

## Capítulo

# 11

## Monitoramento de tráfego em redes de Internet das Coisas

Syllas Rangel C. Magalhães; Victória Tomé Oliveira; Francisco Evangelista N. Filho; Wendley S. Silva

### *Resumo*

*De um modo geral, o conceito de Internet das Coisas (IoT) refere-se à interconexão de objetos através da rede mundial de computadores (Internet), objetos que podem ser desde dispositivos cotidianos, como os encontrados nas casas inteligentes, até equipamentos específicos, como no caso de equipamentos para assistência médica. A massiva quantidade de “objetos” nesse tipo de rede exige sistemas de comunicação cada vez mais robustos. Recentes avanços nos sistemas de comunicações móveis e nas comunicações máquina-a-máquina (M2M) prometem viabilizar a ascensão da IoT. Entretanto, devido ao alto tráfego de dados e a enorme quantidade de dispositivos conectados, surgem alguns desafios como privacidade e segurança. Este minicurso apresentará uma forma simples de analisar o tráfego de dados em uma rede IoT através do microcomputador single-board Raspberry PI®, com foco na análise dos protocolos mais críticos para esse tipo de rede.*

### **11.1. Introdução**

Nos últimos anos houve um significativo crescimento da adoção de dispositivos sem fio, tais como *smartphones*, *tablets*, câmeras, *notebooks* e sensores de saúde. As projeções da empresa Cisco sugerem que esse crescimento continuará, e entre os anos de 2016 e 2021 ocorrerá um crescimento de 7,3 vezes na quantidade de dispositivos *inteligentes* conectados [Cisco, 2017]. Tais dispositivos, associados com as novas tecnologias de comunicação da Internet atual e futura, compõem o que muitas vezes é classificada como Internet das Coisas (IoT, do inglês *Internet of Things*).

Conceituar Internet das Coisas não é uma tarefa fácil. O termo foi utilizado pela primeira vez pelo britânico Ashton e seus colegas de laboratório, no trabalho “*I made at Procter & Gamble*” publicado em 1999 [Ashton, 2009]. Contudo, IoT também pode ser compreendida como uma infraestrutura de rede que liga objetos físicos e virtuais através da captura de dados e comunicação a uma plataforma que possibilita a execução de

uma aplicação [Gubbi et al., 2013]. Essa infraestrutura de comunicação inclui a Internet, outras redes existentes e em desenvolvimento.

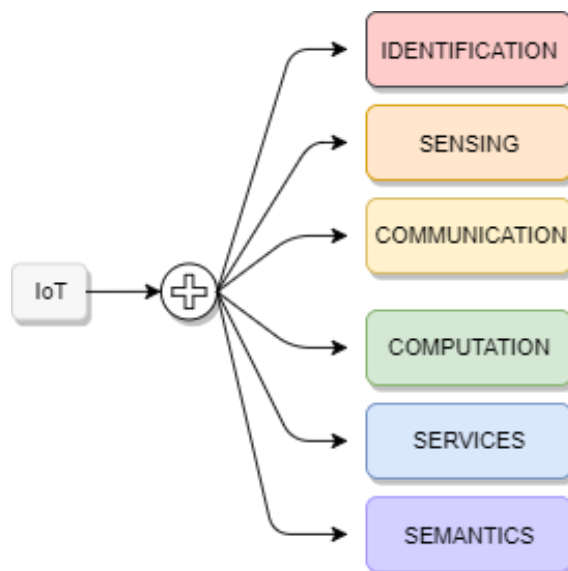
Os sistemas IoT geralmente possuem a identificação específica do objeto e a capacidade de interconexão como princípios para o desenvolvimento de serviços cooperativos independentes. Em razão disso, algumas vezes os termos *Machine-to-Machine* (M2M) e IoT são utilizados de forma intercambiável, no entanto, a característica diferencial e mais importante de IoT é a informação que os objetos conectados podem fornecer, e como essas informações podem ser combinadas e apresentadas por uma aplicação.

Neste trabalho, serão apresentadas as definições e motivações para as pesquisas de *Internet of Things*, com atenção especial às aplicações, que serão descritas a fim de despertar o interesse do participante acerca das técnicas utilizadas. Também serão discutidos os elementos que compõem a IoT e os seus principais protocolos de comunicação. A parte prática do minicurso ficará na Seção **Análise de Tráfego**, onde serão apresentadas as técnicas de monitoramento de tráfego em redes IoT, bem como será detalhada a utilização de *softwares* para o monitoramento.

## 11.2. Elementos da IoT

Elementos de IoT (*Internet of Things*) podem ser definidos como componentes, representados em blocos na Figura 11.1, que integram um ambiente de *Internet of Things*. Compreendê-los é importante, pois contribui na real definição de *Internet of Things*, além de fornecer uma ampla visão de suas funcionalidades.

Figura 11.1. Elementos de *Internet of Things* [Al-Fuqaha et al., 2015].



Como visto acima, um ambiente IoT, do mais básico ao mais complexo, é formado por seis elementos: *identification*, *sensing*, *communication*, *computation*, *services* e *semantics* [Al-Fuqaha et al., 2015]. Cabe ressaltar que alguns autores condensam os seis elementos em três: *hardware*, *middleware* e *presentation* [Gubbi et al., 2013]. A seguir, cada um desses blocos será analisado mais detalhadamente.

### 11.2.1. Identification

O processo de identificação (*identification*) é importante, pois é responsável por atribuir um identificador aos objetos e serviços ofertados. Dessa forma, reconhecer que determinada solicitação ocasionou uma tarefa qualquer torna-se um processo mais simples. Além disso, esse procedimento permite diferenciar o ID (nome) de um objeto de seu endereço (valor atribuído dentro da rede de comunicação), o que permite solucionar até mesmo problemas ocasionados pela utilização de redes públicas [Al-Fuqaha et al., 2015]. A etapa *identification* é equivalente, por exemplo, ao *Electronic Product Code* (EPC), *uCode* e ao *Digital Object Identifier* (DOI), que são, respectivamente, padrões para identificar unicamente objetos físicos no mundo real, objetos/lugares no mundo real e documentos digitais. Para realizar esse processo em ambientes IoT, são utilizados, sobretudo, cabeçalhos de protocolos, como por exemplo IPv4, IPv6 e 6LoWPAN [Ferreira, 2013].

### 11.2.2. Sensing

Comumente chamados de sensores ou atuadores, são dispositivos capazes de coletar os dados dos objetos contidos em uma rede de comunicação. Eles são capturados e enviados de volta a um *data warehouse* (sistema de armazenamento de dados digitais), banco de dados ou nuvem, para que possam ser armazenados, analisados e para que ações necessárias sejam executadas. Atualmente, muitas empresas estão investindo em produtos para realização dessa etapa, como a WeMo, ZigBee, entre outras.

### 11.2.3. Communication

O objetivo das redes de comunicação de *Internet of Things* é conectar diferentes objetos, para que sejam capazes de fornecer serviços inteligentes. Comumente, essas redes são de baixas frequências e operam em ambientes onde há perdas e ruídos [Al-Fuqaha et al., 2015]. A etapa *communication* consiste em estudar a utilização de protocolos e tecnologias capazes de realizar as tarefas acima citadas, com perdas e ruídos reduzidos. Entre as tecnologias mais populares, encontram-se: *WiFi*, *Bluetooth*, IEEE 802.15.4, *Z-wave*, *NFC* (*Near Field Communication*), *UWB* (*Ultra-wide bandwidth*), *RFID* (*Radio-frequency identification*) e *LTE-advanced*. As tecnologias utilizadas na realização desse minicurso serão melhor descritas na Seção 11.3.

### 11.2.4. Computation

Essa etapa é definida como o “cérebro físico” da *Internet of Things*. É responsável por realizar o processo descrito na Subseção 11.2.2. Atualmente, muitos microcontroladores e *Single Board Computers* (SBCs) são capazes de simular ambientes IoT. Entre os mais comuns estão: *Arduino*, *Raspberry Pi*<sup>®</sup> - utilizado na produção do minicurso -, *BeagleBone*, *Galileo Gen 2*, entre outros. Além dos *hardwares*, os *softwares* também se encontram mais refinados. Ações como sistemas operacionais com funcionalidades nativas para ambientes IoT ou até mesmo empresas desenvolvendo *softwares* específicos (*Contiki RTOS*, *TinyOS*, *LiteOS*, *RiotOS* e *OAA*) estão se difundindo. Cabe ressaltar que a evolução dos serviços em nuvem também representou um grande avanço para a *Internet of Things*, pois permitiu armazenamento e processamento em tempo real de um volume considerável de dados, o que acarretou em mais informações para que os usuários aperfeiçoem suas redes de comunicação IoT [Al-Fuqaha et al., 2015].

### 11.2.5. Services

Ambientes IoT podem ser classificados em quatro classes: *identity-related services*, *information aggregation services*, *collaborative-aware services* e *ubiquitous services* [Al-Fuqaha et al., 2015]. Essas classes definem as funcionalidades e aplicações de uma rede de comunicação de *Internet of Things*, como, por exemplo: *smart grids*, *smart city*, *smart healthcare*, *industrial automation*, entre outras. O bloco *services* diz respeito ao que uma rede de comunicação IoT é capaz de aprovisionar. Explorar cada uma das classes foge do escopo do minicurso. Entretanto, esses exemplos são suficientes para entendermos o poder e importância da *Internet of Things*.

### 11.2.6. Semantics

A etapa de semântica (*semantics*) consiste em extrair conhecimento de uma rede de comunicação de *Internet of Things* - composta por diversos componentes/objetos -, para que as solicitações realizadas possam ser atendidas, ou seja, para fornecer os serviços mínimos necessários. Um ambiente IoT realiza esse procedimento através da descoberta, utilização, análise e modelagem dos recursos fornecidos. Uma equivalência da *semantics* seria a utilização de uma Inteligência Artificial (IA) na automatização de um procedimento.

De forma análoga à definição de *communication* da Subseção 11.2.4, o processo de *semantics* pode ser analisado como as sinapses do “cérebro físico”. Ademais, pode-se citar algumas tecnologias utilizadas nessa etapa: *Resource Description Framework* (RDF), *Web Ontology Language* (OWL), *Efficient XML Interchange* (EXI).

Por fim, na Tabela 11.1 é exibido um resumo do que foi exposto na Seção 11.1.

**Tabela 11.1. Síntese da Seção 11.2 [Al-Fuqaha et al., 2015].**

| Elementos de Internet das Coisas |           | Exemplos                                                                                       |
|----------------------------------|-----------|------------------------------------------------------------------------------------------------|
| Identification                   | Nomear    | EPC, <i>uCode</i> , DOI, etc.                                                                  |
|                                  | Endereçar | IPv4, IPv6, 6LoWPAN, etc.                                                                      |
| Sensing                          |           | Sensores, atuadores, dispositivos vestíveis, embarcados, etc.                                  |
| Communication                    |           | WiFi, Bluetooth, IEEE 802.15.4, Z-wave, LTE-advanced, UWB, RFID, etc.                          |
| Computation                      | Hardware  | Arduino, Raspberry Pi <sup>®</sup> , BeagleBone, Galileo Gen 2, Smartphones, Smartthings, etc. |
|                                  | Software  | Contiki RTOS, TinyOS, LiteOS, RIOTOS, OAA, Nimbits, etc.                                       |
| Services                         |           | Smart grids, Smart homes, Smart city, etc.                                                     |
| Semantics                        |           | RDF, EXI, OWL, etc.                                                                            |

### 11.3. Protocolos de IoT

Com o crescimento de redes de comunicação IoT, está havendo um aumento no número de dispositivos conectados à Internet. Esse fenômeno ocorre, sobretudo pela heterogeneidade dos dispositivos utilizados em ambientes IoT. Devido a isso, torna-se essencial o estudo e desenvolvimento de protocolos que possam, por exemplo, suprir limitações impostas pelos dispositivos, como ocorre em muitos casos. Diferentes grupos foram criados para fornecer protocolos de apoio à Internet das Coisas, incluindo alguns liderados pela *World Wide Web Consortium (W3C)*, *Internet Engineering Task Force (IETF)*, *EPCglobal*, *Institute of Electrical and Electronics Engineers (IEEE)* e o *European Telecommunications Standards Institute (ETSI)* [Al-Fuqaha et al., 2015]. Na Tabela 11.2 são exibidos alguns dos principais protocolos utilizados em ambientes IoT, separados em 4 grupos, como proposto em [Al-Fuqaha et al., 2015], denominados: protocolos de aplicação, protocolos de descoberta de serviços, protocolos de infraestrutura e outros protocolos influentes. Nas subseções a seguir, será apresentada uma visão geral de alguns destes protocolos, citando suas principais características e funcionalidades, de acordo com a classificação proposta. Dar-se-á uma atenção especial aos protocolos CoAP e MQTT, ambos da camada de aplicação.

**Tabela 11.2. Protocolos relacionados à IoT [Al-Fuqaha et al., 2015].**

|                                      |                           |               |           |       |               |             |        |
|--------------------------------------|---------------------------|---------------|-----------|-------|---------------|-------------|--------|
| Protocolos de aplicação              |                           | AMQP          | CoAP      | DDS   | HTTP REST     | MQTT        | XMPP   |
| Protocolos de descoberta de serviços |                           | mDNS          |           |       | DNS-SD        |             |        |
| Protocolos de infraestrutura         | Roteamento                | RPL           |           |       |               |             |        |
|                                      | Camada de Rede            | 6LoWPAN       |           |       | IPv4/IPv6     |             |        |
|                                      | Camada de Enlace          | IEEE 802.15.4 |           |       |               |             |        |
|                                      | Camada Física/Dispositivo | LTE-A         | EPCglobal |       | IEEE 802.15.4 |             | Z-Wave |
| Outros protocolos influentes         |                           | IEEE 1888.3   |           | IPSec |               | IEEE 1905.1 |        |

#### 11.3.1. Protocolos de aplicação

O protocolo HTTP é utilizado na Internet para prover o acesso à informação desde 1990, constituindo a base para a comunicação de dados na *World Wide Web (WWW)*. Embora tenha passado por algumas reformulações depois de sua criação, o HTTP foi projetado para redes com computadores pessoais, com maior poder de processamento se comparado aos dispositivos utilizados em redes IoT. As restrições impostas pelos equipamentos utilizados em ambientes IoT limitam o uso do protocolo HTTP nesses. Sendo assim, surgem alguns protocolos na camada de aplicação como alternativa ao HTTP, como, por exemplo, o CoAP e MQTT, ambos projetados para troca de informação entre dispositivos com baixo poder computacional. Além destes, outros protocolos vêm sendo utilizados em redes IoT, como é caso do XMPP, AMQP e DDS. A seguir, alguns destes protocolos serão detalhados.

**Tabela 11.3. Formada da mensagem CoAP [Shelby et al., 2014].**

|                                               |                                  |     |        |                |
|-----------------------------------------------|----------------------------------|-----|--------|----------------|
| 0 1                                           | 2 3                              | 4 7 | 8 15   | 16 31          |
| Ver                                           | T                                | TKL | Código | ID da Mensagem |
| <i>Token</i> (caso exista, com TKL bytes) ... |                                  |     |        |                |
| Opções (caso existam) ...                     |                                  |     |        |                |
| 1 1 1 1 1 1 1 1                               | <i>Payload</i> (caso exista) ... |     |        |                |

#### 11.3.1.1. *Constrained Application Protocol* (CoAP)

O *Constrained Application Protocol* (CoAP) foi proposto pelo IETF *Constrained RESTful Environments (CoRE) working group*, com o objetivo de otimizar o uso da arquitetura REST (*REpresentational State Transfer*) em nós - por exemplo, microcontroladores de 8 bits com RAM e ROM limitadas - e redes (e.g. 6LoWPAN) com poder computacional restrito [Kuladinithi et al., 2011] [Shelby et al., 2014]. A forma de interação do CoAP é semelhante ao modelo cliente/servidor no protocolo HTTP, entretanto, ele também oferece recursos para M2M (*Machine-to-Machine*), como descoberta interna, suporte *multicast* e trocas de mensagens assíncronas. Semelhante ao HTTP, uma solicitação CoAP é enviada por um cliente para solicitar uma ação usando *Uniform Resource Identifiers* (URIs), que em seguida é respondido pelo servidor com um código de resposta [Joshi and Kaur, 2015]. Ao contrário do REST (que utiliza HTTP sobre TCP), o CoAP funciona sobre UDP por padrão, o que o torna mais adequado para as aplicações IoT. Além disso, o CoAP modifica algumas funcionalidades HTTP para atender aos requisitos de IoT, como baixo consumo de energia e operação na presença de links com perdas e ruídos. No entanto, como o CoAP foi projetado com base em REST, a tradução entre os protocolos HTTP e CoAP é facilitada.

O CoAP pode ser representado com duas camadas em sua arquitetura: uma responsável pela implementação dos mecanismos de requisição/resposta e a outra pela comunicação e, opcionalmente, confiabilidade sobre UDP. Ele possui, ainda, quatro tipos de mensagens: *confirmable*, *non-confirmable*, *reset* e *acknowledgement*. Uma mensagem do tipo *confirmable* (CON) requer uma resposta do receptor com uma confirmação de recebimento. Essa é contrária à mensagem *non-confirmable* (NON), que não garante o recebimento no receptor, uma vez que este tipo de mensagem, como o próprio nome sugere, não exige confirmação. A mensagem de confirmação propriamente dita é chamada de *acknowledgement* (ACK). Ela é transmitida em resposta a uma mensagem *confirmable* recebida de forma correta. Por fim, a tipo *reset* (RST) é enviada basicamente em três situações: quando ocorre erro na mensagem; quando a mensagem não é compreensível; quando o receptor não está interessado na comunicação com o remetente.

As mensagens CoAP são codificadas em um formato binário simples. A primeira parte da mensagem é um cabeçalho fixo de 4 bytes, que pode ser seguido por um *token*, algumas opções e um *payload*, como ilustrado na Tabela 11.3. O cabeçalho fixo é dividido em cinco partes: Ver (versão do protocolo), T (tipo de mensagem), TKL (tamanho do campo *Token*), Código e ID da mensagem. Os três bits mais significativos do campo código carregam a informação de qual classe a mensagem pertence, podendo indicar uma requisição (0), uma resposta bem-sucedida (2), uma resposta de erro do cliente (4) ou

uma resposta de erro do servidor (5). Já os cinco bits menos significativos desse campo, representam uma subclasse de mensagem. Por exemplo, para uma mensagem de requisição que utiliza o método GET, o campo código seria representado por “0.01”, em que o primeiro dígito indica a classe (nesse caso requisição) e os dois dígitos depois do ponto representam a subclasse (nesse caso GET) [Shelby et al., 2014]. Na tabela 11.4 estão representados todos os códigos de resposta CoAP.

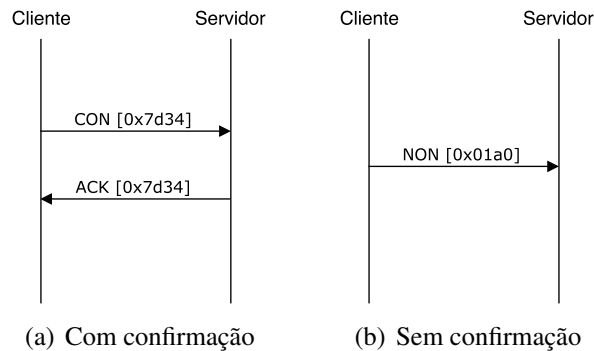
**Tabela 11.4. Códigos de resposta CoAP [Shelby et al., 2014].**

| Código | Nome                       | Descrição                                                                   |
|--------|----------------------------|-----------------------------------------------------------------------------|
| 2.01   | Created                    | Resposta a uma requisição PUT ou POST (criado)                              |
| 2.02   | Deleted                    | Resposta a uma requisição que torna um recurso indisponível                 |
| 2.03   | Valid                      | Indica que a resposta identificada pela <i>entity-tag</i> é válida          |
| 2.04   | Changed                    | Resposta a uma requisição PUT ou POST (modificado)                          |
| 2.05   | Content                    | Indica a presença do conteúdo requisitado em um GET                         |
| 4.00   | Bad Request                | Indica que o servidor não “entendeu” a requisição                           |
| 4.01   | Unauthorized               | O cliente não está autorizado a executar a ação solicitada                  |
| 4.02   | Bad Option                 | O servidor não reconheceu uma ou mais opção crítica                         |
| 4.03   | Forbidden                  | Acesso ao recurso é proibido                                                |
| 4.04   | Not Found                  | Não encontrado                                                              |
| 4.05   | Method Not Allowed         | Método não permitido                                                        |
| 4.06   | Not Acceptable             | O recurso de destino não possui uma representação aceitável                 |
| 4.12   | Precondition Failed        | Precondição falhou                                                          |
| 4.13   | Request Entity Too Large   | Entidade de requisição muito grande                                         |
| 4.15   | Unsupported Content-Format | <i>Context-format</i> não suportado                                         |
| 5.00   | Internal Server Error      | Erro interno no servido                                                     |
| 5.01   | Not Implemented            | Não implementado                                                            |
| 5.02   | Bad Gateway                | Erro quando o servidor atuava como <i>gateway</i> ou <i>proxy</i>           |
| 5.03   | Service Unavailable        | Serviço indisponível                                                        |
| 5.04   | Gateway Timeout            | <i>Timeout</i> quando o servidor atuava como <i>gateway</i> ou <i>proxy</i> |
| 5.05   | Proxying Not Supported     | <i>Proxying</i> não suportado                                               |

Na Figura 11.2 são exibidas duas situações de uma comunicação CoAP. Na esquerda (Figura 11.2a) é mostrado um exemplo de uma transmissão de mensagem com confirmação (ACK). Neste caso, o transmissor continua retransmitindo a mensagem de acordo com um *timeout* padrão até que receba uma mensagem ACK com o mesmo ID da mensagem enviada (neste caso 0x7d34). Já na Figura 11.2b é mostrado um exemplo simples de uma transmissão não confiável (sem confirmação), em que o transmissor simplesmente envia uma mensagem e não requer nenhum tipo de confirmação. Em ambas as situações quando o destinatário não é capaz de processar a mensagem, ou seja, nem mesmo é capaz de fornecer uma resposta de erro adequada, ele responde com uma mensagem *reset* (RST).

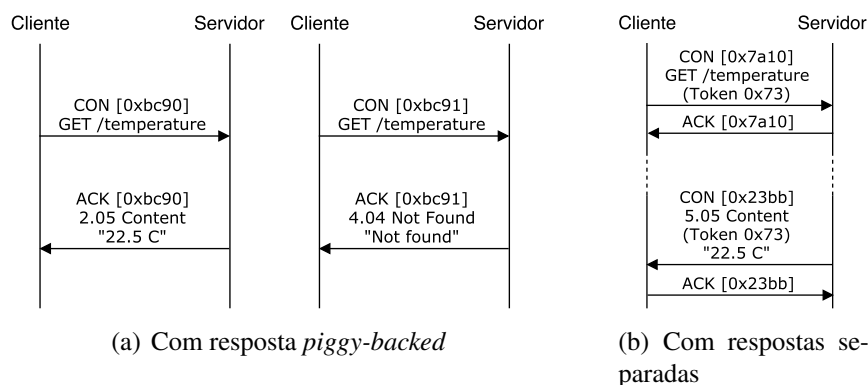
Existem quatro métodos de requisição CoAP que foram definidos em [Shelby et al., 2014], sendo eles *GET* (código 0.01), *POST* (código 0.02), *PUT* (código 0.03) e *DELETE* (código 0.04). As requisições podem ser realizadas em mensagens do tipo *confirmable* (CON) ou *non-confirmable* (NON). No caso de a resposta a uma requisição CoAP do tipo *confirmable* estar disponível imediatamente, esta é enviada junto à mensagem de confirmação (ACK), ocasião em que é chamada de resposta *piggy-backed*. Na Figura 11.3a são ilustradas duas situações onde ocorre uma resposta *piggy-backed*. Na

**Figura 11.2. Comunicação CoAP com e sem confirmação de recebimento [Shelby et al., 2014].**



primeira, a requisição obteve uma resposta satisfatória e no segundo caso uma resposta de 4.04 (*Not found*). Por outro lado, a Figura 11.3b mostra uma requisição GET do tipo *confirmable* com respostas separadas. Neste caso, o servidor não estava disponível para responder imediatamente a requisição. Dessa forma, para que o cliente fique ciente de que a requisição foi recebida corretamente e pare de retransmitir a mensagem, é enviada uma confirmação (ACK) vazia. Após algum tempo, quando a resposta estiver disponível, o servidor a envia com solicitação de confirmação. Cabe ressaltar que, neste caso, a distinção entre as respostas esperadas pelo cliente é feita a partir do *Token* (não confundir com ID da mensagem) que, apesar de ter sido abstraído nos exemplos anteriores, está em todas as mensagens CoAP. No caso de uma requisição sem confirmação (NON), a comunicação ocorrerá de forma parecida, porém todas as respostas também serão do tipo *non-confirmable* (não existirão mensagens de ACK na comunicação).

**Figura 11.3. Requisições CoAP [Shelby et al., 2014].**



Informações detalhadas sobre a arquitetura e funcionamento do protocolo CoAP podem ser encontradas em [Shelby et al., 2014].



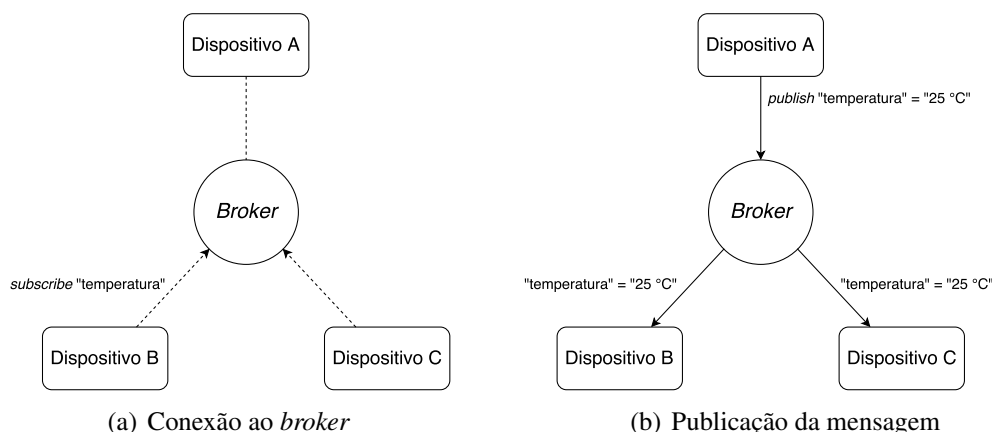
### 11.3.1.2. Message Queue Telemetry Transport (MQTT)

O MQTT é um protocolo de mensagens introduzido por Andy Stanford-Clark da IBM e Arlen Nipper da Arcom (agora Eurotech) em 1999 e foi padronizado em 2013 pela OASIS<sup>1</sup>. Esse protocolo é otimizado para redes com capacidade de processamento limitado, pequena capacidade de memória ou alta latência. Comparado ao HTTP, o MQTT oferece várias vantagens aos aplicativos móveis: tempos de resposta mais rápidos, maior produtividade, maior confiabilidade de mensagens, baixo uso de banda e baixo consumo da bateria [Chen et al., 2014]. Desde a sua criação, o MQTT passou por algumas modificações significativas e atualmente está em sua versão 3.1.1.

O MQTT possui arquitetura cliente/servidor e utiliza, para troca de mensagens, o paradigma *publish/subscribe* (*pub/sub*). Diferente do CoAP, neste protocolo todas as “coisas” são clientes que se conectam a um servidor, denominado *broker*. O *broker* é responsável por receber, enfileirar e disparar as mensagens recebidas dos publicadores (*publishers*) para os assinantes (*subscribers*) [Desai, 2015]. Neste protocolo, a conexão ao *broker* pode ser feita através do TCP, TLS ou *WebSocket* e toda a troca de mensagens é baseada no conceito de tópico (*topic*), que nada mais é do que um endereço.

Na comunicação MQTT, inicialmente, os dispositivos se conectam ao *broker* e em seguida podem se registrar em um ou mais tópicos do *broker* (*subscribe*) para obter todas as informações que forem “publicadas” nestes tópicos. As informações, por sua vez, são introduzidas pelos publicadores (*publishers*), que também precisam estar conectados ao *broker*. Para exemplificar o funcionamento deste protocolo, imagine a seguinte situação: um dispositivo A possui um sensor que mede constantemente a temperatura em determinada região e precisa relatar periodicamente a outros dois dispositivos (B e C). Utilizando a comunicação MQTT, este problema poderia ser facilmente resolvido como ilustrado na Figura 11.4.

**Figura 11.4. Exemplo de comunicação MQTT [Elaborada pelos autores].**



Como mostrado na Figura 11.4a, inicialmente, todos os dispositivos precisam iniciar uma conexão TCP com o *broker* e os dispositivos interessados na temperatura. Além

<sup>1</sup> <http://mqtt.org/faq>

disso, precisam se inscrever no tópico “temperatura”. Dessa forma, toda vez que o dispositivo A publicar a temperatura no tópico de interesse, o *broker* se encarregará de enviar a informação para todos os *subscribers* deste tópico. Na Figura 11.4b é mostrado um exemplo de *publish*, em que o dispositivo A publica uma temperatura de 25 °C.

O protocolo MQTT funciona através da troca de uma série de pacotes de controle. Um pacote de controle MQTT pode conter até três partes, sendo elas o cabeçalho fixo (*fixed header*), o cabeçalho variável (*variable header*) e o *payload*, sempre nesta ordem [Standard, 2014]. Todo pacote de controle MQTT deve conter, pelo menos, o cabeçalho fixo. O tamanho deste cabeçalho varia de 2 a 5 *bytes*, onde o primeiro *byte* contém o tipo de mensagem (do *bit* 7 ao 4) e algumas *flags* (do *bit* 3 ao 0), que, entre outras informações, podem indicar o nível de QoS (qualidade de serviço) da transmissão (*bits* 1 e 2), divididos em três: (i) QoS 0, (ii) QoS 1 e (iii) QoS 2. Em (i) a mensagem é entregue, no máximo, uma vez; já em (ii) a mensagem é entregue, pelo menos, uma vez; por fim, em (iii), a mensagem é entregue exatamente uma vez. A segunda parte do cabeçalho fixo, chamada “comprimento restante”, codifica a quantidade de *bytes* restantes no pacote (não considera o tamanho do cabeçalho fixo) e contém entre 1 e 4 *bytes*. Neste caso, os sete *bits* menos significativos de cada *byte* codificam a informação e o *bit* mais significativo indica continuação, ou seja, caso seja “1”, indica que existe, pelo menos, mais um *byte* de “comprimento restante”. Pode existir, ainda, um cabeçalho variável e um *payload* no pacote, dependendo do tipo de mensagem. A estrutura básica de um pacote MQTT é representada na Tabela 11.5.

**Tabela 11.5. Estrutura de um pacote de controle MQTT [Al-Fuqaha et al., 2015].**

|                                  |   |   |   |                                |   |   |   |
|----------------------------------|---|---|---|--------------------------------|---|---|---|
| 7                                | 6 | 5 | 4 | 3                              | 2 | 1 | 0 |
| Tipo do pacote                   |   |   |   | Sinalizadores ( <i>Flags</i> ) |   |   |   |
| Comprimento restante (1~4 bytes) |   |   |   |                                |   |   |   |
| Cabeçalho variável (opcional)    |   |   |   |                                |   |   |   |
| <i>Payload</i> (opcional)        |   |   |   |                                |   |   |   |

Apesar de existirem diversos tipos de pacotes MQTT, representados na Tabela 11.6, muitas aplicações do protocolo podem ser realizadas apenas com os comandos *CONNECT*, *PUBLISH*, *SUBSCRIBE* e *DISCONNECT* [Chen et al., 2014]. O comando *CONNECT* deve, obrigatoriamente, ser o primeiro pacote enviado após o estabelecimento da conexão da rede com o servidor. Além disso, esse deve ser enviado apenas uma vez. Caso contrário, o cliente será desconectado. No pacote *CONNECT* deve ser enviado um identificador único para o cliente e, opcionalmente, informações como nome de usuário, senha, *Will topic* (tópico que receberá a mensagem “Will” caso a conexão seja encerrada sem a utilização do comando *DISCONNECT*), entre outras, desde que sinalizadas no cabeçalho variável. Por outro lado, o comando *DISCONNECT* é utilizado quando o cliente quer informar explicitamente sua desconexão ao servidor. Neste caso, a mensagem “Will”, caso exista, deve ser descartada. Na Seção 11.4 serão dados mais detalhes dos pacotes *PUBLISH* e *SUBSCRIBE*, bem como outros aspectos relevantes da comunicação neste protocolo e exemplos práticos. Todas as *flags* e informações sobre os pacotes MQTT são detalhadas em [Standard, 2014].

**Tabela 11.6. Tipos de pacotes de controle do MQTT versão 3.1.1 [Standard, 2014].**

| Nome        | Código | Direção de Fluxo   | Descrição                                      |
|-------------|--------|--------------------|------------------------------------------------|
| Reservado   | 0      | Proibido           | Reservado                                      |
| CONNECT     | 1      | Cliente → Servidor | Requisição de conexão                          |
| CONNACK     | 2      | Cliente ← Servidor | ACK de conexão                                 |
| PUBLISH     | 3      | Cliente ⇌ Servidor | Publicação de mensagem                         |
| PUBACK      | 4      | Cliente ⇌ Servidor | ACK de publicação                              |
| PUBREC      | 5      | Cliente ⇌ Servidor | Publicação recebida (QoS 2, parte 1)           |
| PUBREL      | 6      | Cliente ⇌ Servidor | Publicação liberada (QoS 2, parte 2)           |
| PUBCOMP     | 7      | Cliente ⇌ Servidor | Publicação completa (QoS 2, parte 3)           |
| SUBSCRIBE   | 8      | Cliente → Servidor | Requisição de <i>subscribe</i>                 |
| SUBACK      | 9      | Cliente ← Servidor | ACK de <i>subscribe</i>                        |
| UNSUBSCRIBE | 10     | Cliente → Servidor | Requisição de cancelamento de <i>subscribe</i> |
| UNSUBACK    | 11     | Cliente ← Servidor | ACK de cancelamento de <i>subscribe</i>        |
| PINGREQ     | 12     | Cliente → Servidor | Requisição PING                                |
| PINGRESP    | 13     | Cliente ← Servidor | Resposta PING                                  |
| DISCONNECT  | 14     | Cliente → Servidor | Solicitação de desconexão                      |
| Reservado   | 15     | Proibido           | Reservado                                      |

### 11.3.2. Protocolos de descoberta de serviços

A alta escalabilidade do IoT requer um mecanismo de gerenciamento de recursos que seja capaz de se registrar e descobrir recursos e serviços de forma autoconfigurada, eficiente e dinâmica. Os protocolos mais dominantes nesta área são o DNS de *multicast* (mDNS) e DNS *Service Discovery* (DNS-SD), que podem descobrir recursos e serviços oferecidos pelos dispositivos IoT.

Embora esses dois protocolos tenham sido projetados originalmente para dispositivos ricos em recursos, existem estudos de pesquisa que adaptam versões leves deles para ambientes IoT.

### 11.3.3. Protocolos de infraestrutura

Os protocolos de infraestrutura são necessários para estabelecer a comunicação subjacente necessária para as aplicações IoT. O roteamento é um elemento-chave da infraestrutura IoT e muitos outros parâmetros desses sistemas, como confiabilidade, escalabilidade e desempenho, que dependem fortemente dessa tecnologia. Portanto, há uma necessidade de mais investigação sobre melhorias e otimizações de protocolos de roteamento para atender aos requisitos de IoT.

RPL, 6LowPAN, IEEE 802.15.4, BLE, EPCglobal, LTE-A, Z-Wave, são protocolos de infraestrutura de uma rede de IoT. Cada um contém particularidades, vantagens e desvantagens. Neste trabalho, dar-se-á uma breve descrição dos protocolos 6LowPAN e IEEE 802.15.4.

#### 11.3.3.1. 6LoWPAN

O acrônimo 6LoWPAN significa IPv6 *Over Low Power Wireless Personal Network*. Ele foi criado com a intenção de manter as especificações que nos permitem usar o IPv6 em redes IEEE 802.15.4, pois o IPv6 suporta uma alta quantidade de endereçamentos de dispositivos, porém seu tamanho é de 128 *bits*, o que gera um grande problema, pois se o dispositivo tiver uma baixa capacidade de memória e baixa potência ele não o suportará.

O 6LoWPAN contém RFC (*Request For Comments*). Cada RFC define *standards* de métodos, comportamentos, pesquisas ou inovações capazes de definir a compressão, encapsulação e fragmentação do cabeçalho dos pacotes IPv6 em *frames* IEEE 802.15.4, permitindo que os mesmos sejam enviados e recebidos nessas redes [Ferreira, 2013].

Este protocolo se fundamenta na concepção de que a Internet é construída em IP, ou seja, cada dispositivo deverá possuir um IP fazendo, assim, parte da *Internet of Things* (IoT).

Cada rede de 6LoWPAN compartilha o mesmo prefixo de endereço. A capacidade total de cada rede é de 64.000 dispositivos, devido aos limites impostos pelo endereçamento utilizado pelo IEEE 802.15.4, que usa 16 *bits* de endereços para cada dispositivo na rede, obtendo uma identificação IPv6 única. O 6LoWPAN, sem dúvida, apresenta-se como a melhor alternativa para a integração das WPANs à Internet e às redes IP.

#### 11.3.3.2. IEEE 802.15.4

O protocolo IEEE 802.15.4, mais conhecido como *ZigBee*, foi criado pelo IEEE em parceria com a *ZigBee Alliance*. Foi desenvolvido com a intenção de disponibilizar uma rede de baixa potência de operação, que acarreta em um baixo consumo de energia nos dispositivos, prolongando a vida útil das baterias desses dispositivos. IEEE 802.15.4 é um padrão que especifica a camada física e efetua o controle de acesso para redes sem fio pessoais de baixas taxas de transmissão. A rede tem mais utilidade em dispositivos que não precisam de taxas de transmissão de dados altíssimas.

Quando o protocolo IEEE 802.15.4 foi criado, suas principais aplicações eram: controle remoto e automação. Hoje em dia, além dessas duas finalidades, ele também é utilizado nas áreas de redes e de compartilhamento de dados. Suas principais características são:

- Diferentes frequências de operação e taxa de dados: 868 MHz e 20 Kbps; 915 MHz e 40 Kbps; 2.4 GHz e 250 Kbps;
- Um mesmo dispositivo pode executar diferentes papéis em uma mesma rede;
- Configurações em diversas topologias de rede;
- Habilidade de se auto-organizar e auto-reestruturar – *self-organizing* e *self-healing*;
- Número elevado de dispositivos conectados à rede (máximo de 65.535 dispositivos por cada equipamento coordenador);

- Alta durabilidade da bateria dos dispositivos;
- A capacidade de se comunicar de forma transparente com outros sistemas, ou seja, interoperabilidade.

As comunicações por *ZigBee* são feitas na faixa das frequências ISM (*Industrial Scientific and Medical*), são elas: 2.4 Ghz (mundialmente), 915 Mhz (na América) e 868 Mhz (na Europa). Nesse contexto, a taxa de transferência dos dados é de até 250kbps na frequência de 2.4 Ghz operando com 16 canais; 40 kbps na frequência de 915 Mhz operando com 10 canais; 20 kbps na frequência de 868 Mhz operando com 1 canal [DesmontaCia, 2011].

O padrão IEEE 802.15.4 foi fragmentado em duas camadas: uma desenvolvida pela IEEE (camada inferior) e outra pela *ZigBee Alliance* (camada superior), como pode ser observado na Tabela 11.7.

**Tabela 11.7. Camadas da tecnologia Zigbee [Vasques, 2010].**

| Usuário         | Aplicação                                         |
|-----------------|---------------------------------------------------|
| ZigBee Alliance | Suporte a Aplicação<br>Rede (NWK)/Segurança (SSP) |
| IEEE 802.15.4   | MAC<br>PHY                                        |

Em redes de comunicações que utilizam o protocolo *ZigBee* o dispositivo fica por longos períodos sem se comunicar com o outro. Seu tempo de acesso conectado é cerca de 30 ms. Por conta dessa característica o protocolo IEEE 802.15.4 é mais econômico em relação ao consumo de energia.

#### 11.4. Análise de tráfego

As seções anteriores trouxeram uma visão geral sobre as bases da IoT, em especial dos protocolos CoAP e MQTT. Esta seção, por outro lado, tem como objetivo mostrar ao leitor na prática alguns dos conceitos vistos anteriormente, dando uma ênfase maior a análise de tráfego dos protocolos CoAP e MQTT. Para realizar os experimentos será utilizado o microcomputador *single-board* Raspberry Pi 2 com sistema operacional Raspbian<sup>2</sup>, em sua versão com *desktop*. Entretanto, as práticas podem ser facilmente adaptadas a outros sistemas operacionais, assim como a outros microcomputadores, ficando a cargo do leitor realizar tais adaptações.

Com o intuito de se obter um cenário mais próximo de uma rede IoT real, em que grande parte das “coisas” sofrem com severas restrições de poder computacional, em alguns experimentos será utilizada a plataforma *open-source* Arduino<sup>3</sup>. Toda a análise de tráfego será feita com auxílio da ferramenta de análise de rede Wireshark<sup>4</sup>, entretanto outra ferramenta similar pode ser utilizada.

<sup>2</sup><https://www.raspberrypi.org/downloads/raspbian/>

<sup>3</sup><https://www.arduino.cc/>

<sup>4</sup><https://www.wireshark.org/>

### 11.4.1. Preparando o ambiente

Os requisitos para realizar os experimentos são descritos abaixo:

- 1 Rapberry Pi 2 ou 3 com SO Raspbian
- 1 Arduino Uno ou Mega
- 1 Arduino Ethernet Shield
- 1 Roteador *Wireless*
- Arduino IDE
- Wireshark versão 2.2.6 ou superior
- Conexão com a internet

Para a realização dos experimentos o primeiro passo é a instalação do analisador de protocolos Wireshark. No Raspbian, abra o terminal e digite o seguinte comando:

```
$ sudo apt-get install wireshark
```

Após a instalação do Wireshark, para o primeiro experimento, é necessário a instalação de um servidor CoAP. Existem diversas implementações do protocolo CoAP<sup>5</sup>, algumas bem completas, outras nem tanto. Para os experimentos a libcoap<sup>6</sup>, implementação em C do protocolo CoAP, será utilizada. Para proceder com a instalação no Raspbian, o primeiro passo é instalar as dependências, para isso, no terminal, digite o comando:

```
$ sudo apt-get install autoconf automake libtool
```

Após isso, deve-se realizar o *download* da libcoap, disponível no GitHub. Para isso, ainda no terminal, digite:

```
$ git clone https://github.com/obgm/libcoap.git
```

Após a conclusão do *download*, ainda no Raspberry, vá até o diretório criado (\$ `cd libcoap`) e execute a seguinte sequência de comandos:

```
$ ./autogen.sh
$ ./configure --disable-documentation
$ make
$ sudo make install
```

Se tudo correr bem, neste ponto a implementação libcoap estará instalada e configurada.

Para o segundo experimento será necessária a instalação de um *broker* MQTT. Assim como para o CoAP, existem diversas implementações do MQTT para diferentes plataformas. Neste experimento será utilizada uma das implementações mais famosas e estáveis do protocolo, o Mosquitto. O Mosquitto suporta as versões 3.1 e 3.1.1 do

---

<sup>5</sup><http://coap.technology/impls.html>

<sup>6</sup><https://github.com/obgm/libcoap>

MQTT e pode ser facilmente instalado no Raspberry Pi. Antes da instalação é importante certificar-se que será instalada a versão mais recente do *broker*, para isso basta abrir o terminal do seu Raspberry (ou fazer *login* via SSH) e digitar a seguinte sequência de comandos:

```
$ sudo wget http://repo.mosquitto.org/debian/mosquitto-repo.gpg.key
$ sudo apt-key add mosquitto-repo.gpg.key
$ cd /etc/apt/sources.list.d/
$ sudo wget http://repo.mosquitto.org/debian/mosquitto-stretch.list
$ sudo apt-get update
```

A palavra “stretch” no quarto comando da sequência acima refere-se à versão do Debian que está rodando no Raspberry. Antes da instalação do Mosquitto é necessário atualizar (se já não estiverem) algumas dependências. Para isso, digite:

```
$ wget http://security.debian.org/debian-security/pool/updates/main/o/
  openssl/libssl1.0.0_1.0.1t-1+deb8u6_armhf.deb
$ sudo dpkg -i libssl1.0.0_1.0.1t-1+deb8u6_armhf.deb
$ wget http://ftp.nz.debian.org/debian/pool/main/libw/libwebsockets/
  libwebsockets3_1.2.2-1_armhf.deb
$ sudo dpkg -i libwebsockets3_1.2.2-1_armhf.deb
```

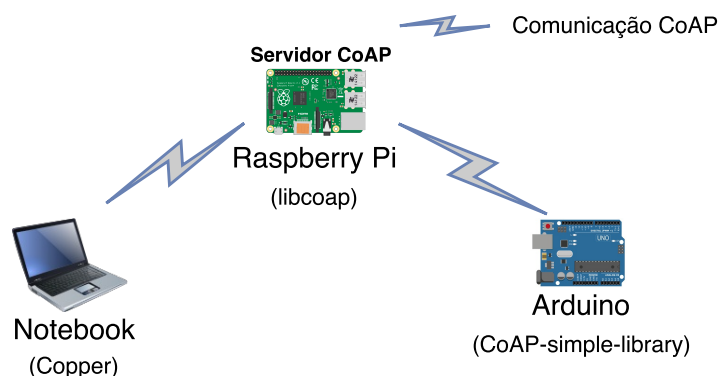
Feito isso, seu Raspberry já estará pronto para a instalação do *broker*, bastando digitar:

```
$ sudo apt-get install mosquitto
```

#### 11.4.2. Experimento 1: Análise de tráfego da comunicação CoAP

O objetivo deste experimento é mostrar como a comunicação em ambientes restritos CoAP ocorre através da análise do tráfego na rede. Toda a análise será feita através do microcomputador Raspberry Pi, com auxílio do *software* Wireshark. Na Figura 11.5 é mostrado o ambiente utilizado para a realização da prática. Entretanto, devido a necessidade de conectar o Arduino através da rede Ethernet, para a interconexão entre os dispositivos foi utilizado um roteador (abstraído na Figura 11.5).

**Figura 11.5. Cenário utilizado no experimento [Elaborada pelos autores].**



Com o servidor CoAP instalado e corretamente configurado, para que seja possível verificar como a comunicação ocorre, a instalação de um cliente CoAP é necessária. Existem diversos clientes CoAP para diferentes plataformas, dentre eles pode-se citar a Aneska<sup>7</sup> para plataforma Android, a própria libcoap (projetada para dispositivos com recursos computacionais limitados) e o Copper<sup>8</sup> (extensão para o Firefox que habilita o acesso aos recursos CoAP diretamente do navegador). Além dessas, existem diversas outras implementações deste protocolo. Com o intuito de tornar a comunicação mais próxima da prática em uma rede IoT, uma implementação simples do protocolo CoAP para a plataforma Arduino será utilizada como cliente. Para isso deve-se baixar a biblioteca CoAP-simple-library disponibilizada no GitHub<sup>9</sup> e instalá-la no Arduino IDE (a própria página web do Arduino disponibiliza um tutorial passo a passo de como fazer a instalação<sup>10</sup>). Todos os exemplos encontrados na biblioteca são direcionados para a utilização de uma placa Arduino em conjunto com uma *Arduino Ethernet Shield*. Entretanto, como as implementações do CoAP para Arduino ainda são bastante limitadas, o experimento não se restringirá a esta plataforma. Para analisar outros aspectos relevantes da comunicação, a extensão Copper, para Firefox, será utilizada em um *notebook*.

No Raspbian, para iniciar a captura com o Wireshark abra o terminal e digite `sudo wireshark`, após isso uma tela como a da Figura 11.6 deve ser mostrada. Escolha a interface na qual o Raspberry está conectado na rede e inicie a captura de pacotes.

Após iniciar a captura de pacotes no Wireshark abra novamente o terminal, navegue até o diretório libcoap (criado no momento da instalação) e digite o seguinte comando: `cd libcoap/examples`. Em seguida, para iniciar o servidor coap digite o seguinte comando: `./coap-server`, este comando irá iniciar o servidor coap de exemplo. Por padrão o servidor é iniciado na porta 5683.

No servidor de exemplo da libcoap existem alguns “recursos” que podem ser requisitados. No *notebook* abra o Firefox e, caso ainda não tenha a extensão Copper, siga os procedimentos de instalação de *add-ons*<sup>11</sup> no navegador e instale-a. Feito isso, na barra de endereços digite: `coap://IPdoRasp:5683/`, em que “IPdoRasp” deve ser substituído pelo endereço IP da *interface* utilizada no Raspberry. Feito isso, deve aparecer uma tela como a da Figura 11.7. Vale ressaltar que para a comunicação ocorrer, todos os nós devem estar na mesma sub-rede.

O primeiro recurso a ser explorado é a descoberta de serviços e recursos, para isso *click* em *core*, a barra de endereços deve mudar para algo como `coap://192.168.0.2:5683/.well-known/core`, em seguida *click* em GET. Feito isso, todos recursos disponíveis no servidor serão mostrados, neste caso os recursos disponíveis são o “/time”, que retorna a data e a hora local do servidor, o “/async”, que simula uma requisição/resposta assíncrona e o “/”, que simplesmente retorna algumas informações relativas à biblioteca libcoap com uma resposta *piggy-backed*. Neste momento alguns pacotes CoAP já foram capturados pelo Wireshark, para filtrar apenas os pacotes CoAP, no Wireshark pressi-

---

<sup>7</sup><https://play.google.com/store/apps/details?id=pl.sixpinetrees.aneska>

<sup>8</sup><http://people.inf.ethz.ch/mkovatsc/copper.php>

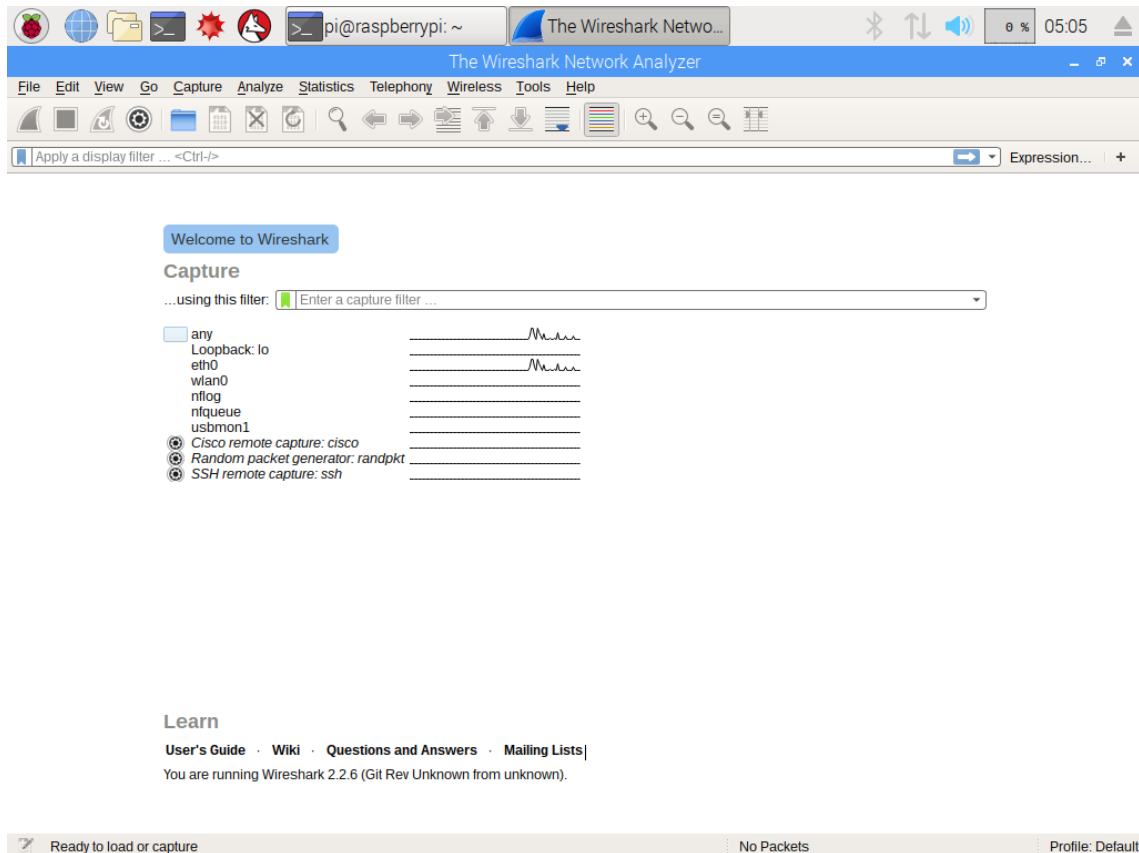
<sup>9</sup><https://github.com/hirotakaster/CoAP-simple-library>

<sup>10</sup><https://www.arduino.cc/en/Guide/Libraries>

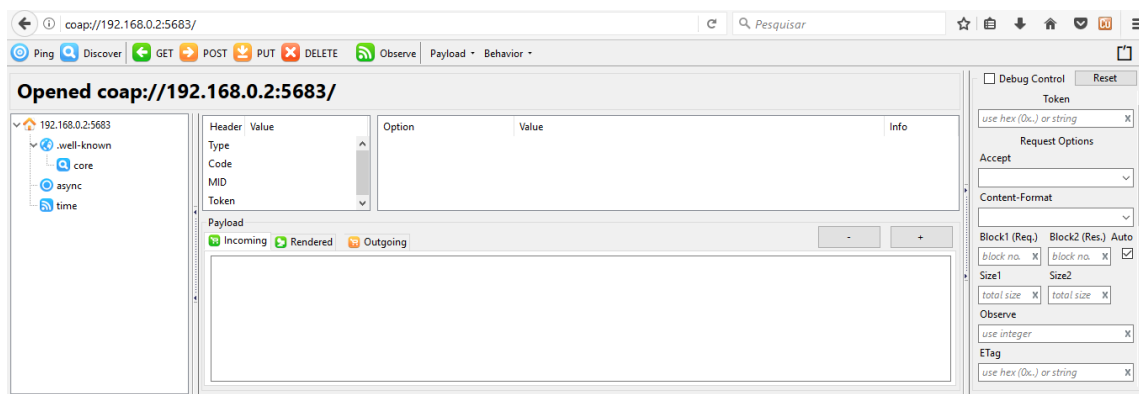
<sup>11</sup><https://support.mozilla.org/en-US/kb/find-and-install-add-ons-add-features-to-firefox>



**Figura 11.6. Captura de tela do Wireshark no Raspberry [Elaborada pelos autores].**



**Figura 11.7. Captura de tela do Copper no Firefox [Elaborada pelos autores].**

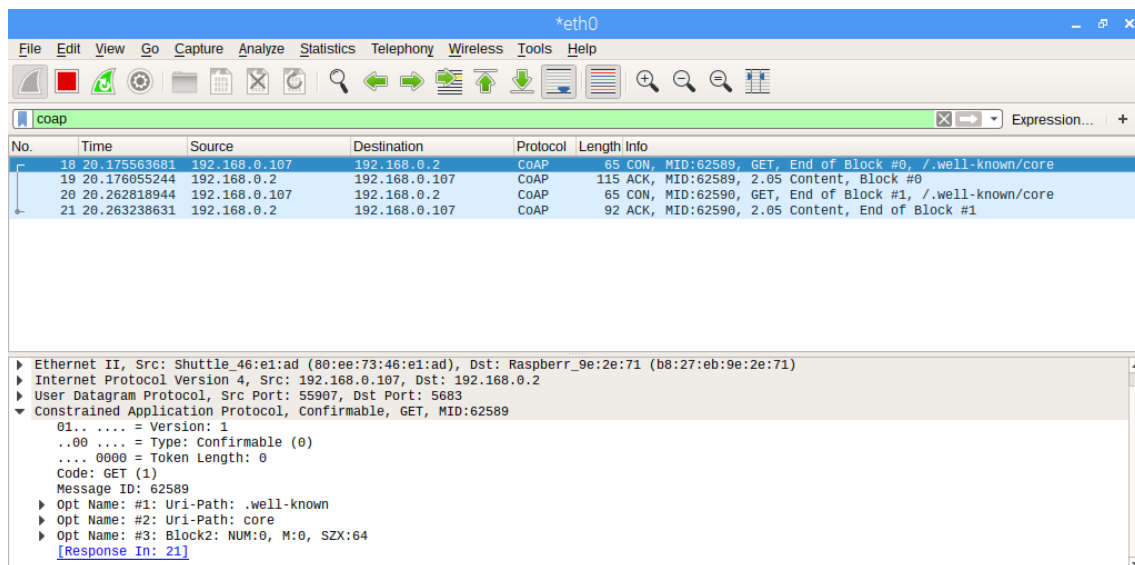


one as teclas “Ctrl+/", esse comando levará o cursor até a barra de filtros, então digite `udp.port == 5683` ou simplesmente “coap”, ambos os filtros mostraram apenas os pacotes CoAP que estão transitando pela rede. Na Figura 11.8 pode-se observar o fluxo de mensagens CoAP na requisição, feita através do método GET, em “/well-known/core”.

Alguns detalhes importantes da comunicação CoAP podem ser destacados a partir da Figura 11.8:

- Toda mensagem possui um id (MID).

**Figura 11.8. Captura de mensagens da descoberta de serviços CoAP [Elaborada pelos autores].**



| No. | Time         | Source        | Destination   | Protocol | Length | Info                                                    |
|-----|--------------|---------------|---------------|----------|--------|---------------------------------------------------------|
| 18  | 20.175563681 | 192.168.0.107 | 192.168.0.2   | CoAP     | 65     | CON, MID:62589, GET, End of Block #0, /.well-known/core |
| 19  | 20.176055244 | 192.168.0.2   | 192.168.0.107 | CoAP     | 115    | ACK, MID:62589, 2.05 Content, Block #0                  |
| 20  | 20.262818944 | 192.168.0.107 | 192.168.0.2   | CoAP     | 65     | CON, MID:62590, GET, End of Block #1, /.well-known/core |
| 21  | 20.263238631 | 192.168.0.2   | 192.168.0.107 | CoAP     | 92     | ACK, MID:62590, 2.05 Content, End of Block #1           |

```

Ethernet II, Src: Shuttle_46:e1:ad (80:ee:73:46:e1:ad), Dst: Raspberr_9e:2e:71 (b8:27:eb:9e:2e:71)
Internet Protocol Version 4, Src: 192.168.0.107, Dst: 192.168.0.2
User Datagram Protocol, Src Port: 55907, Dst Port: 5683
Constrained Application Protocol, Confirmable, GET, MID:62589
  01.. .... = Version: 1
  ..00 .... = Type: Confirmable (0)
  .... 0000 = Token Length: 0
  Code: GET (1)
  Message ID: 62589
  Opt Name: #1: Uri-Path: .well-known
  Opt Name: #2: Uri-Path: core
  Opt Name: #3: Block2: NUM:0, M:0, SZX:64
  [Response In: 21]
  
```

- Toda mensagem do tipo CON espera uma confirmação de recebimento (ACK) com o mesmo ID da mensagem enviada.
- Uma mensagem ACK pode, além de confirmar o recebimento da requisição, transportar o conteúdo da resposta.
- Uma mensagem CoAP pode ser dividida em blocos caso a requisição ou resposta sejam grandes.

Para realizar uma comunicação simples entre o Arduino e o servidor CoAP será utilizado o código de exemplo “coaptest” da biblioteca CoAP-simple-library. No Arduino IDE, após a correta instalação da biblioteca, será possível, no menu “Arquivo->Exemplos->CoAP simple library”, encontrar um código de exemplo chamado “coaptest”. Para que seja possível observar a comunicação entre o Arduino e o Raspberry através do protocolo CoAP é necessário realizar algumas modificações no código de exemplo. Para realizar a comunicação de forma adequada, na linha 54, altere o “(XXX, XXX, XXX, XXX)” pelo endereço IP do servidor CoAP. A função *loop* deve ficar como no Algoritmo 1. Após as alterações, carregue o código para o microcontrolador e, com o Arduino devidamente conectado à rede e à Arduino Ethernet Shield, observe o tráfego de mensagens CoAP no Wireshark.

No Arduino IDE é possível ver as respostas do servidor através do monitor serial. Para abrir o monitor serial, no Arduino IDE, basta clicar na lupa localizada no canto superior direito da tela ou através do atalho “CTRL+SHIFT+M”. Para que não ocorra nenhum erro na interpretação dos dados é importante certificar-se de que o *baud rate* está em 9600. Na Figura 11.9a é mostrado o resultado no serial monitor, nele pode-se perceber que a comunicação ocorreu da forma esperada, já que a cada dez segundos o servidor CoAP está respondendo a requisição do Arduino com a data e hora. Esta comunicação pode ser vista por outra perspectiva, através do Wireshark, mostrado na Figura 11.9b. Enquanto no serial

### Função *loop* do coaptest

```

52
53 void loop() {
54     //Envia requisicao
55     int msgid = coap.get(IPAddress(192, 168, 0, 2),5683, "time");
56
57     //Aguarda 10 segundos
58     delay(10000);
59     coap.loop();
60 }

```

Algoritmo 1: Função *loop* do exemplo *coaptest* após modificações.

monitor existem cinco resultados de data e hora, no Wireshark são mostradas dez mensagens CoAP. Na mensagem número 8 o Arduino, com IP 192.168.0.100, realiza a primeira requisição através do método GET com uma mensagem do tipo CON (que necessita de uma confirmação), em seguida o servidor CoAP do Raspberry, com IP 192.168.0.2, envia uma resposta *piggy-backed*, na mensagem número 9. Este padrão se repete durante todas as outras requisições.

**Figura 11.9. Captura de mensagens CoAP na comunicação entre o Arduino e o Raspberry [Elaborada pelos autores].**



(a) Com confirmação

| No. | Time         | Source        | Destination   | Protocol | Length | Info                                      |
|-----|--------------|---------------|---------------|----------|--------|-------------------------------------------|
| 8   | 1.164534792  | 192.168.0.100 | 192.168.0.2   | CoAP     | 60     | CON, MID:16807, GET, /time                |
| 9   | 1.164842344  | 192.168.0.2   | 192.168.0.100 | CoAP     | 65     | ACK, MID:16807, 2.05 Content (text/plain) |
| 10  | 11.170748590 | 192.168.0.100 | 192.168.0.2   | CoAP     | 60     | CON, MID:15089, GET, /time                |
| 11  | 11.171097600 | 192.168.0.2   | 192.168.0.100 | CoAP     | 65     | ACK, MID:15089, 2.05 Content (text/plain) |
| 12  | 21.176988638 | 192.168.0.100 | 192.168.0.2   | CoAP     | 60     | CON, MID:11481, GET, /time                |
| 13  | 21.177343013 | 192.168.0.2   | 192.168.0.100 | CoAP     | 65     | ACK, MID:11481, 2.05 Content (text/plain) |
| 16  | 31.183258061 | 192.168.0.100 | 192.168.0.2   | CoAP     | 60     | CON, MID:3114, GET, /time                 |
| 17  | 31.183638374 | 192.168.0.2   | 192.168.0.100 | CoAP     | 65     | ACK, MID:3114, 2.05 Content (text/plain)  |
| 18  | 41.189721339 | 192.168.0.100 | 192.168.0.2   | CoAP     | 60     | CON, MID:14210, GET, /time                |
| 19  | 41.190368995 | 192.168.0.2   | 192.168.0.100 | CoAP     | 65     | ACK, MID:14210, 2.05 Content (text/plain) |

(b) Sem confirmação

Outras formas de transmissão podem ser testadas de forma fácil através do Copper. O cliente CoAP para Firefox permite o envio de todos os tipos de requisições definidas para o protocolo, ficando a cargo do leitor realizar testes e analisar a coerência do fluxo de mensagens com o conteúdo teórico exposto na Seção 11.3.

### 11.4.3. Experimento 2: Análise de tráfego da comunicação MQTT

O objetivo deste experimento é mostrar como a comunicação em ambientes restritos MQTT ocorre através da análise do tráfego na rede. O Raspberry Pi será utilizado como *broker* MQTT e será o responsável pela captura do tráfego, auxiliado pelo analisador de protocolos Wireshark. Neste experimento será analisado o processo de comunicação MQTT entre um *smartphone* (com sistema operacional Android) e um Arduino, simulando um ambiente de IoT. Além disso, para analisar outros aspectos relevantes da comunicação com o protocolo MQTT, será utilizado um cliente Mosquitto em um dispositivo com o sistema operacional Windows.

Após a instalação do Mosquitto no Raspberry Pi, como descrito na subseção **Preparando o ambiente**, o servidor já estará rodando automaticamente na porta TCP padrão, 1883. O próximo passo é a preparação dos clientes. Existem diversos clientes MQTT para diferentes plataformas, não sendo, portanto, obrigatória a utilização dos mesmos clientes aqui descritos para a obtenção dos mesmos resultados. Para o envio de comandos da plataforma Android para o Arduino será utilizado o cliente Android MQTT Dash (disponível na Google Play Store<sup>12</sup>) e o cliente Arduino pubsubclient (disponível para *download* no GitHub<sup>13</sup>). No Arduino IDE, após a instalação da biblioteca pubsubclient, acesse os arquivos de exemplo e abra o exemplo *mqtt\_publish\_in\_callback* da biblioteca instalada. Altere os IPs nas linhas 23 e 24 de acordo com a faixa de endereços IP da sua rede, por exemplo, para uma rede que utiliza endereços na faixa de “192.168.0.x” é possível utilizar os valores mostrados no Algoritmo 2.

#### mqtt\_publish\_in\_callback

```
23 IPAddress ip(192, 168, 0, 100);  
24 IPAddress server(192, 168, 0, 2);
```

Algoritmo 2: Linhas 23 e 24 do exemplo *mqtt\_publish\_in\_callback* após modificações.

O endereço do “server”, na linha 24, deve ser substituído pelo endereço IP atribuído ao *broker* (neste caso o endereço do Raspberry Pi). Feito isso, inicie a captura de pacotes no Wireshark (filtrando por “mqtt”), carregue as alterações para a placa Arduino e ligue-a à rede através da *Arduino Ethernet Shield*. Neste momento, já será possível observar alguns pacotes no Wireshark, como é mostrado na Figura 11.10.

**Figura 11.10. Captura de pacotes após a conexão do Arduino na Rede [Elaborada pelos autores].**

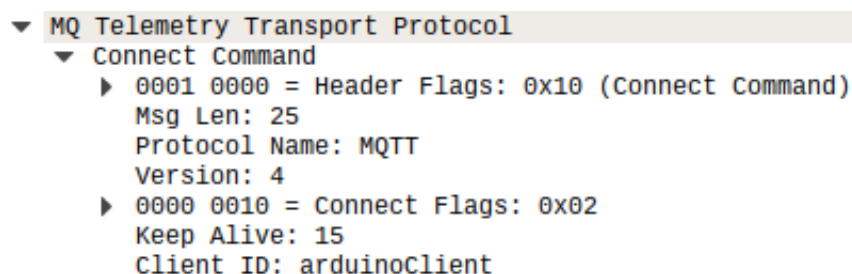
| mqtt |              |               |               |          |        |                   |
|------|--------------|---------------|---------------|----------|--------|-------------------|
| No.  | Time         | Source        | Destination   | Protocol | Length | Info              |
| 59   | 57.448455135 | 192.168.0.100 | 192.168.0.2   | MQTT     | 81     | Connect Command   |
| 61   | 57.448733780 | 192.168.0.2   | 192.168.0.100 | MQTT     | 58     | Connect Ack       |
| 62   | 57.450795030 | 192.168.0.100 | 192.168.0.2   | MQTT     | 77     | Publish Message   |
| 63   | 57.451270968 | 192.168.0.100 | 192.168.0.2   | MQTT     | 68     | Subscribe Request |
| 65   | 57.451462478 | 192.168.0.2   | 192.168.0.100 | MQTT     | 59     | Subscribe Ack     |

<sup>12</sup>[https://play.google.com/store/apps/details?id=net.routix.mqttdash&hl=pt\\_BR](https://play.google.com/store/apps/details?id=net.routix.mqttdash&hl=pt_BR)

<sup>13</sup><https://github.com/knolleary/pubsubclient>

O primeiro pacote MQTT capturado pelo Wireshark é a mensagem de solicitação de conexão. Analisando melhor esta solicitação podemos perceber algumas informações relevantes para a comunicação. A primeira informação importante que este pacote fornece é a versão do protocolo na qual a comunicação se estabelecerá. Como pode ser visto em “Version: 4”, na Figura 11.11, o cliente está solicitando comunicação com a versão 3.1.1 do protocolo MQTT, que conforme especificado em [Standard, 2014] é representado pelo número 4. Caso o *broker* não seja capaz de se comunicar com esta versão do MQTT, a solicitação de conexão é recusada. Entretanto, como pode ser visto na Figura 11.10 a conexão foi aceita (um pacote CONNACK foi recebido), indicando que o servidor é capaz de se comunicar através da versão especificada. A solicitação de conexão deve conter, ainda, outras informações importantes, como usuário, senha, *Client ID*, *Keep Alive*, entre outras. No exemplo não foi utilizado usuário e senha, entretanto, em uma implementação prática a utilização destes campos é extremamente recomendada. Ainda na Figura 11.11, é possível ver o campo *Client ID*, onde é informada uma identificação para o dispositivo solicitante da conexão (neste caso *ArduinoClient*), esta deve ser única para que não ocorram conflitos. Já no campo *Keep Alive* é informado o tempo máximo, em segundos, entre o ponto em que o cliente acaba de transmitir um pacote de controle e o ponto em que ele começa a enviar o próximo. Em caso de não existir nenhuma informação a ser transmitida, o cliente deve enviar periodicamente um pacote PINGREQ, de modo que o tempo não seja excedido. Caso contrário o servidor irá interpretar como uma falha de rede e desconectará o cliente.

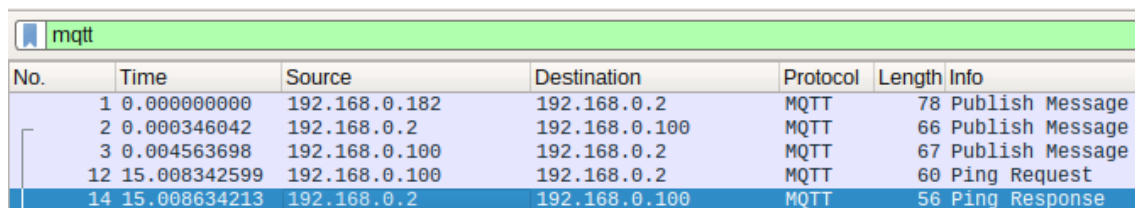
**Figura 11.11. Pacote de solicitação de conexão do Arduino [Elaborada pelos autores].**



No dispositivo Android, já com o MQTT Dash instalado, o primeiro passo é criar a conexão com o servidor. Para isso, abra o *app* e clique no botão (+), no canto superior direito, em seguida preencha as informações necessárias, sendo elas nome da conexão (identificada pelo campo “name”) e endereço do *broker* (identificado pelo campo “Address”). Os outros dados necessários, como porta e *Client ID* já são preenchidos automaticamente com valores padrão. Existe ainda a possibilidade de informar o nome de usuário e senha, entretanto como estes não foram configurados no servidor deverão ficar em branco. Neste momento a conexão com o *broker* já pode ser estabelecida. Para testar a comunicação entre o cliente Android e o cliente Arduino será feita uma publicação através do MQTT Dash no tópico “inTopic”, em que o cliente Arduino está inscrito. No exemplo utilizado, o cliente Arduino irá republicar todas as informações que chegarem no tópico “inTopic” para um tópico chamado “outTopic”. Para ver isto correndo na prática, será criado um botão no MQTT Dash que a cada vez que é tocado faz uma publicação no

tópico “inTopic”. Para criar o botão basta acessar a conexão criada e clicar novamente no botão (+), em seguida algumas opções serão exibidas, selecione a opção “Switch/button” e dê um nome ao botão, em seguida, no campo “Topic (pub)” digite o tópico no qual o cliente Arduino está inscrito, neste caso “inTopic”. Feito isso, clique no botão salvar (canto superior direito) e em seguida pressione o botão criado. Neste momento a informação “0” deverá ser publicada no tópico “inTopic”. Na Figura 11.12 é mostrado o fluxo de pacotes MQTT gerado por este comando.

**Figura 11.12. Comunicação entre o cliente Android e o cliente Arduino intermediada pelo *broker* [Elaborada pelos autores].**



| No. | Time         | Source        | Destination   | Protocol | Length | Info            |
|-----|--------------|---------------|---------------|----------|--------|-----------------|
| 1   | 0.000000000  | 192.168.0.182 | 192.168.0.2   | MQTT     | 78     | Publish Message |
| 2   | 0.000346042  | 192.168.0.2   | 192.168.0.100 | MQTT     | 66     | Publish Message |
| 3   | 0.004563698  | 192.168.0.100 | 192.168.0.2   | MQTT     | 67     | Publish Message |
| 12  | 15.008342599 | 192.168.0.100 | 192.168.0.2   | MQTT     | 60     | Ping Request    |
| 14  | 15.008634213 | 192.168.0.2   | 192.168.0.100 | MQTT     | 56     | Ping Response   |

Como pode ser visto na Figura 11.12, a mensagem foi inicialmente publicada do *smartphone* (IP 192.168.0.182) para o servidor (IP 192.168.0.2), em seguida o servidor enviou a mensagem para todos os inscritos (*subscribers*) do tópico “inTopic”, que neste caso é apenas o cliente Arduino (IP 192.168.0.100). O cliente Arduino, por sua vez, republicou a mensagem para o servidor informando o tópico “outTopic”. Entretanto, como não havia nenhum inscrito neste tópico nenhum novo pacote do tipo PUBLISH foi enviado pelo *broker*, a não ser uma resposta a um pacote *Ping Request*, enviado pelo cliente Arduino aproximadamente quinze segundos após o envio do último pacote de controle (devido ao tempo de *Keep Alive* visto anteriormente). Perceba que em nenhuma das mensagens houve confirmação de recebimento, isso ocorre porque o nível de qualidade de serviço utilizado na comunicação foi o QoS 0 (*Fire and Forget*). Para ilustrar os três níveis de QoS providos pelo protocolo MQTT foi instalado um cliente Moquitto em um ambiente Windows.

Na plataforma Windows, os procedimentos de instalação são um pouco diferentes. Após a instalação do arquivo executável, que pode ser baixado no site oficial do *broker*<sup>14</sup>, será necessário seguir os procedimentos especificados em um arquivo chamado “readme-windows.txt”, localizado no diretório onde o *software* foi instalado. Após a conclusão do processo de instalação o cliente MQTT já estará pronto para ser utilizado. Para testar os três níveis de QoS providos pelo MQTT foram enviados três pacotes do tipo PUBLISH para o *broker* através do comando “mosquitto\_pub -q <QoS> -h <endereço> -t <tópico> -m <mensagem>”. Para o envio do primeiro pacote com o nível de QoS mais baixo (QoS 0), por exemplo, no Prompt de Comando do Windows, dentro do diretório de instalação do Mosquitto, foi utilizado o comando “mosquitto\_pub -q 0 -h 192.168.0.2 -t teste -m “Teste de QoS”. Este comando está dizendo explicitamente para o cliente Windows enviar um pacote MQTT do tipo PUBLISH com nível de QoS 0 para o endereço “192.168.0.2” no tópico “teste” com a mensagem “Teste com QoS 0”. Além disso, implicitamente outras informações estão sendo enviadas, como a porta TCP, que em caso de não ser informada de forma explícita é utilizada a porta padrão 1883. Todos os parâmetros

<sup>14</sup><https://mosquitto.org/download/>



aceitos pelo *broker* podem ser visto através do comando “mosquitto\_pub -help”. O fluxo gerado pelo envio de um pacote PUBLISH para o mesmo tópico, no mesmo *broker* e com a mesma mensagem, mudando apenas o nível de QoS é mostrado na Figura 11.13.

**Figura 11.13. Captura de mensagens CoAP na comunicação entre o Arduino e o Raspberry [Elaborada pelos autores].**

| No. | Time        | Source        | Destination   | Protocol | Length | Info            |
|-----|-------------|---------------|---------------|----------|--------|-----------------|
| 8   | 1.195575573 | 192.168.0.140 | 192.168.0.2   | MQTT     | 93     | Connect Command |
| 10  | 1.196541198 | 192.168.0.2   | 192.168.0.140 | MQTT     | 58     | Connect Ack     |
| 11  | 1.198460833 | 192.168.0.140 | 192.168.0.2   | MQTT     | 78     | Publish Message |
| 12  | 1.198464739 | 192.168.0.140 | 192.168.0.2   | MQTT     | 60     | Disconnect Req  |

(a) QoS 0 (no máximo uma entrega)

| No. | Time        | Source        | Destination   | Protocol | Length | Info            |
|-----|-------------|---------------|---------------|----------|--------|-----------------|
| 10  | 8.256484476 | 192.168.0.140 | 192.168.0.2   | MQTT     | 93     | Connect Command |
| 12  | 8.257487340 | 192.168.0.2   | 192.168.0.140 | MQTT     | 58     | Connect Ack     |
| 13  | 8.260650778 | 192.168.0.140 | 192.168.0.2   | MQTT     | 80     | Publish Message |
| 14  | 8.260892340 | 192.168.0.2   | 192.168.0.140 | MQTT     | 58     | Publish Ack     |
| 15  | 8.265063538 | 192.168.0.140 | 192.168.0.2   | MQTT     | 60     | Disconnect Req  |

(b) QoS 1 (pelo menos uma entrega)

| No. | Time        | Source        | Destination   | Protocol | Length | Info             |
|-----|-------------|---------------|---------------|----------|--------|------------------|
| 8   | 8.504837706 | 192.168.0.140 | 192.168.0.2   | MQTT     | 93     | Connect Command  |
| 10  | 8.506189424 | 192.168.0.2   | 192.168.0.140 | MQTT     | 58     | Connect Ack      |
| 11  | 8.508007185 | 192.168.0.140 | 192.168.0.2   | MQTT     | 80     | Publish Message  |
| 12  | 8.508240466 | 192.168.0.2   | 192.168.0.140 | MQTT     | 58     | Publish Received |
| 13  | 8.509729529 | 192.168.0.140 | 192.168.0.2   | MQTT     | 60     | Publish Release  |
| 14  | 8.509901195 | 192.168.0.2   | 192.168.0.140 | MQTT     | 58     | Publish Complete |
| 15  | 8.511656924 | 192.168.0.140 | 192.168.0.2   | MQTT     | 60     | Disconnect Req   |

(c) QoS 2 (exatamente uma entrega)

Na primeira situação, em que o nível de QoS é igual a zero (Figura 11.13a), é possível perceber que nenhum retorno do *broker* foi recebido em relação ao comando *publish*. Após a conexão ser estabelecida com o servidor (ser recebido o pacote *Connect Ack*) o cliente Windows simplesmente envia o pacote a ser publicado e, em seguida, solicita a desconexão, já que nenhum retorno é esperado. Já na segunda situação (Figura 11.13b), com nível de QoS igual a um, é esperado um pacote de confirmação de recebimento da publicação (*Publish Ack*). Nesta situação, caso o transmissor não receba uma mensagem de confirmação dentro de um *timeout* padrão, o pacote *publish* é retransmitido periodicamente até que a confirmação seja recebida. Por fim, quando o nível de QoS mais alto é utilizado (QoS 2) é garantida a entrega de um, e somente um, pacote do tipo PUBLISH. Neste caso a confirmação de entrega é feita em três partes. Como ilustrado na Figura 11.13c, após o envio do pacote de publicação, o transmissor irá aguardar o recebimento da primeira confirmação (*Publish Received*), ao receber este pacote, o transmissor envia um segundo pacote (*Publish Release*) e novamente aguarda a confirmação do receptor, que deve responder com um pacote *Publish Complete*, indicando que a publicação foi bem sucedida.

## 11.5. Conclusão

Este capítulo apresentou as definições e motivações para as pesquisas de *Internet of Things* (IoT), bem como um breve panorama histórico. O foco ficou para o monitoramento prático das redes de Internet das Coisas. Foram utilizados *softwares sniffers* para a análise de tráfego em tempo real de um sistema de IoT, este prototipado usando Arduinos e Raspberry.

Os trabalhos futuros poderão contemplar cenários mais desafiadores e realísticos, com vários equipamentos transmitindo entre si, bem como novas práticas e estratégias para análise de tráfego. Outros protocolos utilizados em sistemas IoT, alguns discutidos superficialmente neste trabalho, também poderão ser abordados com mais detalhes.

## Referências

- [Al-Fuqaha et al., 2015] Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., and Ayyash, M. (2015). Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials*, 17(4):2347–2376.
- [Ashton, 2009] Ashton, K. (2009). In the real world, things matter more than ideas. *RFID Journal*.
- [Chen et al., 2014] Chen, W.-J., Gupta, R., Lampkin, V., Robertson, D. M., Subrahmanyam, N., et al. (2014). *Responsive Mobile User Experience Using MQTT and IBM MessageSight*. IBM Redbooks.
- [Cisco, 2017] Cisco (2017). Global mobile data traffic forecast update, 2016-2021. [http://www.cisco.com/assets/sol/sp/vni/forecast\\_highlights\\_mobile/index.html](http://www.cisco.com/assets/sol/sp/vni/forecast_highlights_mobile/index.html).
- [Desai, 2015] Desai, P. (2015). *Python Programming for Arduino*. Packt Publishing Ltd.
- [DesmontaCia, 2011] DesmontaCia (2011). Zigbee ou ieee 802.15.4 – conheça a tecnologia a fundo. <https://desmontacia.wordpress.com/2011/02/22/zigbee-ou-ieee-802-15-4-conheca-a-tecnologia-a-fundo>.
- [Ferreira, 2013] Ferreira, S. (2013). Ipv6 nas redes de sensores sem fio. [https://comun.rcaap.pt/bitstream/10400.26/17524/2/SergioFerreira\\_Tese\\_Mestrado\\_2012-2013.pdf](https://comun.rcaap.pt/bitstream/10400.26/17524/2/SergioFerreira_Tese_Mestrado_2012-2013.pdf).
- [Gubbi et al., 2013] Gubbi, J., Buyya, R., Marusic, S., and Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660.
- [Joshi and Kaur, 2015] Joshi, M. and Kaur, B. P. (2015). Coap protocol for constrained networks. *International Journal of Wireless and Microwave Technologies*, 5(6):1–10.
- [Kuladinithi et al., 2011] Kuladinithi, K., Bergmann, O., Pötsch, T., Becker, M., and Görg, C. (2011). Implementation of coap and its application in transport logistics. *Proc. IP+ SN, Chicago, IL, USA*.



[Shelby et al., 2014] Shelby, Z., Hartke, K., and Bormann, C. (2014). The constrained application protocol (coap).

[Standard, 2014] Standard, O. (2014). Mqtt version 3.1.1. <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>.

[Vasques, 2010] Vasques, B. L. R. P. (2010). Zigbee. [https://www.gta.ufrj.br/grad/10\\_1/zigbee/introducao.html](https://www.gta.ufrj.br/grad/10_1/zigbee/introducao.html).