresenta um cliente implementado em Java.

Seguindo o Código 3.8, na linha 4 é configurado o servidor HTTP na porta 80. A escolha da porta é arbitrária, mas é comum que a porta 80 seja utilizada para aplicações HTTP. Em seguida, são realizadas as configurações específicas da aplicação para o Wi-Fi e para os LEDs.

Nas linhas 9 e 8 os serviços REST são configurados: o caminho — seguindo o exemplo apresentado na Seção 3.4.1, é utilizado `/servico/LEDService/vermelho` —, e qual função será responsável por tratar as requisições, de acordo com o método HTTP utilizado (*GET* ou *PUT*). Na linha 10 o servidor RESTful é, então, inicializado e fica disponível.

A função de *loop* (linhas 12 a 14) contém uma chamada específica para um procedimento da biblioteca utilizada, `ESP8266WebServer` [60]. Não será detalhada aqui.

A função responsável por tratar as requisições recebidas por mensagens HTTP com método *PUT* foi implementada de forma a utilizar um estrutura `if-else` para dividir as possíveis solicitações. A primeira parte verifica se a requisição é válida e contém todos os cabeçalhos necessários (linha 16). Se for reconhecido algum erro o servidor responde com um código 400 (*Bad Request*) e com um HTML de erro pré-definido (linha 17). Caso contrário, é verificado se a requisição é um pedido para desligar o LED vermelho (linha 21), o LED é desligado (linha 22) e então envia para o cliente um HTML específico (linha 23). Essa estrutura se repete de forma análoga para o caso de uma solicitação para ligar o LED vermelho (linhas 24, 25 e 26). Por último, é tratado as solicitações que possuem os cabeçalhos necessários, mas o conteúdo inválido, neste caso é enviando um HTML de erro (linha 28) com o mesmo código 400.

```
1 #include <ESP8266WiFi.h>
2 #include <ESP8266WebServer.h>
3 #define ledVermelho 13
4 ESP8266WebServer server(80);
5 void setup() {
6   configuraWifi(ssid, senha);
7   configuraLED();
8   server.on("/servicos/LEDServer/vermelho", HTTP_PUT, handlerPut);
9   server.on("/servicos/LEDServer/vermelho", HTTP_GET, handlerGet);
10  server.begin();
11 }
12 void loop() {
13  server.handleClient();
14 }
15 void handlerPut(){
16  if(!server.hasHeader("led")){
17    server.send(400, "text/plain", HTMLPageErro);
18    return;
19  }
20  String valor = String(server.header("led"));
21  if(valor == "0"){
22      digitalWrite(led_vermelho, 0);
23      server.send(200, "text/plain", HTMLPageLedDesligado);
24  }else if(valor == "1"){
25      digitalWrite(led_vermelho, 1);
26      server.send(200, "text/plain", HTMLPageLedLigado);
27  }else{
28      server.send(400, "text/plain", HTMLPageErro);
29 }}
30 void handlerGet(){
31  server.send(200, "text/plain", HTMLPageDefault);
32 }
```

Código 3.8. NodeMCU operando como servidor HTTP.

A função que responde as requisições *GET* e envia um HTML padrão com o código 200 (*OK*) é definida na linha 31.

Observa-se a facilidade para montar mensagens HTTP através de API utilizada, bastando chamar o método *send* e alguns parâmetros.

O Código 3.9 mostra o programa cliente, acessando o servidor e realizando uma solicitação para ligar o LED. Na linha 4 a URL para o recurso é montada e passada para a função que irá realizar a requisição. A linha 7 utiliza essa informação para criar um objeto que será utilizado para estabelecer a conexão com o servidor (linha 8). Com a conexão estabelecida o cliente pode receber dados através de um `InputStream` ou enviar dados através de um `OutputStream`.

Neste exemplo estamos interessados em enviar uma informação para o servidor através dos cabeçalhos do HTTP, então iremos utilizar o método PUT (linha 9) e adicionar o cabeçalho `led` com o valor 1 (linha 10). Para simplificar iremos armazenar a informação recebida em uma `String`(linha 11). A conexão é finalizada pelo cliente e por

último a resposta do servidor é utilizada pelo cliente, neste caso, ele simplesmente exibe a mensagem (linha 13).

```
1 public class HTTPClient {
2 public static void main(String[] args) throws Exception {
3   HTTPClient http = new HTTPClient();
4   http.sendPut("http://domínio/servicos/LEDServer/vermelho");
5 }
6 public void sendPut(String url) throws Exception {
7   URL obj = new URL(url);
8   HttpURLConnection con = (HttpURLConnection) obj.openConnection();
9   con.setRequestMethod("PUT");
10  con.setRequestProperty("led","1");
11  String response = con.getResponseCode();
12  con.disconnect();
13  System.out.println(response);
14 }}
```

Código 3.9. Cliente HTTP/REST escrito em Java.

É importante observar que dadas as características do HTTP/REST o cliente poderia ser simplesmente um navegador acessando este serviço através de uma URL. O Código 3.10 apresenta uma requisição realizada por um navegador e a resposta do servidor nodeMCU.

```
1 PUT /servicos/LEDServer/vermelho HTTP/1.1
2 Host: teste.lcc.uerj.br
3 Connection: keep-alive
4 Content-Length: 0
5 User-Agent: Mozilla/5.0 (X11; Linux x86_64)
6         AppleWebKit/537.36 (KHTML, like Gecko)
7         Chrome/65.0.3325.181 Safari/537.36
8 led: 1
9 Content-type: application/json; charset=UTF-8
10 Accept: */*
11 Accept-Encoding: gzip, deflate
12 Accept-Language: pt-BR,pt;q=0.9,en-US;q=0.8,en;q=0.7

13 HTTP/1.1 200 OK
14 Content-Type: text/plain
15 Content-Length: 22
16 Connection: close
```

Código 3.10. Mensagens de requisição e resposta HTTP com método *PUT*.

Para exemplificar iremos detalhar os cabeçalhos envolvidos na requisição *PUT*, que tem como objetivo alterar o estado do recurso, neste caso ligar o LED. A linha 1 apresenta o método HTTP utilizado (*PUT*), o caminho do recurso (`/servicos/LED/vermelho`)

e qual a versão do HTTP utilizada (1.1). Em seguida, na linha 2, é definido o *Host* para qual essa requisição será entregue. Só então os diversos cabeçalhos utilizados são definidos, entre eles a informação específica desta aplicação, identificada pela chave `led` (linha 8).

A resposta do servidor começa na linha 13 e contem a versão do HTTP utilizada (1.1) e o código de resposta (200). Além disso, é informado também qual o tipo (linha 14) e o tamanho (linha 15) da resposta. Por último, o servidor informa que irá fechar a conexão (linha 16).

3.5.3. CoAP

Para o exemplo do CoAP, será reusado o mesmo cenário do exemplo HTTP: o dispositivo NodeMCU será utilizado como servidor, e o cliente desenvolvido em Java tem como objetivo desligar o LED. Foi utilizada a biblioteca Esp-CoAP [61] para o servidor e para o cliente a biblioteca Californium [62]. Estas mesmas bibliotecas permitem que o NodeMCU seja utilizado como cliente e um dispositivo capaz de executar programas Java como servidor.

O Código 3.11 apresenta o servidor CoAP desenvolvido Nas primeira linhas são realizadas tarefas específicas, como a importação de bibliotecas, configurações do Wi-Fi e a configuração do LED. Na linha 5 é indicada qual função será responsável por responder às requisições e qual o caminho utilizado. Na linha 6 o servidor é, então, iniciado.

```
1 #include <coap_server.h>
2 void setup() {
3   configuraWifi(ssid, senha);
4   configuraLED();
5   coap.server(callbackLed, "ledVermelho");
6   coap.start();
7 }
8 void loop() {
9   coap.loop();
10  delay(500);
11 }
12 void callbackLed(coapPacket &pacote, IPAddress ip, int port, int obs) {
13   String message = char2string(pacote->payload);
14   if (message.equals("0"))    {
15     digitalWrite(led_vermelho, 0);
16     coap.sendResponse(ip,port,RespostaDesligado);
17   }else if (message.equals("1")){
18     digitalWrite(led_vermelho, 1);
19     coap.sendResponse(ip,port,RespostaLigado);
20   }else{
21     coap.sendResponse(ip,port,RespostaDefault);
22 }}
```

Código 3.11. NodeMCU operando como servidor CoAP.

A função *loop* deverá ser modificada de acordo com a biblioteca escolhida, no caso da Esp-CoAP o desenvolvedor deve chamar a biblioteca a cada 500*ms* (linha 9).

Por último a função `callbackLed` é executada sempre que o servidor recebe uma requisição no caminho declarado anteriormente. Na linha 13 o conteúdo da requisição é convertido para *String* a fim de facilitar o tratamento da mensagem. Neste ponto poderiam ser realizadas outras etapas como validação da mensagem, conversões de formato(JSON, XML, etc) ou até mesmo uma verificação de autenticidade através de uma assinatura digital, por exemplo.

De forma similar ao exemplo da Seção 3.5.2, esta função utilizou uma estrutura `if-else` para tratar as requisições. A primeira possibilidade trata uma requisição para desligar o LED, então é verificado o código recebido (linha 14), alterado o estado do LED (linha 15) e então enviado uma resposta para o cliente (linha 16). De maneira análoga as linhas 17, 18 e 19 tratam uma requisição para ligar o LED. Caso o código recebido não seja reconhecido, uma resposta padrão é enviada para o cliente (linha 21).

O Código 3.12 apresenta o cliente CoAP, desenvolvido em Java, que realiza uma solicitação para desligar o LED. Utilizando a URL do servidor, o cliente cria um objeto para intermediar a comunicação (linha 2). Na linha 3 o cliente realiza o procedimento para abrir uma determinada porta para receber a resposta do servidor e caso não aconteça nenhum erro, na linha 9 essa porta é atribuída a conexão.

```
1 public void coapRequest(){
2   CoapClient client = new CoapClient("coap://domínio/ledVermelho");
3   CoapEndpoint endpoint = new CoapEndpoint(PORTA);
4   try{
5     endpoint.start();
6   }catch(Exception ex){
7     trataExcecao(ex);
8   }
9   client.setEndpoint(endpoint);
10  Request request = new Request(CoAP.Code.PUT);
11  request.setPayload("0");
12  request.setType(CoAP.Type.CON);
13  CoapResponse response = client.advanced(request);
14  if (response != null){
15    System.out.println(response.getCode());
16    System.out.println(response.getOptions());
17    System.out.println(response.getResponseText());
18  }else {
19    System.out.println("Request failed");
20 }}
```

Código 3.12. Cliente CoAP escrito em Java.

Com os parâmetros da conexão definidos o cliente monta uma requisição *PUT* (linha 10) com os dados necessários para realizar a desativação do LED (linha 11) e a mensagem é configurada para o tipo *Confirmado*. Só então essa requisição é realizada. Como a requisição é *PUT* e o tipo *CON* foi configurado, uma resposta é aguardada de forma síncrona (linha 13). Após uma verificação da resposta (linha 14) o cliente pode informar ao usuário a resposta ou eventuais erros.

3.5.4. MQTT e WebSockets

No exemplo com MQTT, o módulo NodeMCU faz o papel de um cliente subscritor, através da biblioteca PubSubClient [63]. Um navegador foi usado como subscritor e como publicador, com o objetivo de mostrar o uso do MQTT com o WebSocket como mecanismo de transporte. Neste caso, a biblioteca utilizada foi a Eclipse Paho para Javascript [64]. O RabbitMQ [53], usado como *broker* neste exemplo, foi implantado em um RaspberryPi e disponibilizado para a conexão de clientes através do *número IP* e *porta*.

Neste cenário, o navegador no papel do publicador, pode enviar mensagens para um tópico do *broker* e este repassa esta informações para todos os subscritores interessados. Neste momento, tanto o módulo NodeMCU como o navegador, devem tratar a mensagem recebida. Para este exemplo, será utilizado o tópico `topic/led/vermelho` para controlar um LED no nodeMCU.

O Código 3.13 apresenta o programa em linguagem C para o nodeMCU como subscritor MQTT. As primeiras linhas do programa realizam algumas configurações específicas desse exemplo, como importar a biblioteca, configurar o Wi-Fi, etc. Na linha 6 é configurada a URL e a porta do *broker* ao qual o cliente deve se conectar. Já na linha 7 é indicada qual a função que responderá aos eventos de chegada de mensagens.

```
1 #include <PubSubClient.h>
2 PubSubClient MQTT(conexaoObj);
3 void setup() {
4   configuraWifi(ssid, senha);
5   configuraLED();
6   MQTT.setServer(URL, PORTA);
7   MQTT.setCallback(mqttCallback);
8 }
9 void loop() {
10   while (!MQTT.connected()){
11     if (MQTT.connect(ID,Usuario,Senha)){
12       MQTT.subscribe("topic/led/vermelho");
13     }else{
14       delay(500);
15   }}
16   MQTT.loop();
17 }
18 void mqttCallback(char* topic, byte* payload, unsigned int length) {
19   String message = char2string(payload);
20   if (message == "vermelhoon") {
21     digitalWrite(ledVermelho, 1);
22   else
23     digitalWrite(ledVermelho,0);
24 }}
```

Código 3.13. NodeMCU como subscritor MQTT.

De maneira similar aos exemplos anteriores que utilizaram o nodeMCU, a função `loop` contém as chamadas para a biblioteca utilizada, que precisam executar periodica-

mente. Verifica-se, a cada iteração, se a conexão foi perdida (linha 10) e, se necessário, ela é reestabelecida (linha 11). Na linha 12, o nodeMCU, então, subscreve o tópico `topic/led/vermelho`.

Quando uma mensagem é recebida do *broker*, *mqttCallback* é chamada. Primeiramente, a mensagem é tratada e convertida para `String` (linha 19). Observe-se que, de forma similar ao discutido na Seção 3.5.3, a rotina de tratamento de mensagem pode ser utilizado para outras tarefas além de uma simples conversão de formato. Após essa etapa, o conteúdo da mensagem é verificado (linha 20) e, então, o LED pode ser ligado ou desligado (linhas 21 e 23).

O Código 3.14 apresenta o publicador e subscritor MQTT desenvolvido em JavaScript para ser executado em navegadores. O código faz parte de uma página HTML, carregada pelo navegador, que iria prover campos para o usuário entrar com o texto da mensagem a ser publicada. Também seria provido um campo para visualização das mensagens recebidas, publicadas por outros clientes naquele determinado tópico.

```
1 var client = new Paho.MQTT.Client(URL,PORTA,ID);
2 client.onConnectionLost = function (responseObject) {
3   reconectar();
4 };
5 client.onMessageArrived = function (message) {
6   print(message.payloadString);
7 };
8 var options = {
9   timeout: 3,
10  userName: Usuario,
11  password: Senha,
12  onSuccess: function () {
13    client.subscribe('topic/led/vermelho', {qos: 1});},
14  onFailure: function (message) {
15    erro(message);},
16  useSSL: true
17  };
18 client.connect(options);
19 var send = function(data) {
20    message = new Paho.MQTT.Message(data);
21    message.destinationName = "topic/led/vermelho";
22    client.send(message);
23 };
24 form.submit(function() {
25                     send(input.val());
26                     input.val('');});
```

Código 3.14. Cliente JavaScript Publicador-Subscritor MQTT utilizando WebSocket.

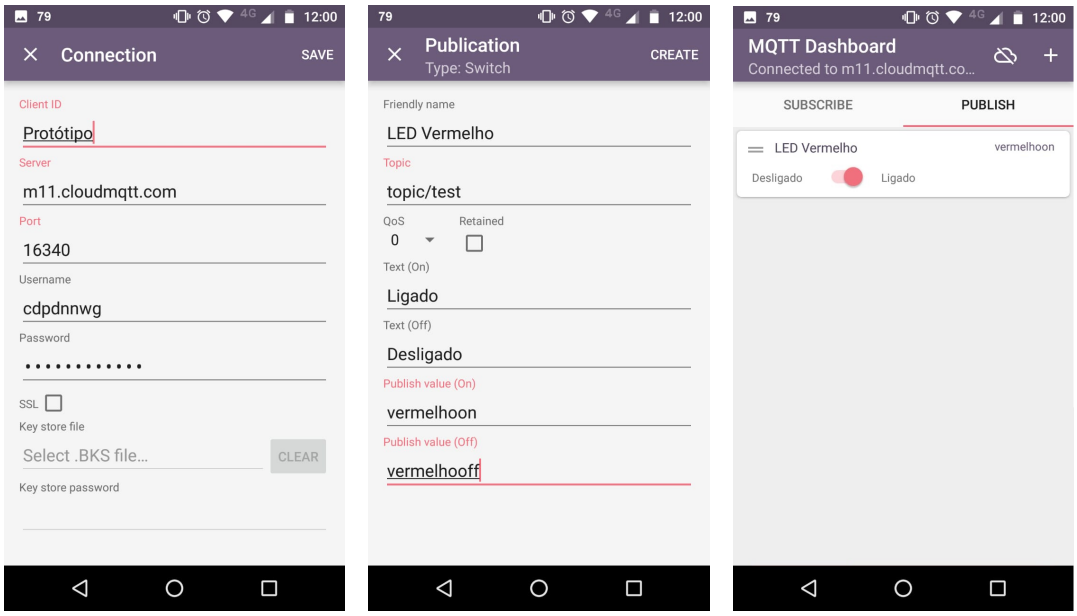
Na linha 1 é configurado a URL e a porta do *broker* e o ID do cliente para essa conexão. Este ID é uma *String* utilizada pelo *broker* para identificar de forma não-ambígua os clientes conectados. Em seguida na linha 3 é incluído um *callback* para a perda de conexão. De forma análoga, na linha 6 é determinado o comportamento do programa ao

receber uma mensagem, neste caso ele ira exibir a informação para o usuário.

Antes da conexão ser estabelecida mais alguns parâmetros são definidos (linha 8), como por exemplo o nome e senha do usuário (linhas 10 e 11), em quais tópicos está interessado como subscritor e seus respectivos níveis de QoS (linha 13), bem como o tratamento de erro 15. Com todos os parâmetros definidos, essas informações são passadas para o cliente criado e a conexão é estabelecida na linha 18.

Até então nosso cliente seria apenas um subscritor. Porém, na linha 19 é definida a função que trata o envio de mensagens para o *broker*. Nesta função é especificado o tópico onde as mensagens serão publicadas (linha 21) e, então, é realizado o envio das informações (linha 22). O conteúdo das mensagens é digitado pelo usuário em um campo de texto de um formulário HTML, que são passadas para a função detalhada acima (linha 24).

Existem ferramentas úteis para prototipação e testes com o MQTT. Estas ferramentas exploram a simplicidade da API e dos tipos de mensagens do protocolo para oferecer ao programador uma forma prática para testar a interação com o *broker*, mesmo sem programas desenvolvidos. Incluímos aqui um exemplo de teste utilizando a ferramenta IoT MQTT Dashboard [65]. Na Figura 3.14, temos a sequência de configurações do servidor (Figura 3.14(a)) e do tópico (Figura 3.14(b)). Em seguida, um teste de publicação no tópico (Figura 3.14(c)).



(a) Configuração do servidor. (b) Configuração do tópico. (c) Publish (on-off).

Figura 3.14. Ferramentas de prototipação e testes para MQTT.

3.5.5. AMQP

Para demonstrar o uso do AMQP, foram desenvolvidos dois clientes: um publicador e um subscritor. Ambos foram desenvolvidos na linguagem de programação Java, fazendo uso do *broker* e da API do RabbitMQ [53]. O suporte do RabbitMQ ao MQTT e ao AMQP facilita a interoperabilidade, como será demonstrado na Seção 3.6, e certamente

facilitou a implementação do exemplo.

Para este exemplo, o Código 3.16 apresenta um publicador que envia uma mensagem de "Hello World"para o *broker*, que então a repassa para os respectivos interessados, neste caso o cliente apresentado no Código 3.17.

Independente do papel do cliente AMQP, publicador ou subscritor, ele primeiro deve se conectar ao *broker*. Como o estabelecimento da conexão independe do papel do cliente, a rotina foi apresentada separadamente no Código 3.15. Na linha 3 é criada uma fábrica de conexões e com ela diversos parâmetros podem ser definidos. Para manter a simplicidade do exemplo, definimos apenas o IP do *broker* (linha 4) e então estabelecemos a conexão (linha 5). Para este cenários os dois clientes vão se conectar na mesma fila, para isso eles devem criar um canal (linha 6), o que permite que a mesma conexão TCP estabelecida com o *broker* seja reutilizada em diversos momentos. Só então, o cliente declara uma fila no *broker* (linha 7).

Por padrão a biblioteca irá criar a fila, caso ela não exista, ou então retornar a referência para a fila existente. Além disso, esta fila criada será, por padrão, "não-exclusiva", não será descartada automaticamente quando não houver mais clientes conectados e não será persistida caso o servidor tenha que ser reiniciado. É importante notar que a biblioteca utilizada oferece esta mesma função com mais parâmetros para que o programador possa customizar a fila de acordo com sua necessidade.

```
1 import com.rabbitmq.client.*;
2 /.../
3 ConnectionFactory fabrica = new ConnectionFactory();
4 fabrica.setHost(IP);
5 Connection conexao = fabrica.newConnection();
6 Channel canal = conexao.createChannel();
7 canal.queueDeclare(FILA);
```

Código 3.15. Cliente AMQP realizando uma conexão com o broker, escrito em Java.

Caso o cliente queira exercer o papel de publicador ele também deve utilizar o Código 3.16. Para isso ele deve apenas criar uma mensagem (linha 1) e então utilizar o canal para publicar (linha 2).

```
1 String mensagem = "Hello World!";
2 canal.basicPublish("", FILA, null, mensagem.getBytes());
```

Código 3.16. Cliente AMQP no papel de Publicador, escrito em Java.

No momento da publicação podem ser definidas diversas propriedades para aquela mensagem. No exemplo apresentado foi utilizado o *Exchange* do tipo direto, representado pela *String* vazia no primeiro parâmetro da função, e não foram criadas propriedades personalizadas, representadas por *NULL* no terceiro parâmetro da função. Essas propriedades

poderiam ser utilizadas, por exemplo, para determinar o tipo da mensagem que esta sendo enviada, uma data de validade, cabeçalhos ou prioridade (Seção 3.4.4).

Caso o cliente esteja interessado nas mensagens de uma fila ele deve utilizar o Código 3.17. Para isso, é preciso declarar um consumidor (linha 1) que irá tratar cada mensagem. Neste caso do exemplo, a informação é exibida (linha 8). Neste ponto, o cliente poderia recuperar todas as propriedades personalizadas definidas anteriormente pelo publicador através do parâmetro `properties` recebido pela função. Uma vez criado o consumidor, este deve ser registrado no canal com a informação da fila (linha 11). No exemplo, o cliente definiu que a biblioteca deve tratar o envio das mensagens de confirmação(*ACK*) através do segundo parâmetro da função.

```
1 Consumer consumidor = new DefaultConsumer(channel) {
2   @Override
3   public void handleDelivery(
4                     String consumerTag,
5                     Envelope envelope,
6                     AMQP.BasicProperties properties,
7                     byte[] body) throws IOException {
8     String mensagem = new String(body, "UTF-8");
9     print(mensagem);
10 }};
11 canal.basicConsume(FILA, true, consumidor);
```

Código 3.17. Cliente AMQP no papel de Subscritor, escrito em Java.

3.5.6. XMPP

Para o exemplo com XMPP, uma aplicação cliente Java foi desenvolvida, utilizando a biblioteca Jaxmpp2 [66]. Como servidor XMPP, foi usada a implementação OpenFire [67], instalada em um ambiente Windows. O exemplo está dividido em três partes: (i) Conexão, (ii) Publicador e (iii) Subscritor.

De forma similar ao apresentado na Seção 3.5.5, o procedimento para estabelecer conexão não sofre modificações entre o publicador e o subscritor e é apresentado separadamente no Código 3.18. Como primeiro passo, é necessário criar um objeto da biblioteca (linha 1), que irá permitir a configuração de diversos parâmetros da conexão como o *Jabber ID* (linha 3), a senha (linha 4) e aspectos de segurança da conexão (linha 5).

Além disso, diversos módulos podem ser utilizados para determinar o comportamento do cliente, como, por exemplo, o módulo de presença, que informa o *status* (online/offline) do usuário para os outros clientes (linhas 6 a 8).

Com todas as configurações realizadas, o módulo de publicador/subscritor é registrado (linha 9) e, então, realizada a conexão (linha 10). Observa-se que as mensagens para registro e notificação *publish-subscribe* são montadas pela biblioteca como *stanzas* IQ de tipo *put*, com uma extensão XMPP já disponível e documentada, utilizando uma *tag* específica chamada *pubsub*.

```
1  Jaxmpp jaxmpp = new Jaxmpp();
2  UserProperties up = jaxmpp.getProperties();
3  up.setUserProperty( SessionObject.USER_BARE_JID, JID );
4  up.setUserProperty( SessionObject.PASSWORD, Senha );
5  jaxmpp.getConnectionConfiguration().setDisableTLS(false);
6  PresenceModule.setPresenceStore(
7         jaxmpp.getSessionObject(),
8         new J2SEPresenceStore());
9  jaxmpp.getModulesManager().register(new PubSubModule());
10 jaxmpp.login();
```

Código 3.18. Exemplo de Conexão XMPP, escrito em Java.

O Código 3.19 apresenta o cliente que exerce o papel de subscritor. Na linha 1 é obtida uma referência direta para o módulo de publicador/subscritor que será utilizada para registrar o interesse deste cliente em um determinado tópico (linha 4). É importante notar que são utilizados dois *Jabber ID* diferentes, um para especificar o domínio de interesse e o outro para identificar o usuário.

Ao se registar em um determinado tópico, o cliente passa, como um dos parâmetros, uma função de *callback* (linha 5) que deve tratar os eventos relacionados ao estouro de temporizador (linha 6), estabelecimento da conexão (linha 9) e possíveis erros (linha 14).

```
1  PubSubModule pubsub = jaxmpp.getModule(PubSubModule.class);
2  BareJID jid = JID-Dominio;
3  JID subJid = JID-Usuario;
4  pubsub.subscribe(jid, Topico, subJid,
5    new PubSubModule.SubscriptionAsyncCallback() {
6    public void onTimeout() throws JaxmppException {
7    }
8    protected void onSubscribe(IQ resp,
9              PubSubModule.SubscriptionElement element) {
10   }
11   protected void onEror(IQ resp,
12                 XMPPException.ErrorCondition e1,
13                 PubSubErrorCondition e2)
14                 throws JaxmppException {
15   }});
```

Código 3.19. Cliente Subscritor XMPP, escrito em Java.

O Código 3.20 apresenta o cliente como publicador. Novamente, ele deve primeiro obter a referência para o módulo publicador/subscritor (linha 1). Só então o cliente cria a mensagem (linha 2) e adiciona o conteúdo (linha 3).

Com a mensagem construída o cliente pode utilizar a referência do módulo, junto com a identificação do tópico e o *Jabber ID* para publicar a mensagem (linha 4). O cliente, então, registra uma função de *callback* que deve tratar eventos de erro (linha 7), sucesso (linha 10) e estouro de temporizador (linha 13).

```
PubSubModule pubsub = jaxmpp.getModule(PubSubModule.class);
Element payload = ElementFactory.create("pubsub", null, XML-Schema);
payload.setValue(Conteudo);
pubsub.publishItem(jid, Topico, null, payload, new AsyncCallback() {
  public void onError(Stanza resp,
                      XMPPException.ErrorCondition e)
                      throws JaxmppException {
  }
  @Override
  public void onSuccess(Stanza resp) throws JaxmppException {
  }
  @Override
  public void onTimeout() throws JaxmppException {
  }});
```

Código 3.20. Cliente Publicador XMPP, escrito em Java.

3.6. Estudo de Caso - Gateway MQTT-AMQP

Nesta seção integramos dois protocolos e servidores apresentados na Seção 3.5 para construir um estudo de caso mais estruturado. A Figura 3.15 apresenta o cenário proposto.

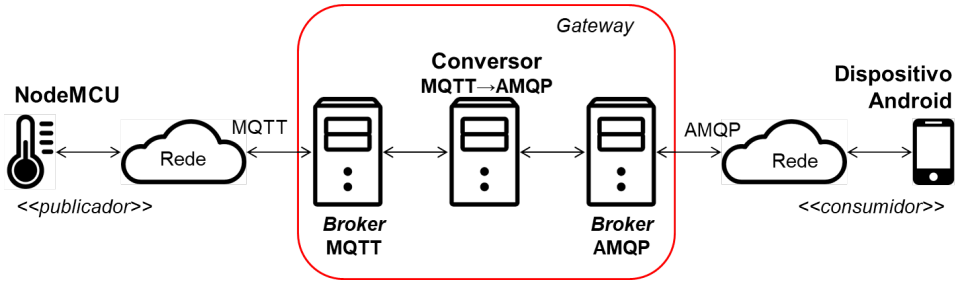


Figura 3.15. Estudo de caso: conversão MQTT para AMQP.

Clientes. Um NodeMCU equipado com um sensor de temperatura *DHT11* periodicamente publica o valor da temperatura em um determinado tópico MQTT. Uma aplicação Java, executando em um *smartphone* Android opera como consumidor AMQP e se inscreve em uma fila para obter o valor da temperatura.

Suporte. As mesmas bibliotecas e *brokers* usados nas Seções 3.5.4 e 3.5.5 serão reusados aqui. Os *brokers* executam em nós independentes, mas podem executar em um mesmo nó.

***Gateway*.** Tanto MQTT e o AMQP, utilizados neste estudo de caso, dão suporte ao estilo *publish-subscribe*, mas têm suas diferenças de funcionamento. O foco do exemplo é conciliar os dois protocolos, fazendo o papel de um *gateway*. Isso será feito com uma aplicação Java que acessa os dois *brokers*, covertendo as chamadas às APIs e adaptando os conteúdos quando necessário.

Observa-se que as aplicações-cliente são similares às utilizadas na seção anterior, deferindo essencialmente no conteúdo sendo publicado/recebido.

Cliente NodeMCU. O Código 3.21 apresenta uma parte da função *loop*, que neste exemplo tem por objetivo obter a temperatura do *DHT11* (linha 4), inserir o valor em uma mensagem de texto *JSON* (linhas 5 a 8) e publicá-la no tópico `uerj/sala01/temperatura` utilizando o protocolo MQTT (linha 9). O passo anterior, de conexão ao *broker*, é similar ao do exemplo da Seção 3.5.4, e por isso é omitido. Na linha 10 o programa *dorme* por cinco segundos, quando então volta a executar o método *loop* novamente.

```
1 char mensagem[tamanho];
2 void loop() {
3   /.../
4   float t = dht.readTemperature();
5   StaticJsonBuffer<200> jsonBuffer;
6   JsonObject& obj = jsonBuffer.createObject();
7   obj["valor"]=t;obj["unidade"]="C";
8   obj.printTo(mensagem);
9   MQTT.publish("uerj/sala01/temperatura",mensagem);
10  delay(5000);
11 }
```

Código 3.21. NodeMCU como publicador de temperatura MQTT.

Conversor MQTT-AMQP. O conversor é implementado como um serviço Java (Código 3.22). O programa pode ser dividido em cinco partes: (i) conexão ao servidor AMQP (linha 2); (ii) conexão ao *broker* MQTT (linha 3); (iii) subscrição ao tópico no *broker* MQTT, `uerj/sala01/temperatura`(linha 4); (iv) recebimento de uma mensagem através do MQTT (linha 6) e sua conversão; e, por último, (v) envio da mensagem convertida para uma fila AMQP (linha 10).

```
1 /.../
2 connectAMQPBroker();
3 connectMQTTBroker();
4 subscribeMQTT("uerj/sala01/temperatura");
5 /.../
6 public void onMessage(String msg, String topico) throws Exception {
7   String newJSON = adicionaDataAoJson(msg,System.currentTimeMillis());
8   publishAMQP(topico.replace('/', '.'),newJSON);
9 }
10 public void publishAMQP(String fila, String message){
11   channel.queueDeclare(fila, false, false, false, null);
12   channel.basicPublish("", fila, null, message.getBytes());
13 }
```

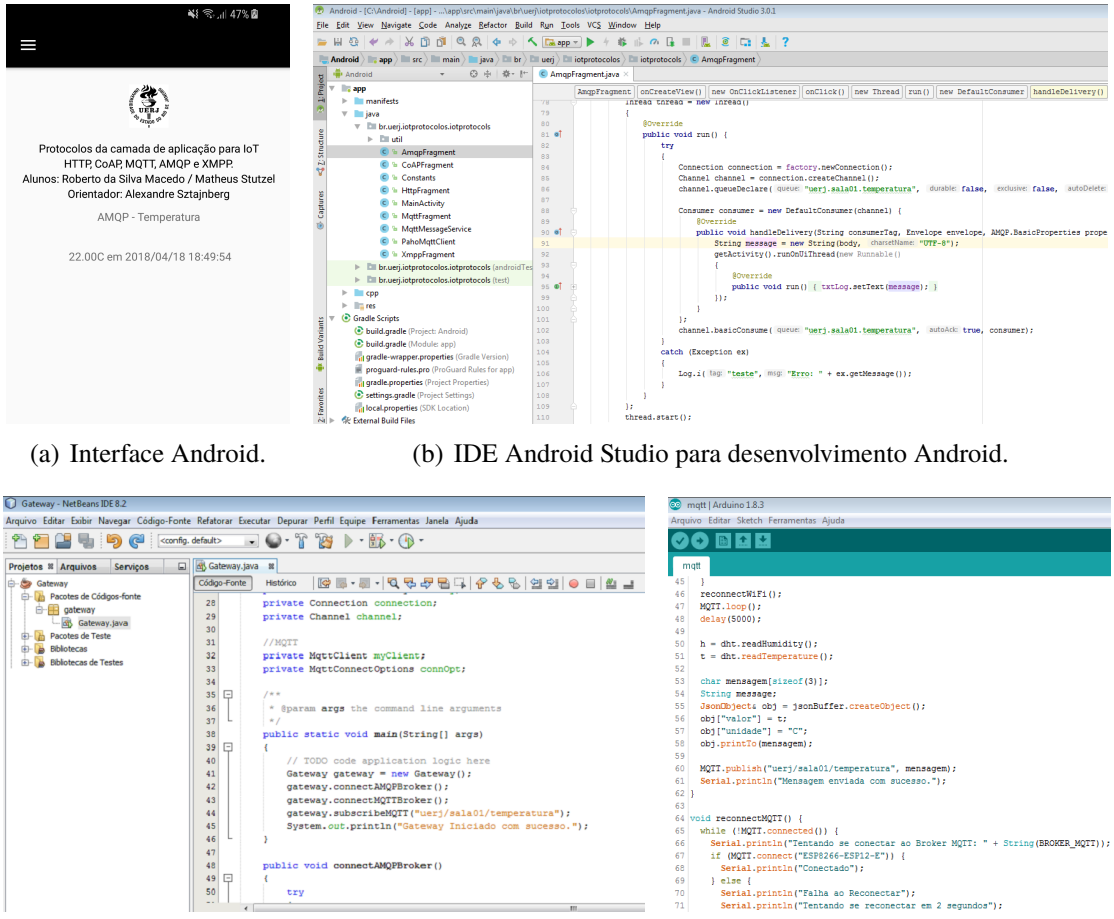
Código 3.22. Gateway realizando a conversão de MQTT para AMQP escrito em Java.

As três primeiras etapas foram apresentadas nas Seções 3.5.5 e 3.5.4, respectivamente, e são essencialmente as mesmas para o estudo de caso. As diferenças estão apenas no nome do tópico e da fila. Assim, destacamos as etapas (iv) e (v), discutindo as rotinas para a conversão entre os protocolos.

A etapa de recebimento e conversão de uma notificação MQTT, é representada no Código 3.22 pelo método de *callback* `onMessage`, linha 6, chamado toda vez que uma mensagem é recebida através do tópico MQTT subscrito por este cliente. Este método recebe como parâmetro a mensagem recebida e o tópico pelo qual ela foi recebida. Neste exemplo adicionamos a informação de data à mensagem recebida e salvamos a mesma novamente em uma estrutura *JSON* (linha 7). Então, a mensagem e o tópico são passados para a etapa (v) do código. Como precisamos conciliar o tópico MQTT com uma fila AMQP, adaptamos o nome original do tópico para o padrão aceito no AMQP. A solução adotada foi substituir o caractere "/" por ".", como mostrado na linha 8.

Por último, a etapa de retransmissão da informação para uma fila AMQP, é apresentada através da função `publishAMQP` (linha 10). Na linha 11 a fila é criada. Como visto anteriormente, caso a fila já exista apenas uma referência é retornada. Então, na linha 12, a mensagem é publicada na fila `uerj.sala01.temperatura` do servidor AMQP.

Cliente Adroid. O cliente executando num dispositivo Android deve se conectar ao *broker* AMQP, criar um canal para a fila desejada e então registrar um consumidor para as mensagens recebidas. Esse procedimento foi discutido na Seção 3.5.5 no Código 3.17.



(a) Interface Android. (b) IDE Android Studio para desenvolvimento Android.

(c) IDE NetBeans para desenvolvimento Java. (d) IDE para desenvolvimento Arduino.

Figura 3.16. Interface Android e IDEs para desenvolvimento.

Concluindo o estudo de caso, para ilustrar o ambiente de desenvolvimento utilizado e a interface desenvolvida, algumas telas foram reproduzidas na Figura 3.16. A interface Android é preparada para exibir a última temperatura e a hora recebida (Figura 3.16(a)), desenvolvida com a IDE Android Studio (Figura 3.16(b)). Os módulos do *gateway*, em Java, foram desenvolvidos com o NetBeans (Figura 3.16(c)). Cabe ao desenvolvedor obter e configurar as bibliotecas, no caso para acesso ao MQTT e AMQP, adequadamente na IDE. O módulo nodeMCU é programado no ambiente oferecido pelo Arduino, que integra o compilador e facilita o *upload* para o dispositivo (Figura 3.16(d)).

3.7. Conclusão

Neste capítulo foram apresentados os principais Protocolos de Aplicação utilizados em aplicações para a Internet das Coisas: CoAP, MQTT, AMQP e o XMPP. Além destes, o mecanismo de WebSocket e o uso de REST sobre HTTP também foram apresentados neste contexto. Todos são padronizados por organizações como o ITU-T, IEEE, e o IETF ou por organizações como o OASIS e o W3C.

Estes protocolos são suportados atualmente por bibliotecas disponíveis para, praticamente, todas as plataformas, incluindo dispositivos com poucos recursos. O CoAP e o MQTT foram concebidos considerando dispositivos e redes com recursos limitados.

O AMQP e XMPP são protocolos baseados em filas de mensagens, adequados para uso nos principais sistemas operacionais e plataformas. Não foram projetados inicialmente para IoT, mas tornaram-se parte integrante da solução, provendo a integração entre redes com dispositivos IoT, sistemas executando na nuvem e dispositivos móveis, permitindo a transferência de dados de forma síncrona e assíncrona entre estes elementos.

Observa-se que as soluções baseadas em um *broker* e em TCP também facilitam a comunicação elementos implantados em redes protegidas por NAT e *firewall*. A rede onde o *broker* executa precisa ser configurada com critério, mas as redes onde executam os clientes, em princípio, não precisam. Com isso, um ambiente inteligente implantado dentro de uma residência pode interagir com elementos em outras redes ou pode ser controlado externamente, desde que consiga manter uma conexão com um *broker*. O MQTT é um exemplo.

Os protocolos e mecanismos apresentados oferecem suporte à interação entre aplicações, dispositivos e serviços nos estilos *request-response* e *publish-subscribe*, além de variações destas, sobre UDP e TCP. As discussões e os exemplos desenvolvidos, envolvendo os vários mecanismos e bibliotecas disponíveis, permitem ao leitor decidir a direção a seguir em um próximo passo para experimentações com os protocolos apresentados.

Referências

[1] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *IEEE Internet of Things Journal*, vol. 1, pp. 22–32, Feb 2014.

[2] M. Díaz, C. Martín, and B. Rubio, "State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing," *Journal of Network and Computer Applications*, vol. 67, pp. 99 – 117, 2016.

[3] P. Bellavista, G. Cardone, A. Corradi, and L. Foschini, "Convergence of manet and wsn in iot urban scenarios," *IEEE Sensors Jrnl.*, vol. 13, pp. 3558–3567, Oct 2013.

[4] F. Kon and E. Zambom, "Cidades inteligentes: Tecnologias, aplicações, iniciativas e desafios," in *CSBC 2016. JAI 1. Porto Alegre, RS.*, 2016.

[5] B. P. Santos, L. Silva, C. Celes, J. Borges, B. Peres, M. Vieira, L. F. Vieira, and A. A. F. Loureiro, "Internet das coisas: da teoria à prática," in *SBRC 2016. Minicurso 1 (MC-1). Salvador, BA.*, pp. 256–315, 2016.

[6] F. Kon and E. F. Z. Santana, "Computação aplicada a cidades inteligentes: Como dados, serviços e aplicações podem melhorar a qualidade de vida nas cidades," in *CSBC 2017. JAI 4. São Paulo, SP.*, p. 2536, 2017.

[7] H. Sundmaeker, P. Guillemin, P. Friess, and S. Woelfflé, "Vision and challenges for realising the internet of things," Tech. Rep. 2, European Commission Information Society and Media, The address of the publisher, 3 2010.

[8] T. Teixeira, S. Hachem, V. Issarny, and N. Georgantas, *Service Oriented Middleware for the Internet of Things: A Perspective*, pp. 220–229. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[9] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys Tutorials*, vol. 17, pp. 2347–2376, Fourthquarter 2015.

[10] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE International Systems Engineering Symposium (ISSE)*, pp. 1–7, Oct 2017.

[11] L. Nastase, "Security in the internet of things: A survey on application layer protocols," in *2017 21st International Conference on Control Systems and Computer Science (CSCS)*, pp. 659–666, May 2017.

[12] J. Ramirez and C. Pedraza, "Performance analysis of communication protocols for internet of things platforms," in *2017 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pp. 1–7, Aug 2017.

[13] CPSE Labs, "Fiware." Web page, 2017. Europ. Union for the development and global deployment of appl. for Future Internet, https://www.fiware.org/.

[14] cpse-labs.eu, "Sofia - smart objects for intelligent applications." Web pg., 2012. ARTEMIS project, http://www.cpse-labs.eu/sp_sofia.php.

[15] J. Soldatos, N. Kefalakis, M. Hauswirth, M. Serrano, J.-P. Calbimonte, M. Riahi, K. Aberer, P. P. Jayaraman, A. Zaslavsky, I. P. Žarko, L. Skorin-Kapov, and R. Herzog, *OpenIoT: Open Source Internet-of-Things in the Cloud*, pp. 13–25. Cham: Springer International Publishing, 2015.

[16] Particle, "The particle iot platform." Web page, 2017. https://www.particle.io/.

[17] P. S. Filho and R. Nascimento, "Knot: Integrando plataformas e aplicações de internet das coisas." X Escola Potiguar de Computação e suas Aplicações. Tutorial 2., 2017. Recife-PE. https://www.knot.cesar.org.br/.

[18] Eclipse IoT Working Group, "Iot standards." Web page, 2012. https://iot.eclipse.org/standards/.

[19] C. Pereira, A. Pinto, A. Aguiar, P. Rocha, F. Santiago, and J. Sousa, "Iot interoperability for actuating applications through standardised m2m communications," in *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pp. 1–6, June 2016.

[20] oneM2M, "Standards for m2m and the internet of things." Web page, 2017. http://www.onem2m.org/.

[21] I. P2413, "Standard for an architectural framework for the internet of things (iot)." http://grouper.ieee.org/groups/2413/, 2016. IEEE-SA Project. Chair: Oleg Logvinov (STMicroelectronics), Manager: Brenda Mancuso.

[22] OASIS, "Organization for the advancement of structured information standards." Web page, 2017. https://www.oasis-open.org/.

[23] ITU-T, "M2m service layer: Apis and protocols overview." ITU-T Focus Group on M2M Service Layer, 4 2014. https://www.itu.int/dms_pub/itu-t/opb/fg/T-FG-M2M-2014-D3.1-PDF-E.pdf.

[24] ITU-T, "Framework of constrained device networking in the iot environments." Global Information InfraStructure, Internet Protocol Aspects, Next-generation Networks, Internet of Things and Smart Cities, 9 2016.

[25] IETF, "The internet of things." Web page, 2017. https://ietf.org/topics/iot/.

[26] IEEE, "The internet of things." Web page, 2017. https://iot.ieee.org/.

[27] H. A. B. Pötter and Alexandre, "Adapting heterogeneous devices into an iot context-aware infrastructure," in *SEAMS '16*, (New York, NY, USA), pp. 64–74, ACM, 2016.

[28] Postscapes, "Iot gateways." Web page, 2017. http://www.postscapes.com/iot-gateways/.

[29] Mqtt.org, "Message queuing telemetry transport." Web page, 2017. http://mqtt.org.

[30] Z. Shelby, K. Harthe, and C. Bormann, "Rfc 7252 constrained application protocol (coap)." Web page, 2017. http://coap.technology/.

[31] AMQP.org, "Advanced Message Queuing Protocol (AMQP)." Web Page, 2014. https://www.amqp.org/.

[32] XMPP Standards Foundation (XSF), "Extensible messaging and presence protocol (xmpp)." Web page., 2011. https://xmpp.org/.

[33] P. M. Krishnapur, S. M, and C. A. Y., "Fast realtime data transfer using xmpp," in *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, pp. 477–479, July 2016.

[34] websocket.org, "About html5 websocket." Web page, 2017. https://websocket.org/aboutwebsocket.html.

[35] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*. USA: Addison-Wesley Publishing Company, 5th ed., 2011.

[36] IETF, "The WebSocket Protocol." IETF RFC6455., 2017. Web page. https://tools.ietf.org/html/rfc6455.

[37] W3C, "The websocket api," 2015. https://www.w3.org/TR/websockets.

[38] Node.js Foundation, "Node.js." Web page, 2018. https://nodejs.org/en/.

[39] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – http/1.1." RFC 2616. IETF. Web page, 6 1999. https://tools.ietf.org/html/rfc2616.

[40] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform resource identifier (uri): Generic syntax." RFC 3986. IETF. Web page, 1 2005. https://tools.ietf.org/html/rfc6690.

[41] D. Robinson and K. Coar, "The common gateway interface ver. 1.1." RFC 3876. IETF, 10 2004. https://tools.ietf.org/html/rfc3875.

[42] W3C, "Web services activity," 2011. https://www.w3.org/2002/ws.

[43] H. G. C. Ferreira, E. D. Canedo, and R. T. de Sousa, "Iot architecture to enable intercommunication through rest api and upnp using ip, zigbee and arduino," in *2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 53–60, Oct 2013.

[44] F. Vásquez, "Prototype for monitoring patients at rest with wearable and iot technology (june 2017)," in *2017 IEEE Central America and Panama Student Conference (CONESCAPAN)*, pp. 1–6, Sept 2017.

[45] IETF, "RESTful Design for Internet of Things Systems." Web page. https://tools.ietf.org/id/draft-keranen-t2trg-rest-iot-04.html, 2017.

[46] The Apache Software Foundation, "Apache server." Web page, 2017. Apache Server, https://httpd.apache.org/.

[47] G. G. R. Gomes and P. N. M. Sampaio, "A specification and tool for the configuration of rest applications," in *2009 International Conference on Advanced Information Networking and Applications Workshops*, pp. 500–505, May 2009.

[48] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of ipv6 packets over ieee 802.15.4 networks." RFC 4944. IETF. Web page, 7 2007. https://datatracker.ietf.org/doc/rfc4944/.

[49] S. Bandyopadhyay, "Lightweight internet protocols for web enablement of sensors using constrained gateway devices," in *2013 International Conference on Computing, Networking and Communications, Workshops Cyber Physical System*, Jan 2013.

[50] IETF, "Constrained restful environments (core) link format," 8 2012. https://tools.ietf.org/html/rfc6690.

[51] ISO, "Message Queuing Telemetry Transport (MQTT) v3.1.1." OASIS Standard. ISO/IEC 20922:2016, 2016. Web page. Edited by Andrew Banks and Rahul Gupta. https://www.iso.org/standard/69466.html.

[52] ISO/IEC JTC 1 Information technology, "Advanced message queuing protocol (amqp) v0-9-1 specification." ISO/IEC 19464:2014, 2014. https://www.iso.org/standard/64955.html.

[53] Pivotal Software, Inc., "Rabbit MQ." Web page., 2017. Rabbit MQ Messaging Server, https://www.rabbitmq.com/.

[54] IETF, "Extensible messaging and presence protocol (xmpp): Core." Web page, 2011. https://tools.ietf.org/html/rfc6120.

[55] XMPP Extensions, "Extensible messaging and presence protocol (xmpp)." Web page. https://xmpp.org/extensions/, 2011. XMPP Extensions.

[56] NodeMcu, "Connect things easy," 2014. http://www.nodemcu.com.

[57] Oracle, "Oracle glassfish server." Web page, 2017. http://www.oracle.com/technetwork/middleware/glassfish/overview/index.html.

[58] Eclipse Foundation, "Eclipse tyrus." Web page, 2018. https://projects.eclipse.org/projects/ee4j.tyrus.

[59] https://github.com/Links2004, "Websocket server and client for arduino." Web page, 2018. https://github.com/Links2004/arduinoWebSockets.

[60] Arduino team, "Arduino core for esp8266 wifi chip." Web page, 2018. https://github.com/esp8266/Arduino.

[61] thingTronics Innovations, "Esp-coap server/client library for arduino." Web page, 2018. https://github.com/automote/ESP-CoAP.

[62] The Eclipse Foundation, "Californium." Web page, 2017. https://www.eclipse.org/californium/.

[63] N. O'Leary, "Arduino client for mqtt." Web page, 2017. https://pubsubclient.knolleary.net/.

[64] Eclipse Paho, "Eclipse paho javascript client." Web page, 2018. https://www.eclipse.org/paho/clients/js/.

[65] Nghia TH, "Iot mqtt dashboard." Google Play, 2018. https://play.google.com/store/apps/details?id=com.thn.iotmqttdashboard.

[66] Tigase, Inc., "Tigase JaXMPP Client Library." Web page, 2017. https://tigase.tech/projects/jaxmpp2.

[67] Ignite Realtime, "Openfire XMPP Server." Web page, 2017. https://www.igniterealtime.org/projects/openfire/.

3.8. Anexo

Aqui estão reunidas algumas referências para projetos e bibliotecas não utilizados nas seções anteriores, mas relacionados aos assuntos tratados.

Além do RabbitMQ, outros *brokers* e bibliotecas, estão disponíveis, como o Mosquito (https://mosquitto.org/), da Eclipse Foundation; ou o HIVEMQ (https://www.hivemq.com);

Existem instâncias de *broker* MQTT disponíveis na Internet. Nos testes e prototipação com o Dashboard utilizamos o servidor cloudmqtt.com;

ApacheMQ, sistema de mensagens Apache (http://activemq.apache.org/);

STOMP (Simple (or Streaming) Text Orientated Messaging Prot.). https://stomp.github.io;

Algumas extensões do XMPP são de destaque para uso na Internet das Coisas, como por exemplo, XEP-0060 - Publish-Subscribe, XEP-0030 - serviço de descoberta, XEP-0072 - SOAP Over XMPP, XEP-0239 - Binary XMPP;

Lightweight M2M (LWM2M). https://www.omaspecworks.org. Protocolo de gerenciamento de dispositivos e outras especificações;

CoAP.net (CoAP para .NET). https://github.com/smeshlink/CoAP.NET;

Padrão DDS (Data Distribution Service Spec., da OMG). https://www.omg.org/spec/DDS;

Frameworks para o desenvolvimento de aplicações de ambientes inteligentes. *SmartHome* (https://www.eclipse.org/smarthome/) e OpenHAB, baseado no Smart Home (https://www.openhab.org/) usando MQTT;

DOJOT - Com base no FIWARE, este é uma plataforma brasileira de desenvolvimento de soluções IoT para cidades inteligentes. http://www.dojot.com.br/sobre-a-dojot-iot/;

JSXC (JavaScript XMPP Client), cliente XMPP em JavaScript (https://www.jsxc.org/). GetKaiwa, outra opção de um cliente, baseado em Web (http://www.getKaiwa.com). Clientes XMPP para outras linguagens de programação, podem ser encontrados em: https://xmpp.org/software/clients.html;

XMPP-IoT, página Web dedicada a promover a utilização do XMPP em soluções IoT. http://www.xmpp-iot.org/.