

# Infrastructure and routing for connected objects

**Eugen Dedu**

**Maître de conférences (associate professor)**

Université Marie et Louis Pasteur, UFR STGI

Master 1 IoT

Montbéliard, France

September 2025

<https://dedu.fr>

[eugen.dedu@univ-fcomte.fr](mailto:eugen.dedu@univ-fcomte.fr)

# Organisation of the module

12h CM, 12h TD, 24h labs

Goal: upper-layer communication protocols in IoT,  
radio technology excluded



**Eugen Dedu**

12h CM, 8h TD, 6h labs

written exam of 1h30, w=.5



**Hakim Mabed**

2h TD, 9h labs

IoT modelling,  
simulation on PacketTracer,  
validation



**Dominique Dhoutaut**

2h TD, 9h labs

project to defend  
(project with oral defence), w=.5

MQTT on real sensor/actuator hw,  
RaspberryPI & Arduino,  
data shown in a Web page

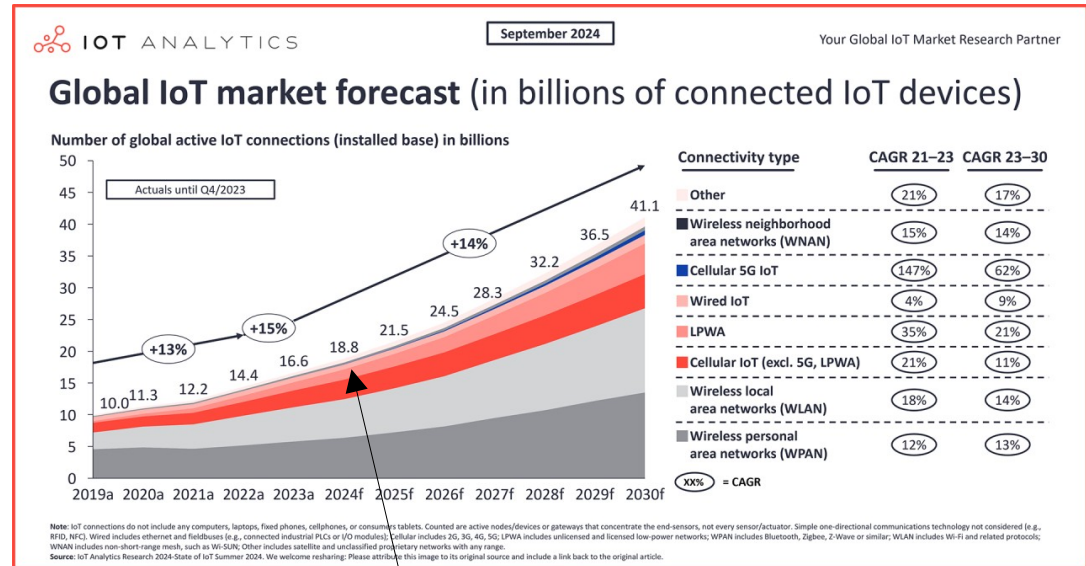
- Communication protocols in IoT:
- app-layer pr.: MQTT, STOMP, AMQP, CoAP
    - routing protocol: RPL
- Labs on a GNU/Linux machine

# Organisation of the module

- Does not treat low-power low-level wireless IoT technologies, which will be presented next semester in *Radio networks* (together with their protocol, if any):
  - Sigfox, LoRa (physical layer) and LoRaWAN (communication protocol and system architecture)
- My TP/labs: bring your laptop with linux VM (preferably debian/ubuntu)
- My exam: on paper, no document authorised, whole CM + what I said + questions from lab

# IoT revolution?

- The Internet revolutionized how people communicate and work together. But the next wave of the Internet is not about people. It's about intelligent, connected devices
- The IoT's opportunity and challenge will be to connect them in a meaningful way to deliver truly distributed machine-to-machine (M2M) applications
- 3 times more objects than users
- Where are all these devices? They are part of the fabric of everyday life. In fact, you own many of them! Recent cars use more than 100 processors. Smart devices pervade industrial systems, hospitals, houses, transportation systems, and more. Today, these systems are weakly connected, but that will quickly change.
- Related narrower concept: M2M Infrastructure & Routing for IoT (machine to machine) communication



We are here

# Useful features of an IoT protocol

- Provide the communication pattern needed by the application (described later)
- Simplicity (cf. STOMP)
- Feature-richness (cf. AMQP)
- Popularity (cf. MQTT)
- Specific functions in IoT: device data collection
- ...

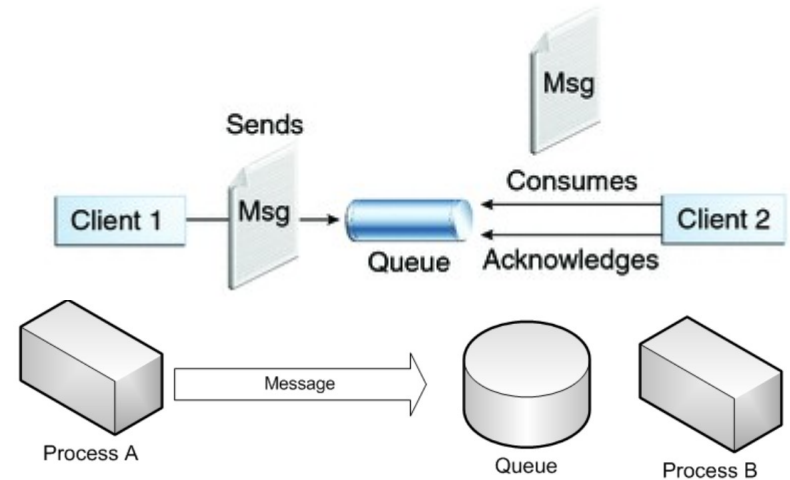
Communication models/patterns  
– how is data transported on the network –

# Communication patterns in IoT – request-response pattern

- **One-to-one**: a machine asks another machine, which answers to it
- Allows a machine to get information in real-time from another machine, like HTTP, RMI, RPC
  - examples: a client asks a sensor some information (e.g. temperature)
- Properties:
  - receiver must be alive when request is sent (phone call/videoconference vs SMS/e-mail)
  - tight coupling: client must know the server address (for ex. in emergency call the caller does not know the precise destination/police car etc.) and both need to discuss the same language (interoperability)
  - sender needs to wait the response; if the response is slow to be generated/collected, it is possible to return partial results to show progress

# Communication patterns in IoT – asynchronous messaging pattern

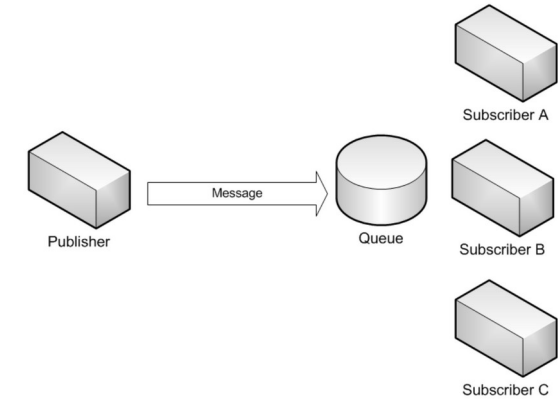
- Also known as “fire-and-forget” exchange
- Loose coupling: the sender puts a message in message queue (event queue) and does not require an immediate response to continue processing, like e-mail (except that it is created by software, for communication between software components and applications, not by people)
- Queues retain all messages sent to them until the messages are consumed or expire
- Request-response model need that both parties be available, whereas queuing takes the message and sends it to destination when it becomes available
- Needs to know the destination
- Read [Why the Internet of Things Needs Messaging](#)



- The **dead letter queue** stores messages in case of:
  - the destination queue does not exist
  - queue length limit exceeded
  - message length limit exceeded
- Allows developers to look for common patterns and potential software problems

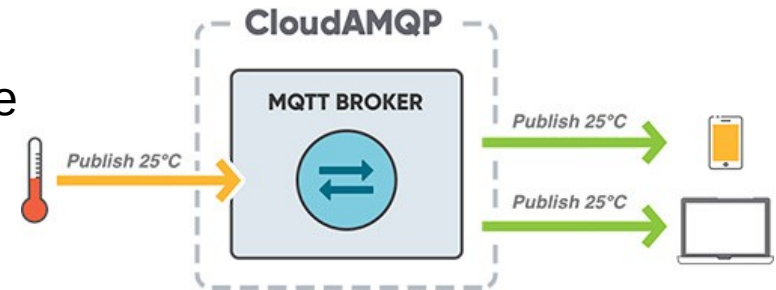
# Communication patterns in IoT – publish-subscribe (pub-sub) pattern

- This pattern consists of clients (publishers of or subscribers to information/data) communicating with a server (“broker”)
  - broker (*représentant*) = “one who transacts business for another; an agent” (Webster 1913)
  - subscribers subscribe to broker to one (or more) topic(s)
  - the publishers send to broker their data along with a topic (its specific function, e.g. temperature)
  - the broker retransmits it to all subscribers of that topic => subscribers receive only messages of the subscribed topic
  - e.g. a smartphone which receives data from all brightness sensors
- **One-to-many**: allows efficient **mass distribution** of data to multiple consumers, allowing system monitoring
- Properties:
  - indirect communication: senders do not know their receivers, but still need to know the broker (be configured with its address)!
  - asynchronous (sender does not wait for all receivers to receive)
  - works in heterogeneous platforms, easier to change or update
- Clients only interact with a broker, but a system may contain several brokers (e.g. they process different topics)



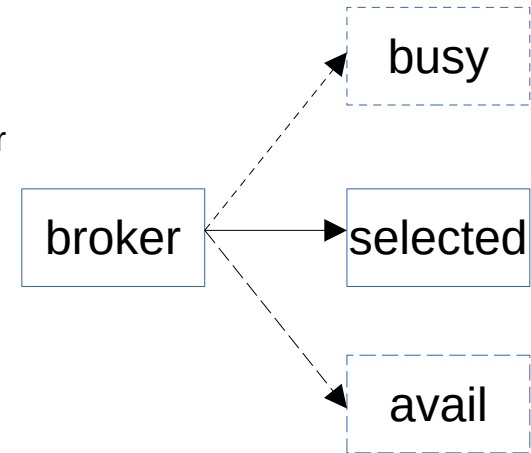
# Pub-sub pattern, an application

- Consider  $n$  sensors and  $m$  readers. Do all the sensors need to listen continuously the network, to answer the requests? Do they answer each reader individually?
- Here, the pub-sub model fits very well
- Additionally, sensors do not need to process their data, they just send their data to broker, which transforms it in temperature or the useful format
- The broker may be on Internet (cloud)
- A temperature sensor in a greenhouse could send the temperature to the broker in a 15-minute interval (about 96 times a day) instead of processing the data in the greenhouse and be connected all the time



# Communication patterns in IoT – fan-out pattern

- To fan out: "to spread out over a wide area; If a group of people fan out, they move in different directions from a single point" (cambridge)
- A list of tasks and several processors/machines
- The message is delivered to **one worker only** (from a pool of workers) in a round-robin fashion (contrarily to pub-sub, where the message is sent to **all** subscribers)
- Analogy with firemen: when someone informs the centre (broker) about a fire, the broker sends this information to a car; when another fire occurs, the server selects another car; and so on
  - people (publishers) do not need to know details about firemen cars (number and availability)
  - when a firemen car is added or removed, only the server needs to be updated
- Fan-in allows to collect the results, if appropriate
- Pub/sub is used for wide message distribution (data diffusion), whereas fan-out is more about resource allocation
- Would this communication pattern be useful in IoT context?



# Communication patterns in IoT – router-dealer pattern

- A server (the market) and clients (dealers/traders)
  - traders place buy and sell orders by sending relevant 'order' messages to the Market
  - the Market responds immediately with an 'acknowledge' message
  - sometime later when an order is fulfilled the Market sends 'order complete' messages to all involved traders
- A game server lets clients join a room until it is full. It keeps track of each client ID in the room. When room is full, it loops over each client ID within that room and sends them a "game started" message
- Responses are asynchronous (contrarily to pub-sub pattern, where the broker answers immediately)
- Each subscriber is identified uniquely through an id, and servers answers to specific clients
- This sort of behavior would be quite cumbersome to implement with pub/sub as you would need both Market and Traders to run a Publisher and a Subscriber socket to allow the 2 way communication. There would also be privacy concerns if all completed transactions were 'published' rather than sent direct to the relevant traders (trader B should not get to know that trader A bought or sold something)
- Would this communication pattern be useful in IoT context?

# Message broker

- A broker takes message from sender, do some processing if needed, and forwards it to appropriate receivers (routing)
  - examples of processing: message validation, translation to receiver's protocol, message decomposition into several messages or aggregation into one message only, transformation
- Provides reliable storage, guaranteed message delivery, transaction management (detailed later)
- If one broker only, it is a single point of failure!

## Software:

- [RabbitMQ](#), written in Erlang, supports MQTT, AMQP, STOMP etc., multiple platforms, several language bindings
- [Apache ActiveMQ](#), written in Java, supports MQTT, AMQP, STOMP etc., multiple platforms, several language bindings
- [Mosquitto](#), written in C, MQTT-only
- Many others...

# What communication patterns do classical protocols provide?

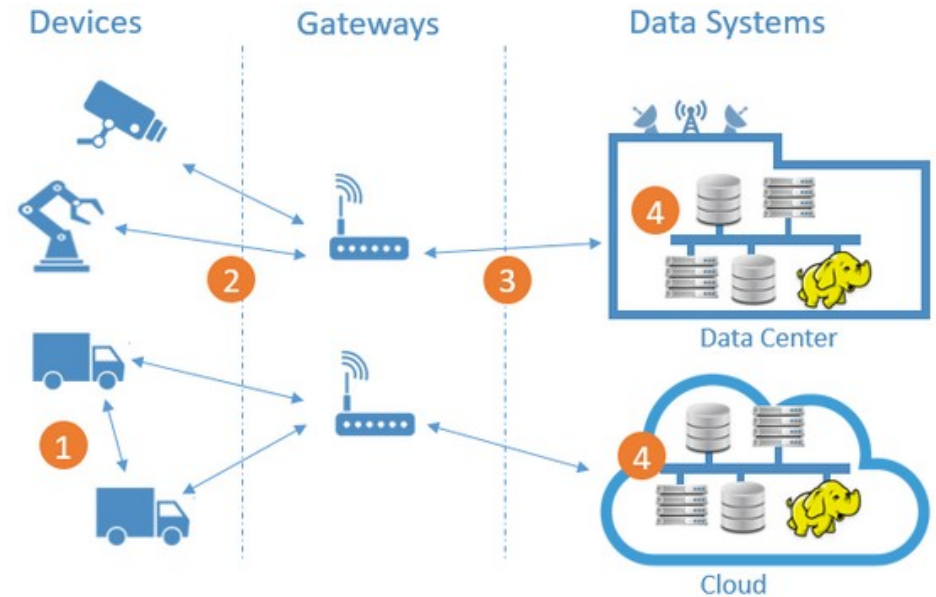
- IP:
  - allows only direct communication (sender has to know precisely its receiver)
  - no QoS
  - synchronous (receiver must be able to receive messages)
  - generally one sender and one receiver (even if multicast allows some form of function-based communication with only one sender)
  - etc.
- HTTP or Web-based application:
  - allows only direct communication
  - synchronous (server must be alive)
  - heavy (on top of TCP), the sender needs to implement HTTP, TCP etc.
  - verbose exchange (e.g. see HTTP header)
  - etc.

**Web page response:**  
HTTP/1.1 200 OK  
Date: Fri, 03 Nov 2023 20:34:18 GMT  
Server: Apache/2.2.9 (Debian) PHP/5.2.6-1+lenny16  
X-Powered-By: PHP/5.2.6-1+lenny16  
Content-language: fr  
Vary: Accept-Encoding, Cookie  
Expires: Thu, 01 Jan 1970 00:00:00 GMT  
Cache-Control: private, must-revalidate, max-age=0  
Last-modified: Mon, 30 Oct 2023 11:43:44 GMT  
Content-Encoding: gzip  
Content-Length: 5451  
Content-Type: text/html; charset=UTF-8  
Keep-Alive: timeout=5, max=98  
Connection: Keep-Alive

```
<!DOCTYPE html>  
<html>  
...
```

# Homework

- Read **Impact des objets sur les protocoles de l'Internet** (generalities, RPL, CoAP) – not available anymore
- Read **Understanding IoT Protocols**



# MQTT

– pub-sub communication model –

# MQTT

- Message Queuing **Telemetry** Transport
- First version in 1999, currently at version 5.0 (2019)
- ISO and OASIS standard
- Available at <http://docs.oasis-open.org/mqtt/mqtt/v5.0/> (137 pages)
- Uses exclusively the **publish-subscribe** pattern, targeting device data collection

# MQTT use cases / purpose/aims / applications / motivations / assumptions

- Designed for resource-constrained devices and low bandwidth, high latency networks such as occasional dial-up lines with healthcare providers and satellite links
- Targets device data collection or management, MQueueingTelemetryT (telemetry = remote monitoring; no queues, despite its name)
- Use cases: in embedded systems and a range of home automation and small device scenarios, where a “small code footprint” is required: monitoring or management of a simple and huge network of small devices via Internet, i.e. underwater lines, energy grid, power usage monitoring, lighting control, intelligent gardening, monitoring a huge oil pipeline for leaks or vandalism, a parking where sensors identify number and location of vacant parking spots, temperature updates, stock price tickers, oil pressure feeds with mobile notifications, connecting machines together, such as an Arduino device to a web service
- These systems need to collect data from many sources (large networks of small devices, thousands of sensors) and concentrate it into a single location for analysis; when the system finds a problem, it can take action to correct that problem
- Pros: low footprint, lightweight for constrained networks + flexibility to choose QoS + simplicity (just five API methods), a compact binary packet payload (no message properties, compressed headers, much less verbose than something text-based like HTTP)
- Cons: high power consumption due to the TCP-based connection

# Standards organisations

- IETF – RFC, Internet protocols: TCP, IP, AV1, SIP, HTTP, ...
- ISO (International Organization for Standardization), ITU (International Telecommunication Union), IEC (International Electrotechnical Commission) – JPG, HEVC/H.265, ISO 9001 (*système de management de la qualité*), OpenDocument, ISO/IEC Office Open XML (Microsoft's .docx), ITU H.323, ISO/IEC HTML
- OASIS (Organization for the Advancement of Structured Information Standards) – MQTT, OpenDocument
- IEEE (Institute of Electrical and Electronics Engineers) – 802.3 (Ethernet), 802.11 (Wi-Fi)
- many others

# MQTT topics

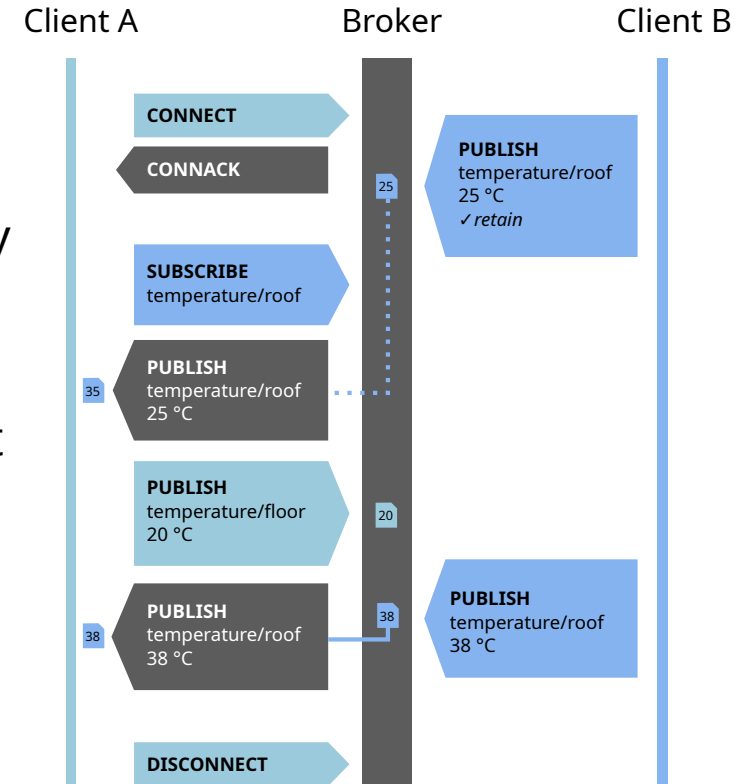
- Recall that in pub-sub model, publishers categorise their messages into topics
- Topics are UTF-8 strings, with one or more topics separated by “/”, thus creating a [hierarchy](#)
  - topic alias: in v5.0, they can be a number (allowing to reduce packet size)
- + wildcard in subscription matches any string for a single topic at that position
  - “home/+/humidity” matches “home/kitchen/humidity” and “home/bedroom/humidity”, but does not match “home/kitchen”
  - +/important matches all important data
- # wildcard as last character in subscription matches any string for zero or more topic levels
  - “home/#” matches the topics “home”, “home/”, “home/kitchen” and “home/bedroom/temperature”
- Brokers automatically create the prefix #P2P/ for each client, which enables messages to be sent directly to that client (for example, in request/reply scenarios) ??

# MQTT layer

- At application layer, on top of TCP/IP, port 1883 (8883 if over TLS)
  - MQTT-SN (for sensor networks) is a variation aimed on embedded systems using Zigbee or Bluetooth

# MQTT message types

- Connect – waits for a connection to be established with the server and creates a link between the nodes
- Disconnect – waits for the MQTT client to finish any work it must do, and for the TCP/IP session to disconnect
- Publish – returns immediately to the application thread after passing the request to the MQTT client
  - if the message has the *retained* flag to true, then the broker retains the message so that any subsequent client subscribing to that topic receive that value (the most current value) rather than waiting for the next update from the publisher
- others



Example of an MQTT connection (QoS 0) with connect, publish/subscribe, and disconnect. The first message from client B is stored due to the retain flag [wikipedia]

# MQTT packet types

- Reserved
- CONNECT, CONNACK
- PUBLISH, PUBACK, PUBREC, PUBREL, PUBCOMP
- SUBSCRIBE, SUBACK, UNSUBSCRIBE, UNSUBACK
- PINGREQ, PINGRESP
- DISCONNECT
- AUTH

# MQTT control packet format

- All MQTT packets are control packets, which have:
  - fixed header, mandatory
    - 1 byte: packet type (see previous slide) + specific flags
    - 1 to 4 bytes: remaining length of this packet
  - variable header, mandatory
    - its length depends on the packet type
    - examples: Packet identifier, Properties, Reason code (success, protocol error, not authorized, unsupported protocol version etc.)
  - payload, optional
    - application specific
- A message can have from 2 bytes to 256 MB

# MQTT security

- MQTT sends connection credentials in plain text format and does not include any security measures (integrity, authentication etc.)
- Security can be provided by the underlying TCP transport (TLS for ex.)

# MQTT – quality of service (QoS)

- Each connection to the broker can specify a QoS:
  - level 0, at most once - the message is sent only once and the receiver takes no additional steps to acknowledge delivery (fire and forget)
  - level 1, at least once - the message is re-tried by the sender multiple times until acknowledgement is received (acknowledged delivery)
  - level 2, exactly once - the sender and receiver engage into a communication that ensures that exactly one copy of the message is received (assured delivery??)
- This QoS is done at application level, between MQTT senders and receivers, whereas TCP is at transport layer between clients to broker only, e.g. TCP ensures that the sender's packet arrive from sender to broker, but does not ensure that it arrives to all the subscribers (the broker might die in the process) [[link](#)]

# MQTT – applications

- Facebook has used aspects of MQTT, but “it is unclear how much and for what”
- Amazon Web Services announced Amazon IoT based on MQTT in 2015
- The OpenStack Upstream Infrastructure's services are connected by an MQTT unified message bus with Mosquitto as the MQTT broker
- Microsoft Azure IoT Hub uses MQTT as its main protocol for telemetry messages
- XIM, Inc. launched an MQTT client called MQTT Buddy in 2017, for Android and iOS
- Open-source software home automation platform Home Assistant is MQTT enabled and offers four options for MQTT brokers

# MQTT Debian packages

- mosquitto - MQTT version 5.0/3.1.1/3.1 compatible message broker
- mosquitto-clients - Mosquitto command line MQTT clients
- libmosquitto1 - MQTT version 5.0/3.1.1/3.1 client library
- libmosquittopp1 - MQTT version 5.0/3.1.1/3.1 client C++ library
- libmqtt-client-java - Java MQTT Client API
- node-mqtt-packet - parse and generate MQTT packets [for node.js]
- python3-paho-mqtt - MQTT client class (Python 3)

# MQTT – questions

- Read <https://en.wikipedia.org/wiki/MQTT>
- What OSI layer does MQTT work at? How many QoS types does MQTT provide?
- What is shared subscription, a feature of MQTT v5.0?
- How many message types does MQTT have? What is the purpose of SUBACK message type and what does it mean if the first byte (bit?) of its payload is 1?

# STOMP

– text-oriented packets –

# STOMP

- Simple (or Streaming) Text Oriented Messaging Protocol
- First version appeared in 20xx, version 1.2 (2012), available at <http://stomp.github.io/stomp-specification-1.2.html> (very short and easy to understand)
- Text-based, similar to HTTP
- Clients and servers: <https://stomp.github.io/implementations.html>
- Design goals: simplicity and interoperability
- Very easy to write a client, “many developers have told us that they have managed to write a STOMP client in a couple of hours in their particular language”
  - you can use telnet as STOMP client

# STOMP commands (frames)

- Client->server:
  - CONNECT, DISCONNECT
  - SEND
  - SUBSCRIBE, UNSUBSCRIBE
  - BEGIN, COMMIT, ABORT
  - ACK, NACK
- Server->client:
  - CONNECTED, MESSAGE, RECEIPT, ERROR
- The communication unit is called a “frame”
- Read <https://stomp.github.io> and the complete 1.0 specification (very easy to understand, <https://stomp.github.io/stomp-specification-1.0.html>)
  - explain [http://stomp.github.io/stomp-specification-1.2.html#Augmented\\_BNF](http://stomp.github.io/stomp-specification-1.2.html#Augmented_BNF) for frame BNF

CONNECT

^@ <-- null (ctrl-@)

SEND

destination:/queue/a

hello queue a

^@

SUBSCRIBE

destination: /queue/foo

ack: client

^@

CONNECTED

session: <session-id>

^@

MESSAGE

destination:/queue/a

message-id: <message-identifier>

hello queue a^@

# Transactions

- Transaction = several exchanges performed in atomic way: either all of them, or none of them
  - example: money transfer

# STOMP peculiarities

- STOMP does not deal in queues and topics—it uses a SEND semantic with a “destination” string. The broker must map onto something that it understands internally such as a topic, queue, or exchange. Consumers then SUBSCRIBE to those destinations. Since those destinations are not mandated in the specification, different brokers may support different flavours of destination. So, it’s not always straightforward to port code between brokers.
- One of the most interesting examples is with RabbitMQ Web Stomp (<https://www.rabbitmq.com/docs/web-stomp>) which allows you to expose messaging in a browser through websockets. This opens up some interesting possibilities—like updating a browser (i.e. Web page), mobile app, or machine in real-time with all types of information.

# STOMP Debian packages

- stompserver - stomp messaging **server** implemented in Ruby
- libnet-stomp-perl - **Perl** module providing a Streaming Text Orientated Messaging Protocol client
- php-stomp - Streaming Text Oriented Messaging Protocol (STOMP) client module for **PHP**
- python3-stomp - STOMP client **library** for **Python 3**
- python3-stomper - **Python** client implementation of the STOMP protocol (Python 3)
- ruby-stomp - **Ruby** client for the stomp messaging protocol
- syslog-ng-mod-stomp - Enhanced system **logging** daemon (STOMP plugin) – publish log messages through the STOMP protocol

# AMQP

– rich queue-based communication model –

# AMQP

- Advanced Message Queuing Protocol
- OASIS standard 2012, [\[standard\]](#) (125 pages)
- ISO/IEC standard in 2014
- Message orientation or queuing (it is all about queues)
- Routing (including point-to-point and pub-sub, e.g. when the message is sent to all subscribers)
- Interoperability
- Application layer, on top of TCP
- Binary
- Reliability or message delivery guarantee (not losing messages, ensuring each message is delivered as intended, regardless of failures or reboots, origins in banking industry): at most once, at least once, exactly once
- Authentication and/or encryption based on SASL and/or TLS
- Relatively high resource utilization, i.e. power and memory usage
- Enterprise-style protocol, rich in features

# Use cases

- Mostly used in business messaging, see <https://www.amqp.org/about/examples>
- Used for reliability in large companies to move data around their organisation
- Provides a wide range of features related to messaging, including pub-sub messaging. AMQP exchanges route messages directly—in fanout form, by topic, and also based on headers
  - a lot of fine-grained control: can restrict access to queues, manage their depth, and more. Features like message properties, annotations and headers make it a good fit for a wide range of enterprise applications
- JP Morgan uses it to process 1 billion messages a day
- Used in one of the world's largest biometric databases India's Aadhar project—home to 1.2 billion identities
- Used in the Ocean Observatories Initiative—an architecture that collects 8 TB of data per day

# AMQP type system

- Defines a rich self-describing encoding scheme allowing interoperable representation of a wide range of commonly used types
- Used on packets exchanged between peers
- Primitive types, both common scalar values and common collections:
  - scalar types: boolean, integral numbers (ubyte, ushort, uint, ulong, byte, short, int, long), floating point numbers (float, double), timestamp, UUIDs, characters, strings, binary data, and symbols
  - collection types: array (monomorphic), list (polymorphic), and map
- Described types, allows to annotate type with semantic information, e.g. URL, customer
- Composite types, composed of a sequence of fields, each with name, type, and multiplicity, and defined with one or more descriptors, all of them described using XML
- Restricted types, which restrict values of an existing type (aka enumeration in programming languages)

```
<type name="array" class="primitive">  
  <encoding name="array8" code="0xe0" category="array" width="1"  
    label="up to 2^8 - 1 array elements with total size less than 2^8 octets"/>  
  <encoding name="array32" code="0xf0" category="array" width="4"  
    label="up to 2^32 - 1 array elements with total size less than 2^32 octets"/>  
</type>
```

# AMQP protocol and frames, message exchange

- The link protocol is at the heart of AMQP
- Connections between two peers are initiated with an **open** frame in which the sending peer's capabilities are expressed, and terminated with a **close** frame.
- A connection can have multiple sessions multiplexed over it, each logically independent. A session is a bidirectional, sequential conversation between two peers that is initiated with a **begin** frame and terminated with an **end** frame. Multiple links, in both directions, can be grouped together in a session
- An **attach** frame body is sent to initiate a new link; a **detach** to tear down a link. Links may be established in order to receive or send messages
- Messages are sent over an established link using the **transfer** frame. Messages on a link flow in only one direction
- **Transfers** are subject to a credit based flow control scheme, managed using **flow** frames. This allows a process to protect itself from being overwhelmed by too large a volume of messages or more simply to allow a subscribing link to pull messages as and when desired.
- Each transferred message must eventually be settled. Settlement ensures that the sender and receiver agree on the state of the transfer, providing reliability guarantees. Changes in state and settlement for a transfer (or set of transfers) are communicated between the peers using the **disposition** frame. Various reliability guarantees can be enforced this way: at-most-once, at-least-once and exactly-once.

# AMQP message format

- The bare message = what is created by sender application, is immutable during transit
  - this allows for end-to-end message signing and/or encryption and ensures that any integrity checks (e.g. hashes or digests) remain valid
- Intermediaries can add annotations to the message, which are added before or after the bare message, yielding an annotated message
  - the header is a standard set of delivery-related annotations that can be requested or indicated for a message and includes time to live, durability, priority
- The bare message itself is structured as an optional list of standard properties (message id, user id, creation time, reply to, subject, correlation id, group id etc.), an optional list of application-specific properties (i.e., extended properties) and application data (the body)

# AMQP transaction feature

- Similar to STOMP's transaction feature

# AMQP security

- Not mandatory
- TLS – data encryption
- SASL – peer authentication

# Debian packages – AMQP clients

- amqp-tools - Command-line utilities for interacting with AMQP servers (from RabbitMQ)
- golang-github-streadway-amqp-dev - Go client for AMQP 0.9.1
- libmessage-passing-amqp-perl - input and output message-pass messages via AMQP
- php-amqp - AMQP extension for PHP
- php-amqp-lib - pure PHP implementation of the AMQP protocol
- python3-aioamqp - AMQP implementation using asyncio (Python3 version)
- python3-amqp - Low-level AMQP client (Python3 version)
- python3-amqp-lib - simple non-threaded Python AMQP client library (Python3 version)
- python3-kombu - AMQP Messaging Framework for Python (Python3 version)
- python3-pika - AMQP client library for Python 3
- ruby-amqp - feature-rich, asynchronous AMQP client
- syslog-ng-mod-amqp - Enhanced system logging daemon (AMQP plugin)
- librabbitmq4 - AMQP client library written in C
- libanyevent-rabbitmq-perl - asynchronous and multi channel Perl AMQP client (from RabbitMQ)

# AMQP applications

- debci - continuous integration system for Debian
  - scans the Debian archive for packages that contain DEP-8 compliant test suites, and run those test suites whenever a new version of the package [...] is available. The requests are distributed to worker machines through AMQP queues.
- syslog-ng-mod-amqp - Enhanced system logging daemon (AMQP plugin)

CoAP  
— ??? —

# CoAP

- Constrained Application Protocol
- IETF standard, RFC 7252 (2014)
- Available at <https://tools.ietf.org/html/rfc7252> (112 text pages)
- Designed for constrained devices: multicast support, very low overhead, and simplicity
- It enables those constrained devices called “nodes” to communicate with the wider Internet using similar protocols:
  - between devices on the same constrained network (e.g., low-power, lossy networks)
  - between devices and general nodes on the Internet
  - and between devices on different constrained networks both joined by an internet
- CoAP is also being used via other mechanisms, such as SMS on mobile communication networks

# CoAP motivation and assumption: LLN, low power and lossy networks

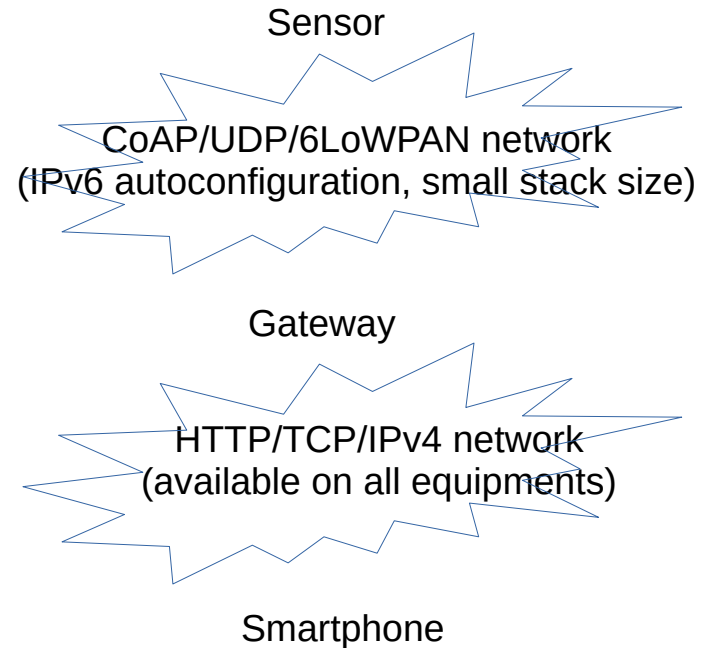
- IoT uses a direct communication between devices
  - sensors to some application software which adjusts some process
  - sensors to personal appliances (*appareils personnels*)
  - need for data exchange standards
- Devices (e.g. wireless sensors) are often tiny, embedded, in harsh environment, ...
- => **Constrained devices:** energy, memory, processing capability etc. (“nodes often have 8-bit  $\mu$ controllers”)
- => **Constrained links:** high loss rates, low data rates, instability etc. (“6LoWPAN often have high packet error rates and a typical throughput of tens of kb/s”)

# CoAP features

- Uses the request/response model
- Simple subscription for a resource, and resulting push notifications (sensor->sink)
- Supports service and resource (through CoRE link format) discovery
- Simple caching based on max-age
- Asynchronous communication (place message in queue, answer can be sent later)
- “Since MQTT’s arrival as a standard, with its equal handling of constrained devices, and much broader feature set beyond that, few people are choosing CoAP for new efforts” [[solace](#)]
- Uses a subset of HTTP for easy integration with the Web
  - includes concepts of the Web, such as URI and Internet media types
  - defines the mapping with HTTP, allowing proxies to be built providing access to CoAP resources via HTTP in a uniform way
  - Web of things, where real objects become part of WWW
  - thus, a complete networking stack of open-standard protocols that are suitable for constrained devices and environments becomes available

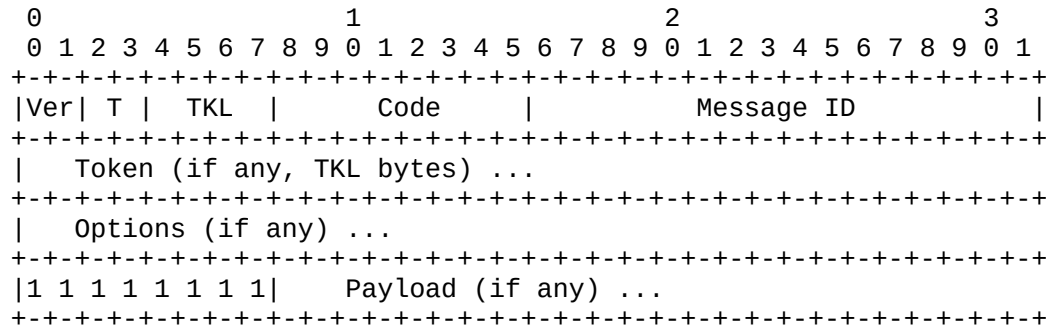
# Use example

- A water meter is installed for at least 20 years, and a sensor inside a wall cannot easily be changed
- Internet protocols are chatty (“bavards”): they send regularly packets, and are non deterministic
- We need to avoid dynamic routes, collision detection, continuous reception etc.
- A smartphone uses HTTP to request sensor's temperature, the gateway changes it to CoAP and sends it to the sensor; it stores its answer and its validity duration given by sensor (it acts as a cache), so that other requests do not contact the sensor again



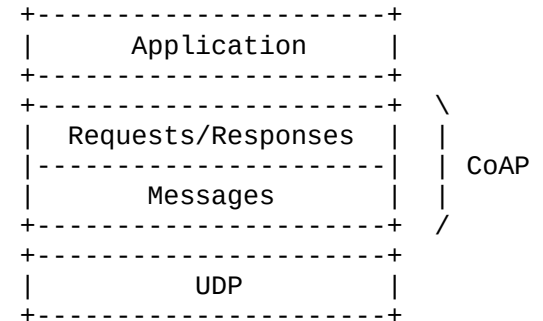
# CoAP message structure

- Use simple, binary, base header format
- Fixed 4-byte header: version (01), type (one of the four, see next slide), token length, code (request/response method)
- Token: between 0 and 8 bytes
- Options are in an optimized Type-Length-Value format
- The length of the message body (payload, as simple HTTP message) is implied by the datagram length; when bound to UDP, the entire message must fit within a single datagram



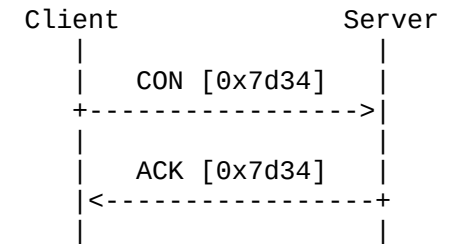
# CoAP message formats

- Four types of messages: Confirmable, Non-confirmable, Acknowledgement, Reset
- Method Codes and Response Codes included in some of these messages make them carry requests or responses
- Two types of data: requests and responses
  - requests can be carried in Confirmable and Non-confirmable messages, and responses can be carried in these as well as piggybacked in Acknowledgement messages
- Bound to UDP by default, port 5683, with optional reliability
- Optionally uses DTLS (Datagram TLS, with similar features as TLS), port 5684, providing a high level of communications security



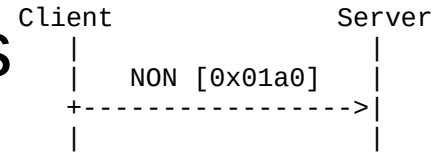
# CoAP confirmable messages – reliability

- Message ID is used to detect duplicates and for optional reliability
- Reliability is provided by marking a message as Confirmable (CON)
- A Confirmable message is retransmitted using a default timeout and exponential back-off between retransmissions (basic congestion control), until the recipient sends an Acknowledgement message (ACK) with the same Message ID (e.g. 0x7d34) from the corresponding endpoint (or runs out of attempts)



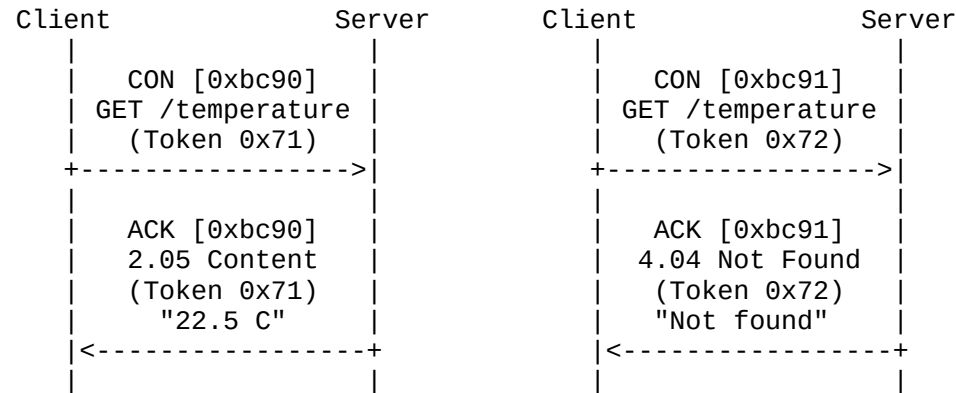
# CoAP non-confirmable messages

- A message that does not require reliable transmission (for example, each single measurement out of a stream of sensor data) can be sent as a Non-confirmable message (NON)
- These messages are not acknowledged, but still have a Message ID for duplicate detection



# CoAP request-response exchange

- Uses GET, POST, PUT, DELETE, like HTTP
- Nodes may cache responses (up to max-age value provided by server), and may be proxies



# CoAP Debian packages

- libcoap3-bin - C-Implementation of CoAP - example binaries API version 3 (client and server)
- libcoap3 - C-Implementation of CoAP - libraries API version 2

# CoAP – questions

- What does it mean **service** discovery in CoAP?  
How does it work?
- Why do packets use a byte of value 255 before the payload field?

# Other messaging protocols

# Other messaging protocols

## XMPP

- much more complex than MQTT
- based on XML (Extensible Markup Language)
- instant messaging: enables the near-real-time exchange of structured yet extensible data between any two or more network entities
- client-server architecture, distributed system (like e-mail: no central point)
- used by Facebook WhatsApp, Chat etc.

- DDS, Data Distribution Service – focus on real-time communication
  - aims to enable scalable, real-time, dependable, high-performance and interoperable data exchanges using a publish-subscribe pattern
  - appropriate to applications like autonomous vehicles, robotics, transportation systems, power generation, medical devices, aerospace and defense, and other real-time applications
  - patents involved
- OPC UA, OPC Unified Architecture – focus on communicating with industrial equipment and systems for data collection and control
- WAMP, Web Application Messaging Protocol – based on microservices
- MSMQ, Microsoft Message Queuing – based on queues, closed-source

# Message-oriented middleware, implementation

# API, libraries

- Until now, we have seen protocols (MQTT, AMQP, XMPP), which, like other protocols such as HTTP, POP3, SMTP, and SNMP, specify message format, or data to send
  - on the contrary, they do not specify an API, e.g. nobody forces you to use function X to create HTTP header and function Y to append the HTML page
- Examples of MOM (messaging libraries):
  - JMS (Java Message Service), Java
  - ZeroMQ, cross-language
    - supports pub-sub, push-pull, request-reply, router-dealer, ... patterns

# JMS – history

- 1995 – Sun creates Java language
- 2009–2010 – Oracle acquires Sun and maintains Java under Java Community Process (JCP)
- 2017 – Oracle transfers Java EE to Eclipse foundation, which decides to base it on Java 8, the current version at that time
- 02/2018 – Oracle does not want Java word in Java EE (Java means Oracle-made); community chooses Jakarta EE as the new name for Java EE
- 09/2019 – Jakarta EE 8 released, compatible with Java EE 8
  - (currently, Java EE is at v8, and Java SE is at v13)
- => two versions available:
  - old: v6 from Oracle, with JMS v1.1 (maintenance release exists, v2.0a from 2015, from JCP, [specification](#))
  - new: v8 from Eclipse, with JMS v2.0a, the same as above
- => we will use the “stable” version from Oracle, v1.1, [specification](#) (2002, 125 pages) and [tutorial](#)

# JMS features

- API specification allowing to create, send, receive, and read messages
- Part of Jakarta EE (ex-Java EE), which provides:
  - publisher-subscriber and point-to-point models
  - message topics
  - message consumption: synchronous (receive) or asynchronous (listener)
  - separation between application and transport layer ??
- Reduces the set of concepts needed by programmer to learn
- Low level, no protocol implemented => JMS systems are not interoperable

# JMS current (hopefully temporary) installation mess

- Install a functional Java SE
- Download JMS: <http://www.java2s.com/Code/Jar/j/Downloadjavaxjms111jar.htm> (or <http://www.java2s.com/Code/Jar/j/Downloadjavaxjms110jar.htm> ?)
- Needs GlassFish (Java EE official implementation), plus either NetBeans or Ant/applclient
- Too much movement
  - JMS is based on old Java EE (2002), hence risk of problems
  - currently, transition to Eclipse foundation, not yet finished
  - GlassFish is not packaged in Debian
  - GlassFish is almost stopped (last version 1 year ago, v5.0.0, end of commercial support); Payara (zip of 143 MB) replaces it, I fear potential version problem
  - tens of lines for the basic example
- => inappropriate to our lab

# IoT security

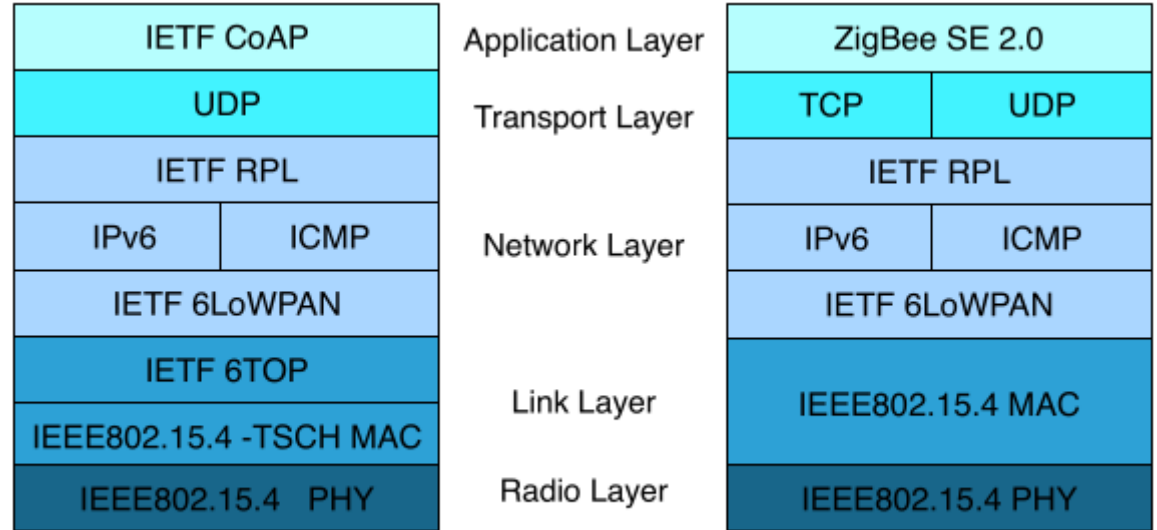
# IoT security

- Many brokers and sensors are available on Internet
- Read [Criminals Hacked A Fish Tank To Steal Data From A Casino](#)
- Write the three most important ideas (in your opinion) from <https://blog.avast.com/mqtt-vulnerabilities-hacking-smart-homes>
- Read Description of [RocketMQ vulnerability](#)
- Be very careful if your broker or sensors have access to Internet, especially if sensor data is confidential!

# RPL, IPv6 Routing Protocol for Low-Power and Lossy Networks – IoT routing protocol –

# RPL

- The previous protocols are at application layer, i.e. used to exchange [data](#) (e.g. humidity values, like HTTP)
- They are on top of a protocol at network layer, which route messages (like IPv4 or IPv6)
- However, sometimes nodes are simply deployed, and do not know each other, there is no routing table, hence a routing protocol is needed (like RIP or OSPF)
- RPL is a [routing protocol](#)
- Pronunciation: ripple (“vaguelette, ondulation”)
- Multi-hop, can support thousands of nodes
- Adopted also by ZigBee alliance
- RFC 6550 (standardised in 2012, 154 text pages), with special applications (home and industrial automation, urban networks) specified in [other RFCs](#)



(a) IETF standardized stack.

(b) ZigbeeIP stack.

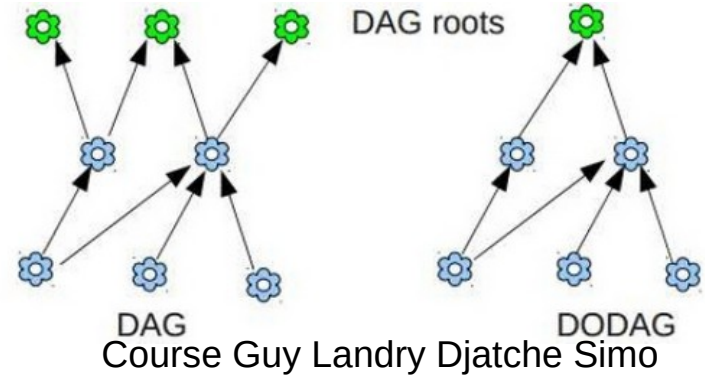
O. Iova et al. RPL, the...

# RPL features

- Optimised for point-to-multipoint (one-to-many, from a central control point to a subset of devices inside the LLN) and multipoint-to-point (many-to-one, from devices inside the LLN towards a central control point), but supports also one-to-one (point-to-point, between devices inside the LLN) communications
- This is the case for sensor data collection networks
- Is proactive and based on distance vectors (best route is based on distance, routers exchange their RT), contrary to link-state (where routers exchange connectivity information)
- Nodes can be hosts (sensors) and routers at the same time
- RPL quickly creates network routes, shares routing knowledge and adapts the topology (when network conditions change) quickly and efficiently

# RPL topology

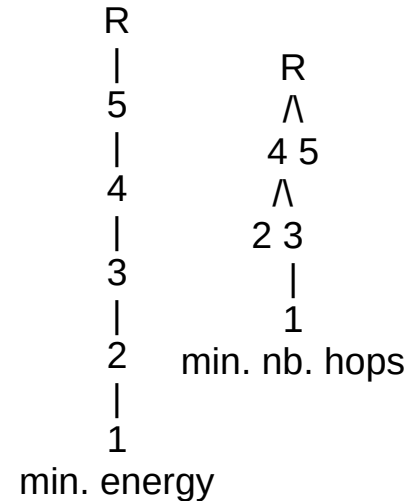
- LLNs are typically not deployed with a predefined topology, so the network needs to be discovered and a topology used in communications to be created
- RPL creates a topology similar to a tree (DAG, directed acyclic graph)
  - all edges are oriented
  - no cycles exist
  - can have several roots
- Rank = distance to a root node (0 for the root), such as number of hops
- DODAG (destination-oriented DAG) = DAG with a single root node
- A DODAG root may act as a border router, i.e. connect the DODAG to external world through a common backbone using other protocols, for ex. a urban network not fully connected so with several DODAG



# RPL instances and objective function

- A DODAG has an optimisation objective, computed using an objective function and defined by the application, e.g.:
  - distance in terms of number of hops to root
  - transmission energy minimisation
  - latency minimisation
  - etc., or application-specific constraints fulfilling
- Nodes must support at least one of these metrics: hop count, latency, or ETX (expected transmission count for successfully reception)
- RPL instance = one or several DODAGs with the same optimisation objective
- A network can run multiple, independent instances of RPL concurrently, with different optimisation objectives

1 --- 2 --- 3 --- 4 --- 5 --- R  
 (communication range is 2)

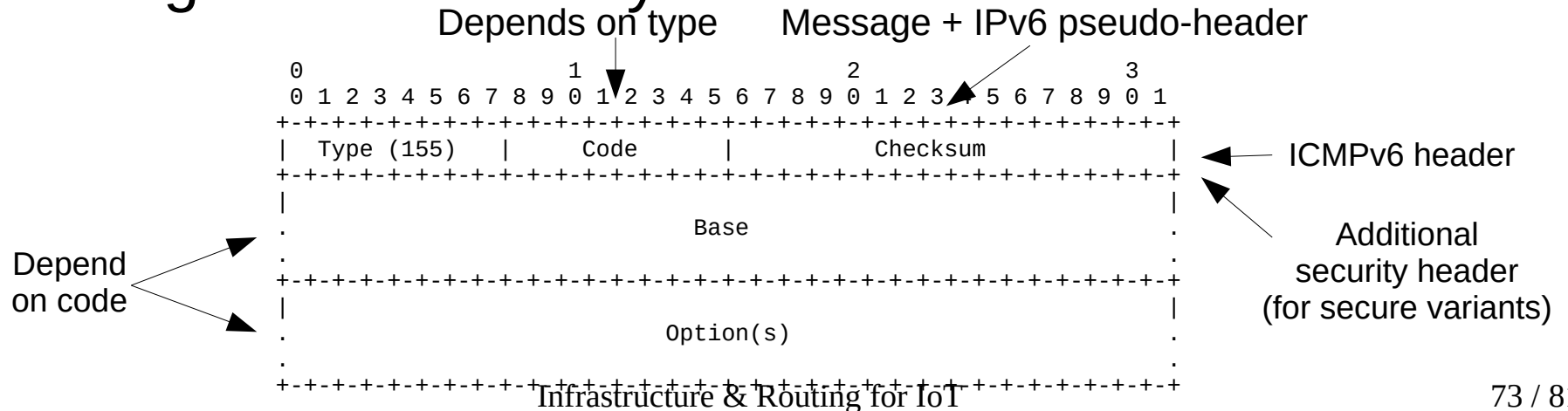


# Forwarding and routing

- Traffic moves either upwards (to a root), or downwards (from a root to a node) inside a DODAG
- Upward path: mp2p, very common (collection point)
- Downward path: p2p and p2mp
- Nodes inform parents of their presence, and their descendants of its reachability
- When going up, if no node with lower rank exists, go to sibling (brother)
- When going down, increase rank

# RPL control message

- An ICMPv6 message (ICMP for IPv6, on top of IPv6 header)
  - other protocols using ICMPv6 messages: PMTU, neighbour discovery



# RPL message codes

- The topology is maintained using DAO (Destination Advertisement Object) messages, sent by nodes to their parent nodes throughout the DAG
  - the waiting time before reforwarding DAO messages is governed by Trickle algorithm
  - Trickle algorithm ([RFC 6206](#), 2011) goal is to ensure consistency (its neighbours have not changed) with very low overhead: when a node's data does not agree with its neighbors, that node communicates quickly to resolve the inconsistency (e.g., in milliseconds); when nodes agree, they slow their communication rate exponentially, such that nodes send packets very infrequently (e.g. a few packets per hour). It is simple to implement; and requires very little state, e.g. 4–11 bytes of RAM and 50–200 lines of C code
- DAO-ACK: answer to a DAO, unicast
- DIS (DODAG Information Solicitation): sent to a RPL node to request information from nearby DODAG, analogous to router request messages used to discover existing networks
- DIO (DODAG Information Object): usually sent in response to DIS messages
  - contains node's routing metric
  - allows to select a DODAG parent set, and maintain the DODAG

# Upward routes – discovery and maintenance

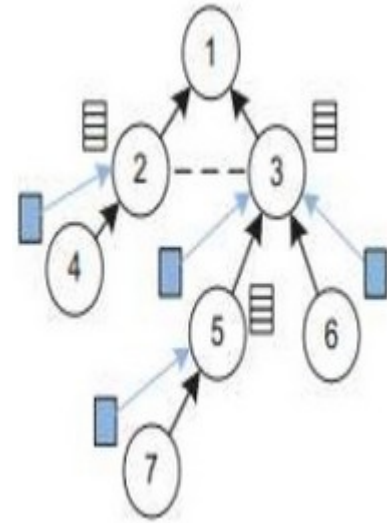
- Typical LLNs exhibit variations in physical connectivity that are transient and harmless to traffic, so a routing protocol which updates quickly the topology is needed
- Initially, RPL consists of one or several roots
- The root(s) send periodically DIO messages
  - they provide information about the DODAG, such as DODAG id, objective function used
  - use Trickle algorithm to compute the spacing between them
- Upon reception, the node computes (integrates the DODAG) or updates its rank and its parent by choosing the smallest possible rank (among all answers)
- A new node can also join a DODAG by sending a DIS message to request a DIO message

# Storing and non-storing modes

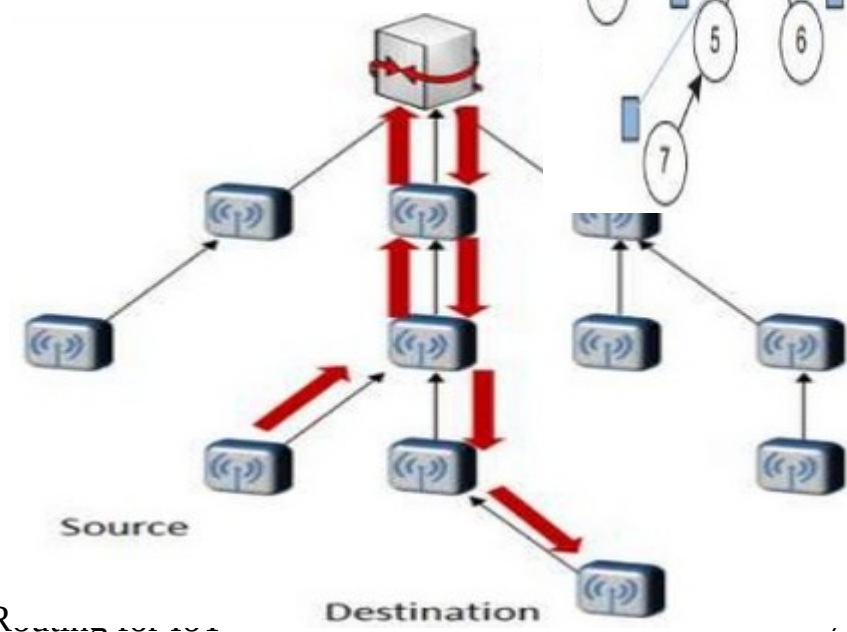
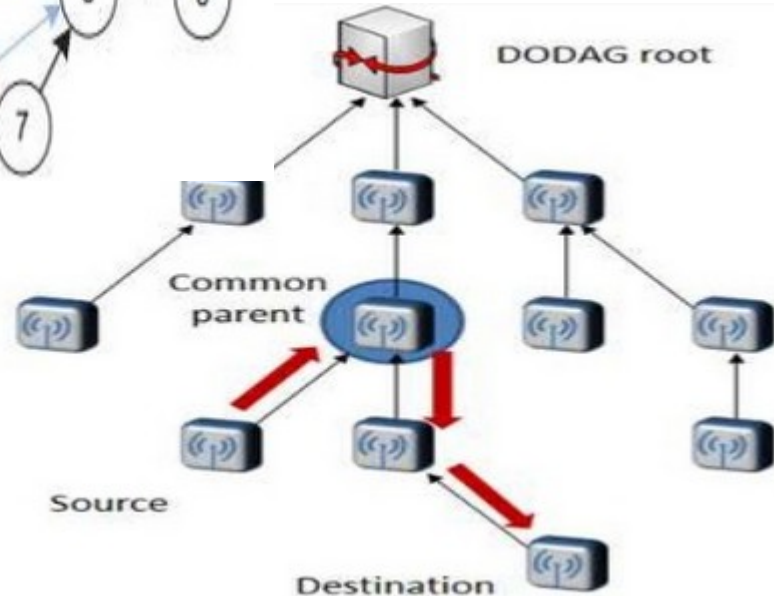
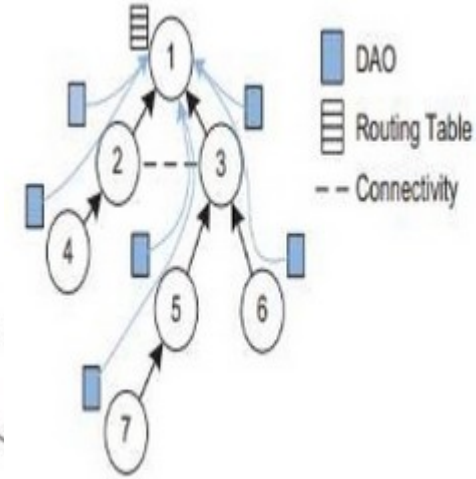
Course Guy Landry Djatche Simo

A DODAG can work only in one mode at a time

RT on parents  
DAO sent to parents



Unique RT on root  
Updates sent only to it  
DAO sent to root



# Security

RPL has three modes to support message confidentiality and integrity:

- unsecured: RPL has no security feature, but link-layer could provide them
- preinstalled: nodes joining a RPL instance need preinstalled keys that enable them to process and generate secured RPL messages
- authenticated: like preinstalled, but this allows to join only as leafs; to be router, it is required to contact an authentication authority to obtain a second key

# Implementation in OSes

- Contiki, small OS for small systems
- LiteOS
- TinyOS, uses events and guided tasks, uses nesC (extension of C)
- RIOT, focuses on low-power wireless IoT devices
- etc.
- Why is not there any RPL package in Debian?

# Homework

- Slides 1–27 of [The IoT/IP protocols](#)
- Section 2.3.2 from [Kamgueu's PhD thesis](#) (in French)
- Section II of [RPL: the Routing Standard for the Internet of Things... Or Is It?](#)
- Sections II and III of [RPL Routing Protocol over IoT: ...](#)

# Conclusions

# Conclusions

- IoT uses specific protocols, at application level generally
- Several protocols exist, each better for its use case
- The broker links all clients together, no matter the protocol
- Messaging paradigms: pub-sub, message-oriented, request-response etc.
- Some protocols target low resources, others simplicity or richness of features
- RPL creates routes in dynamic networks