Infrastructure and routing for connected objects

"*The more one pays attention to the Internet of Things (IoT), the more one learns to appreciate simple, unconnected devices. Your editor long ago acquired an aversion to products that advertise themselves as "smart" or "WiFi-enabled". There can be advantages, though [...] if they are implemented well.*" [J. Corbet, https://lwn.net/Articles/940509]

Eugen Dedu

Maître de conférences (associate professor)
Université de Franche-Comté, IUT Nord Franche-Comté
Master 1 IoT
Montbéliard, France
September 2024

http://eugen.dedu.free.fr
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

Communication protocols in IoT:
- MQTT, STOMP, AMQP, CoAP
- low-power & lossy wireless netw. prot.: RPL

Labs on a GNU/Linux machine

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

Organisation of the module

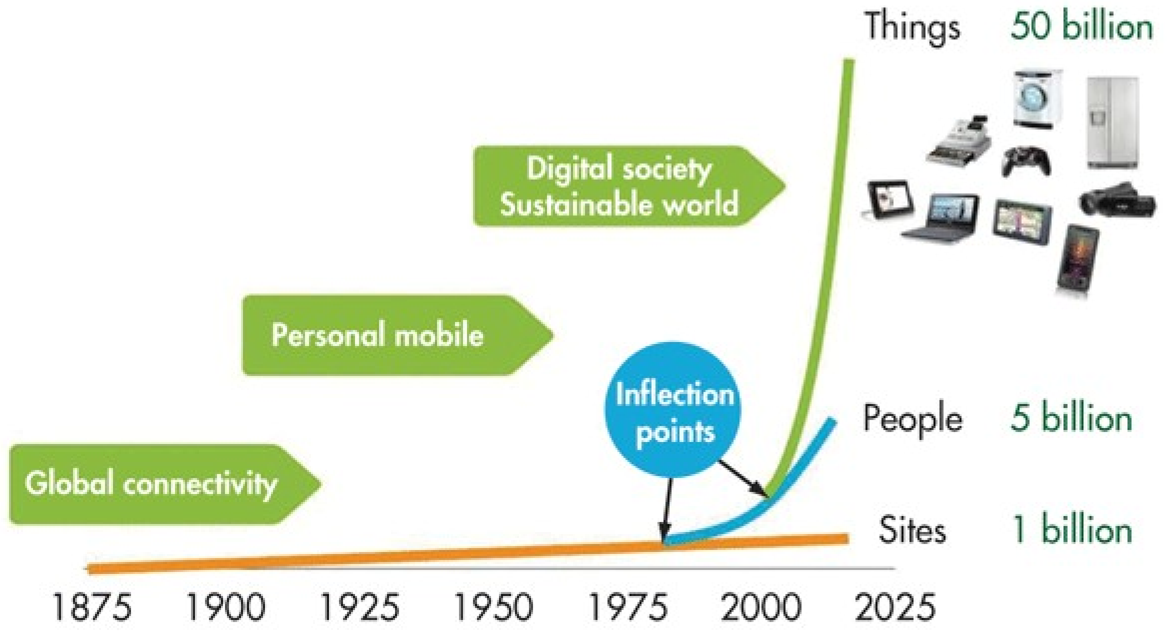
- Does not treat low-power low-level wireless IoT technologies, which will be presented next semester in *Radio networks* (together with their protocol, if any):
  - Sigfox, LoRa (physical layer) and LoRaWAN (communication protocol and system architecture)

- 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

Requirements of IoT protocols

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
- 10 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.
- The IoT and the intelligent systems it enables will fundamentally change our world.



The much-hyped IoT revolution simply hasn't happened to the extent many had anticipated. In 2015, Gartner projected 25 billion connected devices by 2020. In 2019, the firm lowered its 2020 outlook to 5.81 billion devices.

M2M and its wider IoT

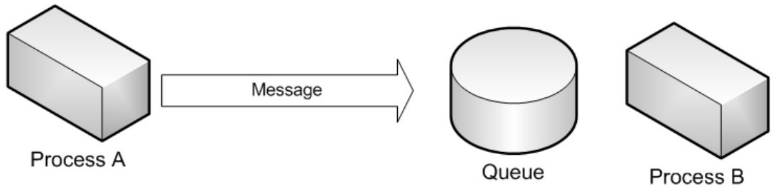
- Want it or not, more and more applications that we use today are connected
- M2M (machine to machine) and IoT both refer to devices communicating with each other
- M2M refers to isolated instances of device-to-device communication
- IoT refers to a grander scale, synergizing vertical software stacks to automate and manage communications between multiple and heterogeneous devices
- M2M uses protocols embedded within hardware, IoT uses IP
- IoT is a broader concept than M2M
- M2M: machine to machine, without human intervention
- IoT: network of devices, connecting systems, people and applications
- IoT uses M2M
- “If you consider M2M in the next larger context, you get the IoT” [Landon Cox]

Communication patterns in IoT – request-response pattern

- Communication pattern: **how** messages are transported in the network to accomplish certain tasks
- Request-response pattern: allows a client to get information in real-time from another client, like HTTP
  - examples: a client asks a server some information (e.g. temperature)
- Properties:
  - if the response is slow to be collected, allow to return partial results to show progress; this might be the case when communicating with devices behind gateways, behind which very slow communication protocols are used
  - interoperability problems: each client needs to know to talk to each server

Communication patterns in IoT – asynchronous messaging pattern

- Also known as “fire-and-forget” exchange
- The sender puts a message in message queue (event queue) and does not require an immediate response to continue processing
- Recipient might be out of office or simply not available
  - e.g. sending an message, and the receiver is not in front of the computer, it is simply shown in the window at appropriate place
  - the destination failed and is recovering, it will answer as soon as the failure is corrected
  - e.g. intermittent connectivity (satellite communication)
- Advantages:
  - solves the problem of intermittent connectivity
- Drawbacks:
  - the broker must ensure that the message is received
  - wait for an answer

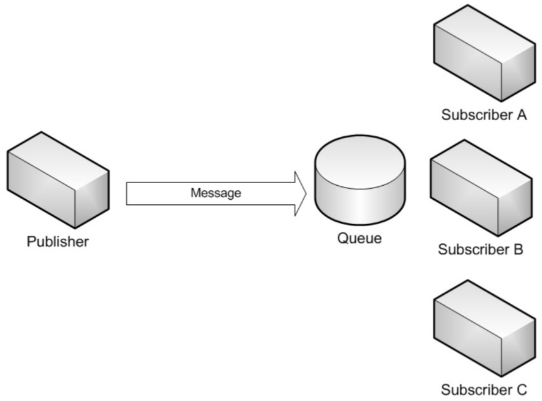


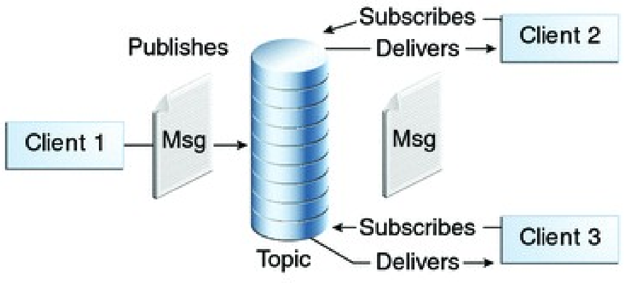
Message queuing – dead letter queue

- The dead letter queue is a service implementation to store messages in case of:
  - message that is sent to a queue that does not exist
  - queue length limit exceeded
  - message length limit exceeded
  - message is rejected by another queue exchange
  - message reaches a threshold read counter number, because it is not consumed (“back out queue”)
- It allows developers to look for common patterns and potential software problems

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

- Read Why the Internet of Things Needs Messaging
- Allows for mass distribution of information to interested parties in an efficient manner
- The publisher of information sends its information only once to the server, which then retransmits it to subscribers
- Allows a client to receive information from all clients of a given class (which have a specific function)
  - example: a smartphone which receives all information from brightness sensors
- Subscribers subscribe to some class of information (e.g. temperature), and when some publisher of that class sends its information to server, the latter automatically informs all the subscribers to that class
- An intermediate machine (server or broker or topic) which dispatches the messages it receives
- Properties:
  - indirect communication, sender does not have to know precisely its receiver
  - asynchronous
  - heterogeneous platforms, easier to change or update



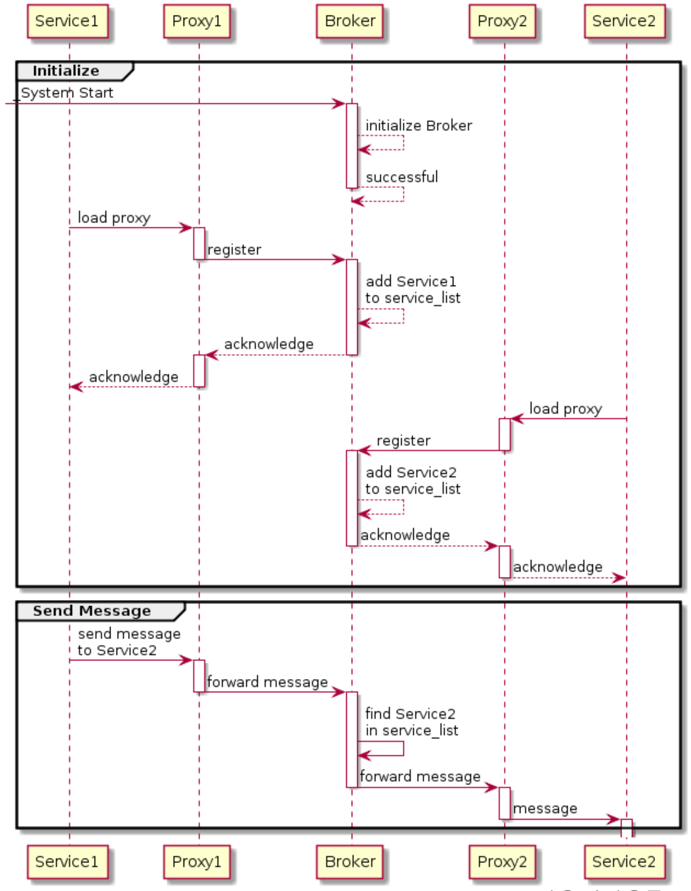


Publish-subscribe pattern implementation

- This pattern consists of clients (publishers or subscribers) communicating with a server (“broker”)
  - broker (*représentant*) = “one who transacts business for another; an agent” (Webster 1913)
- Senders of messages (publishers) do not send messages directly to receivers (subscribers), but to the broker
- Similarly, subscribers express interest in one or more classes and receive only messages that are of interest
- Neither publishers, nor subscribers know each other
- The broker receives messages and distribute them to subscribers which expressed interest in the message topic
- Clients only interact with a broker, but a system may contain several broker servers that exchange data based on their current subscribers' topics

Message broker

- A broker is an intermediary computer program module that translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver
- Purpose: message validation, transformation, and routing
- In our context: routing: take message from sender and, after *filtering*, forward it to appropriate receivers
- If a broker receives a topic for which there are no current subscribers, it will discard the topic unless the publisher indicates that the topic is to be retained (this allows new subscribers to a topic to receive the most current value rather than waiting for the next update from a publisher)
- When a publishing client first connects to the broker, it can set up a default message to be sent to subscribers if the broker detects that the publishing client has unexpectedly disconnected from the broker



(wikipedia)

Message broker

- For example, a message broker may be used to manage a workload queue or message queue for multiple receivers, providing reliable storage, guaranteed message delivery and perhaps transaction management
- Route messages to one or more destinations
- Transform messages to an alternative representation
- Perform message aggregation, decomposing messages into multiple messages and sending them to their destination, then recomposing the responses into one message to return to the user

Communication patterns in IoT – push-pull pattern

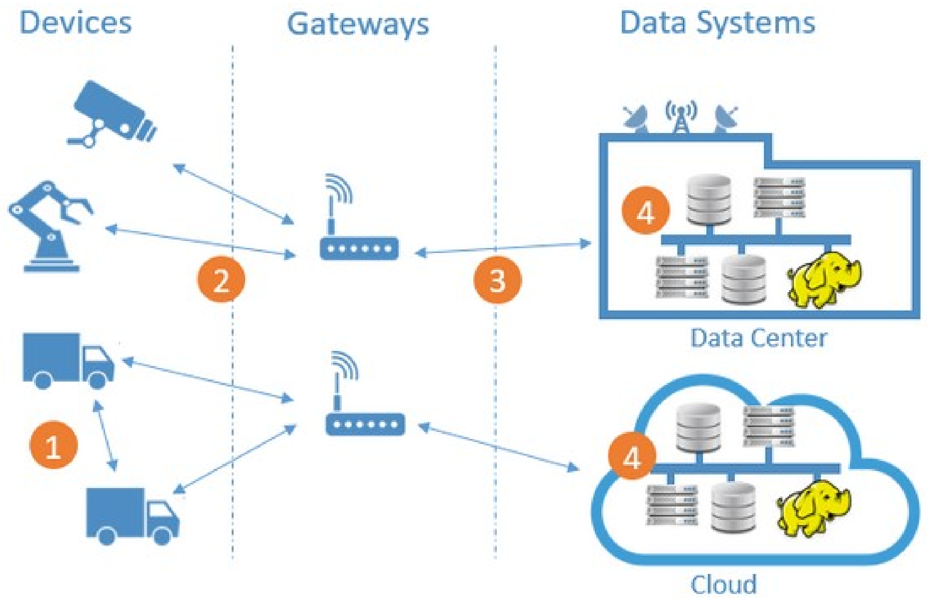
- Or fan-out/fan-in
- It is about resource allocation, contrary to publish-subscribe
- Analogy with firemen: when a fire is detected, the broker sends this information to a car; when another fire occurs, the server selects another car; and so on
  - people (publishers) do not need to know details about firemen cars (number and availability)
  - when a firemen car is added or removed, only the server needs to be informed
- In fan-out, messages are delivered to a pool of workers in a round-robin fashion and each message is delivered to only one worker
- Hence, the difference between pub-sub and push-pull is that a PUB socket sends the same message to **all** subscribers, whereas PUSH does a round-robin amongst all its connected PULL sockets
- The pub/sub pattern is used for wide message distribution according to topics, whereas the push/pull pattern is really a pipelining mechanism
- Fan-in allows to collect the results

Useful features

- Provide the communication patterns presented before
- Simplicity (cf. STOMP)
- Feature-richness (cf. AMQP)
- Popularity (cf. MQTT)
- ...

Homework

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



MQTT

MQTT

- Message Queuing Telemetry Transport
- First version in 1999, currently at version 5.0 (04/2019)
- ISO and OASIS standard
- Available at http://docs.oasis-open.org/mqtt/mqtt/v5.0/ (137 pages)
- Messaging protocol using exclusively the publish-subscribe messaging pattern
- Designed for connections with remote locations where a “small code footprint” is required or the network bandwidth is limited: satellite links, occasional dial-up connections with healthcare providers, a range of home automation and small device scenarios

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

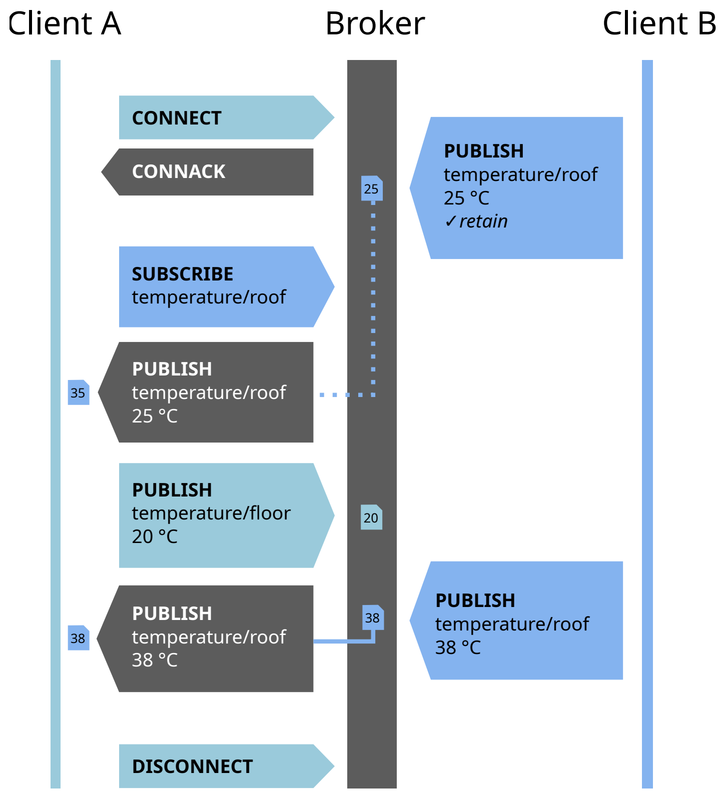
- Data (payload) is organised in a hierarchy of topics
- Senders of messages (publishers) do not send messages directly to receivers (subscribers), but instead categorise their messages into topics and send them to the broker
- Topics are UTF-8 strings, with one or more topics separated by “/”, thus 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”
- # 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 data

- 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 without TCP, such as Zigbee or Bluetooth
- MQTT sends connection credentials in plain text format and does not include any measures for security or authentication; this can be provided by the underlying TCP transport using measures to protect the integrity of transferred information from interception or duplication
- A minimal MQTT control message can have from 2 bytes to nearly 256 MB of data

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
- 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
- details in the specification

MQTT – quality of service (QoS)

- Each connection between broker and client 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 in a two-level handshake to ensure only one copy of the message is received (assured delivery)
- This QoS is done at application level, between MQTT senders and receivers, hence it does not interfere with TCP, which works at transport level
  - for example, 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

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

STOMP

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
- Main 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 to in their particular language”
  - you can use telnet as STOMP client

STOMP commands

- 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 the complete 1.0 specification (very easy to understand)
  - see http://stomp.github.io/stomp-specification-1.2.html#Augmented_BNF for frame BNF

```
CONNECT
login: <username>
passcode:<passcode>

^@  <-- null (ctrl-@)
```

```
CONNECTED
session: <session-id>
^@
```

```
SEND
destination:/queue/a

hello queue a
^@
```

```
SUBSCRIBE
destination: /queue/foo
ack: client

^@
```

```
MESSAGE
destination:/queue/a
message-id: <message-identifier>
hello queue a^@
```

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)
- python-stompy - Implementation of the STOMP protocol in Python
- 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

AMQP

- Advanced Message Queuing Protocol
- OASIS standard 2012, [standard] (125 pages)
- ISO/IEC standard in 2014
- Defining features: message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security
- Application layer
- Binary
- Message delivery guarantee: at most once, at least once, exactly once
- Authentication and/or encryption based on SASL and/or TLS
- It assumes an underlying reliable transport layer protocol such as TCP

AMQP

- AMQP specification spans several layers:
  - a type system – data description, used in messages
  - transport layer – symmetric, asynchronous protocol on top of TCP
  - messaging layer – a standard, extensible message format
  - transactional messaging – two roles: resource and controller
  - security layer – authenticated and/or encrypted transport

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

- Nine AMQP frame bodies are defined, that are used to initiate, control and tear down the transfer of messages between two peers:
  - open (the connection)
  - begin (the session)
  - attach (the link)
  - transfer
  - flow
  - disposition
  - detach (the link)
  - end (the session)
  - close (the connection)

AMQP protocol, 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

- In HTTP, the message can contain a header and data
- 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

- Transaction = several exchanges performed in atomic way: either all of them, or none of them
  - example: money transfer??
- Resource, controller, and coordinator
- Declare and discharge messages are sent by controller to allocate and complete transactions, respectively
- If the control link is closed while there exist non-discharged transactions it created, then all such transactions are immediately rolled back, and attempts to perform further transactional work on them will lead to failure
- Transactional work: posting, acquiring, retiring a message

AMQP security

- Not mandatory, but recommended
- 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-amqplib - pure PHP implementation of the AMQP protocol
- python3-aioamqp - AMQP implementation using asyncio (Python3 version)
- python3-amqp - Low-level AMQP client (Python3 version)
- python3-amqplib - 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)

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

Brokers

Questions

- “RabbitMQ is the most widely deployed open source message broker”
- “Apache ActiveMQ is the most popular and powerful open source messaging and Integration Patterns server”
- Why do they emphasize the broker (server), which is only one part of the system?
- What features are important?

Message broker software

- RabbitMQ, written in Erlang
- Apache ActiveMQ, written in Java
  - both support MQTT, AMQP, STOMP etc.
  - both: multiple platforms, several language bindings
- Mosquitto – MQTT-only
- Apache Qpid – AMQP-only
- Apache RocketMQ – OpenMessaging-only (MQTT and AMQP are planned in the future)
- (OpenMQ, default JMS provider of GlassFish (Java EE – Jakarta EE reference implementation))

RabbitMQ and ActiveMQ Debian packages

- rabbitmq-server - AMQP server written in Erlang
- librabbitmq-client-java - RabbitMQ Java library
- opensips-rabbitmq-module - Interface module to interact with a RabbitMQ server – It is used to send AMQP messages to a RabbitMQ server each time the Event Interface triggers an event subscribed for
- activemq - Java message broker – server
- libactivemq-java - Java message broker core libraries

CoAP

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 µcontrollers”)
- => **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

- 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 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

HTTP header for requests/responses

Web page request:

```
GET /index.php HTTP/1.1
Host: rt.pu-pm.univ-fcomte.fr
User-Agent: Mozilla/5.0 (X11; Linux x86_64;
  rv:56.0) Gecko/20100101 Firefox/56.0
Accept: text/html,application/xhtml+xml,
  application/xml;q=0.9,*/*;q=0.8
Accept-Language: en
Accept-Encoding: gzip, deflate
Cookie: wiki_rt_session=[...]
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Referer: ...
```

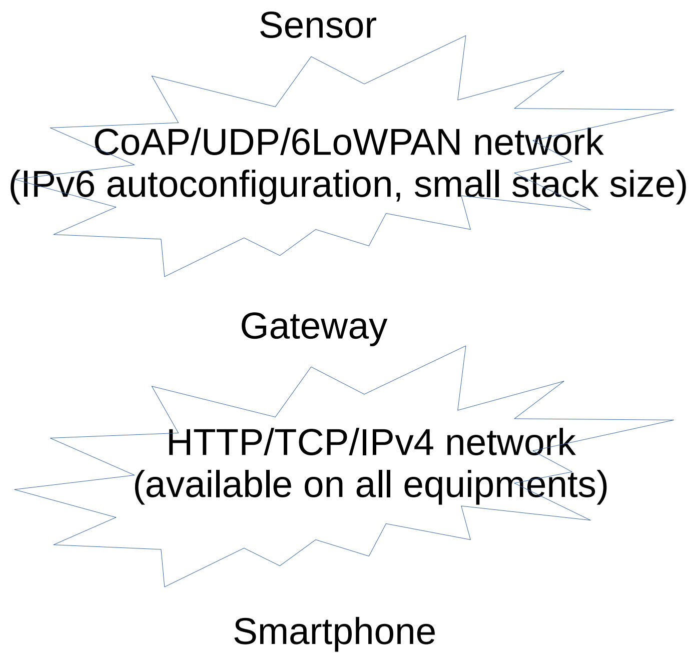
Web page response:

```
HTTP/1.1 200 OK
Date: Fri, 03 Nov 2018 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 2018 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>
...
```

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


Sensor
CoAP/UDP/6LoWPAN network
(IPv6 autoconfiguration, small stack size)
Gateway
HTTP/TCP/IPv4 network
(available on all equipments)
Smartphone

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver| T |  TKL  |      Code     |          Message ID           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Token (if any, TKL bytes) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Options (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 1 1 1 1 1|    Payload (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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

```
+----------------------+
|      Application     |
+----------------------+
+----------------------+  \
|  Requests/Responses  |  |
|----------------------|  | CoAP
|       Messages       |  |
+----------------------+  /
+----------------------+
|          UDP         |
+----------------------+
```

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)

```
Client              Server
   |                  |
   |   CON [0x7d34]   |
   +----------------->|
   |                  |
   |   ACK [0x7d34]   |
   |<-----------------+
   |                  |
```

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

```
Client              Server
   |                  |
   |   NON [0x01a0]   |
   +----------------->|
   |                  |
```

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

```
Client              Server       Client              Server
   |                  |             |                  |
   |   CON [0xbc90]   |             |   CON [0xbc91]   |
   | GET /temperature |             | GET /temperature |
   |   (Token 0x71)   |             |   (Token 0x72)   |
   +----------------->|             +----------------->|
   |                  |             |                  |
   |   ACK [0xbc90]   |             |   ACK [0xbc91]   |
   |   2.05 Content   |             |  4.04 Not Found  |
   |   (Token 0x71)   |             |   (Token 0x72)   |
   |     "22.5 C"     |             |   "Not found"    |
   |<-----------------+             |<-----------------+
   |                  |             |                  |
```

CoAP Debian packages

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

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?
- MQTT vs CoAP:
  - MQTT: many2many, broker, appropriate for live data (event-based), TCP (e.g. NAT-friendly)
  - CoAP: one2one, direct communication, suited for state transfer, UDP (e.g. NAT-unfriendly)

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/appclient
- 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
- => unappropriate to our lab

RPL, IPv6 Routing Protocol for Low-Power and Lossy Networks

RPL

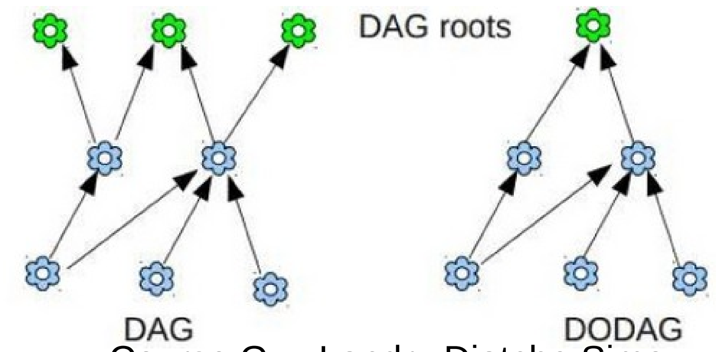
- RFC 6550 (standardised in 2012, 154 text pages), with special applications (home and industrial automation, urban networks) specified in other RFCs
- Pronunciation: ripple (“vaguelette, ondulation”)
- Proactive routing protocol, specifies a mechanism to disseminate information over dynamically formed network topology
- Multi-hop, can support thousands of nodes
- Operates (usually??) on IEEE 802.15.4, which provides physical and MAC layers for protocols of LR-WPAN (low-rate wireless PAN), LLN (like CoAP)
  - used also by Zigbee and 6LoWPAN (IPv6 over low-power wireless PAN)

RPL features

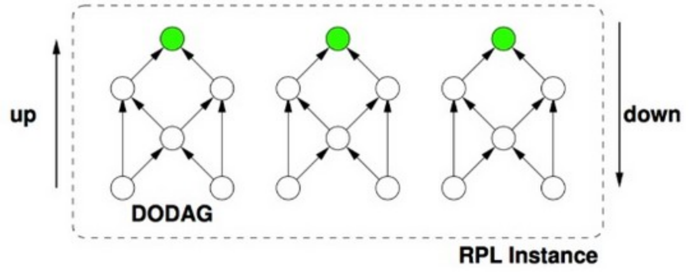
- Optimised for point-to-multipoint (one-to-many, from a central control point to a subset of devices inside the LLN) and multipoint-to-point (many-to-one, from devices inside the LLN towards a central control point), but supports also one-to-one (point-to-point, between devices inside the LLN) communications
- 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 and routers at the same time
- Supports a wide variety of link layers (contradiction??)
- Designed to be highly adaptive to network conditions, and to provide alternate routes when default ones become unusable
- RPL quickly creates network routes, shares routing knowledge and adapts the topology efficiently

RPL topology

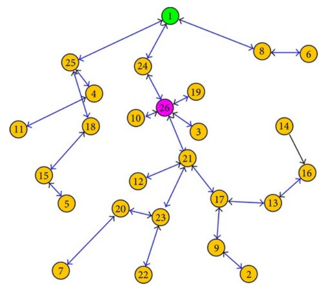
- LLNs do not typically have predefined topologies, so a topology needs to be created
- RPL creates a topology similar to a tree (DAG, directed acyclic graph)
  - all edges are oriented
  - no cycles exist
  - can have several roots
- Rank = distance to a root node (0 for the root), such as number of hops
- DODAG (destination-oriented DAG) = DAG with a single root node
- A DODAG root may act as a border router for the DODAG, and could route to external world through a common backbone usin other protocols



Course Guy Landry Djatche Simo



Course Pradeep Kumar



Linux Wallgren et al.

Examples

A single DODAG:

- A DODAG optimized to minimize latency rooted at a single centralized lighting controller in a Home Automation application

Multiple uncoordinated DODAGs with independent roots:

- Multiple data collection points in an urban data collection application that do not have suitable connectivity to coordinate with each other or that use the formation of multiple DODAGs as a means to dynamically and autonomously partition the network

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 instances and objective function

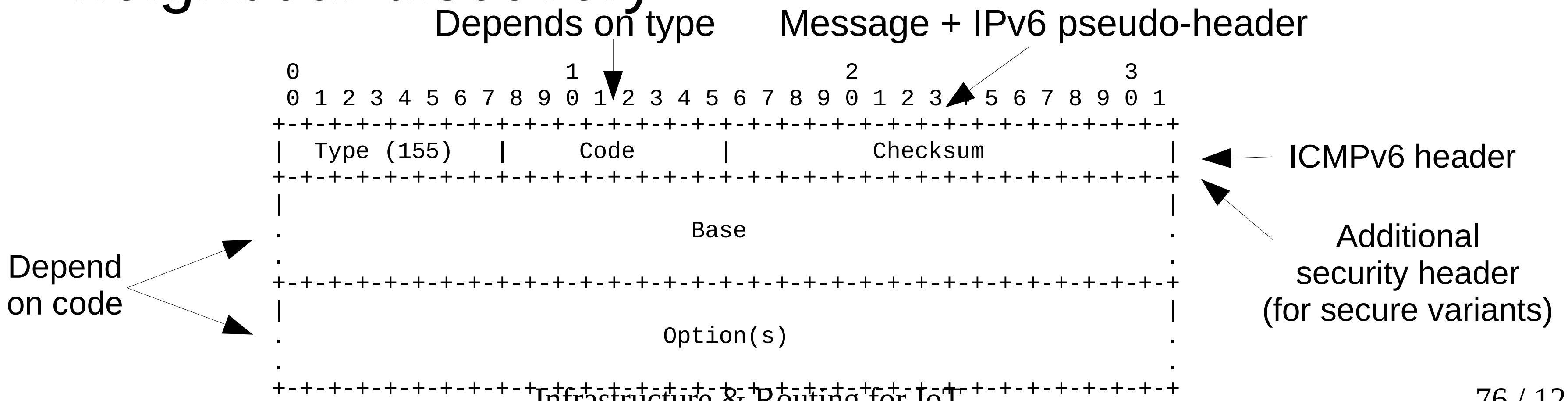
- A DODAG has an optimisation objective, computed using an objective function and defined by the application, e.g.:
  - distance in terms of number of hops to root
  - transmission energy minimisation, node remaining energy
  - latency minimisation
  - throughput maximisation
  - node availability, link reliability
  - etc., or application-specific constraints fulfilling
- 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

MRHOF as an example of objective function

- Minimum Rank with Hysteresis Objective Function (MRHOF) (RFC 6719) – minimize a metric, while using hysteresis to reduce churn in response to small metric changes
- First, find the minimum cost path, i.e. path with the minimum rank
- Second, switch to that minimum rank path only if it is shorter (in terms of path cost) than the current path by at least a given threshold (this second mechanism is called "hysteresis")
- MRHOF may be used with any additive metric as long as the routing objective is to minimize the given routing metric
- Nodes must support at least one of these metrics: hop count, latency, or ETX (expected transmission count for successfully reception)
- (Another example: OF0)

RPL control message

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


Depends on type
Message + IPv6 pseudo-header
ICMPv6 header
Additional security header (for secure variants)
Depend on code

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Type (155)   |     Code      |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
.                             Base                              .
.                                                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
.                           Option(s)                           .
.                                                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

RPL message types (codes)

- 0x00: DIS (DODAG Information Solicitation)
- 0x01: DIO (DODAG Information Object)
- 0x02: DAO (Destination Advertisement Object)
- 0x03: DAO-ACK (DAO Acknowledgment)

Secure variants, using CCM and AES-128 encryption algorithms:

- 0x80: Secure DIS
- 0x81: Secure DIO
- 0x82: Secure DAO
- 0x83: Secure DAO-ACK

- 0x8A: Consistency Check

RPL message codes

- DIS (solicitation): sent to a RPL node to request information from nearby DODAG, analogous to router request messages used to discover existing networks
- DIO (information): usually sent in response to DIS messages
  - contains information about a RPL instance, its configuration parameters (topology)
  - allows to select a DODAG parent set, and maintain the DODAG

RPL message codes

- DAO (advertisement): sent by the teams to propagate/update the information of their “parent” nodes throughout the DAG
  - sent by nodes towards root, either to a selected parent (storing mode), or to the root (non-storing mode)
  - a DODAG can work only in one mode at a time
- DAO-ACK: answer to a DAO, unicast

Upward routes – discovery and maintenance

- Typical LLNs exhibit variations in physical connectivity that are transient and harmless to traffic, so a routing protocol 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

Trickle algorithm: ensuring consistency with very low overhead

- The Trickle algorithm establishes a density-aware local communication primitive with an underlying consistency model that guides when a node transmits. 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). Instead of flooding a network with packets, the algorithm controls the send rate so each node hears a small trickle of packets, just enough to stay consistent.
- it is simple to implement; and requires very little state. Current implementations use 4–11 bytes of RAM and are 50–200 lines of C code
- Trickle's basic primitive is simple: every so often, a node transmits data unless it hears a few other transmissions whose data suggest its own transmission is redundant
- Details: RFC 6206 (2011)

Trickle algorithm

- Parameters:
  - Imin, minimum interval, e.g. 100ms
  - Imax, number of doublings, e.g. 16
  - k, redundancy constant
- Variables:
  - I, current interval size
  - t, time within the current interval I
  - c, counter

Simplified algorithm:

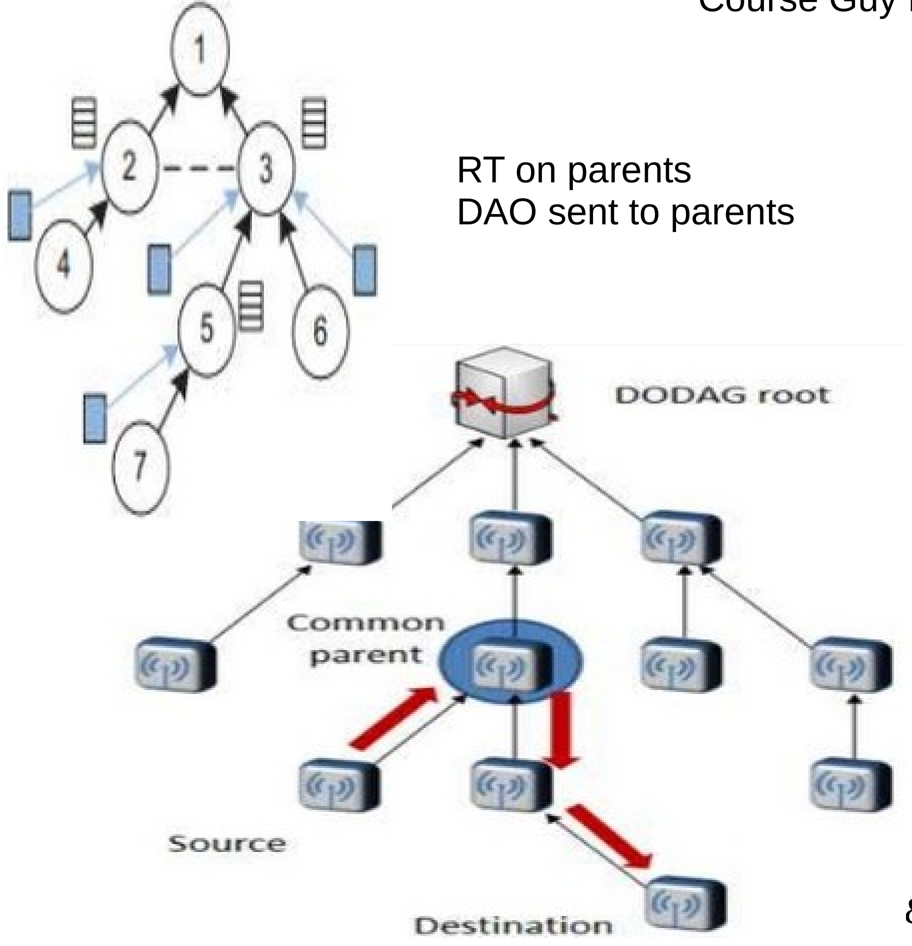
- I in [Imin,Imax]
- c=0, t random in [I/2,I)
- At consistent reception, c++
- At t, node transmits iff c<k
- When I expires, I doubles
- At inconsistent reception (or at external “event”), I=Imin

Downward routes

- Nodes send DAO messages containing their sub-DODAG
- Parents aggregate them and thus discover downward routes

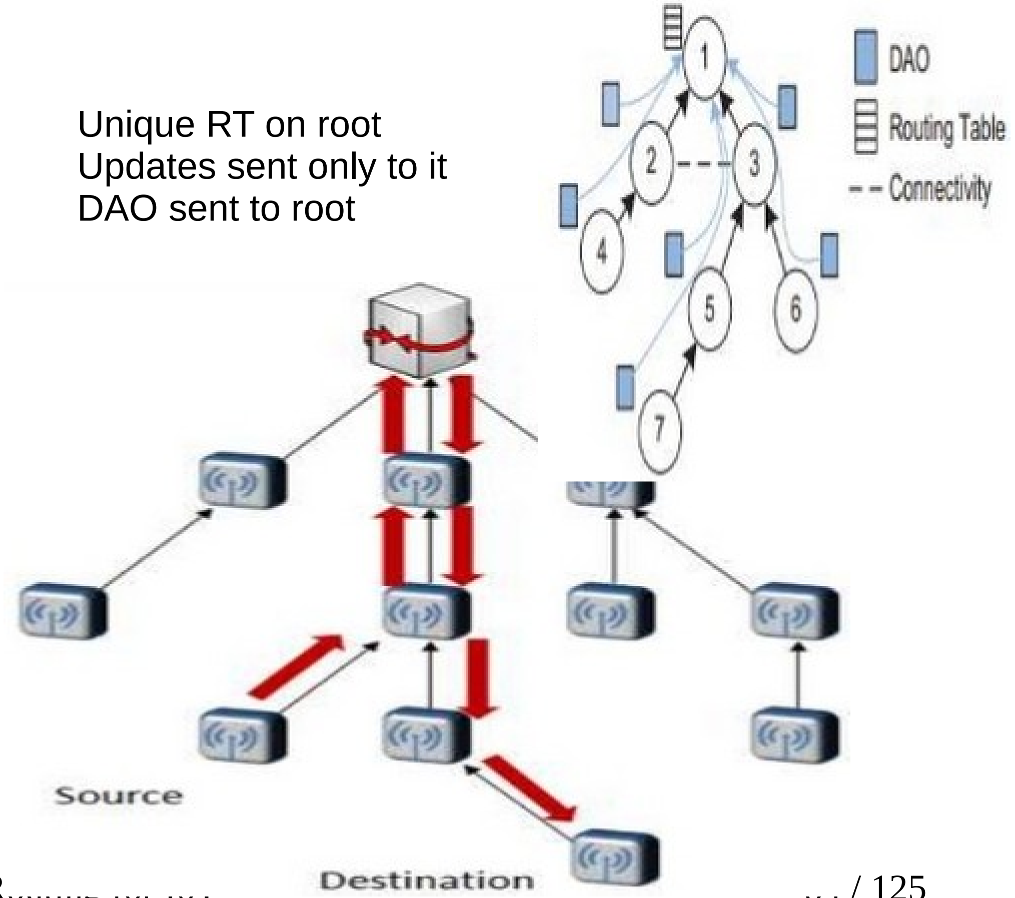
Storing and non-storing modes

Course Guy Landry Djatche Simo



RT on parents
DAO sent to parents

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



DODAG protection

- Various mechanisms are available to avoid loops, incoherences and to repair the graph
- Loop avoidance and detection:
  - uses IPv6's RPL option [RFC 6553]
  - bit indicating transmission direction (up or down)
  - upon message reception, a node compares its rank with the rank of sending node
- Poisoning = setting rank to ∞ (poisoning) to avoid loop

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 have 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

Homework

- More information:
  - section 2.3.2 from Kamgueu's PhD thesis (in French)
  - slides 1–27 of The IoT/IP protocols
  - read the whole RFC 6206 (Trickle)

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?

Conclusions

Conclusions

- IoT uses specific protocols, at application level generally (MOM)
- Several protocols and API for MOM exist
- The broker links all objects together, no matter the protocol
- Messaging paradigms: pub-sub, asynchronous messaging, peer to peer
- Some protocols target low resources, others simplicity
- RPL creates routes in dynamic networks

STOP

XMPP

XMPP

- Extensible Messaging and Presence Protocol (XMPP)
- IETF standard, RFC 6120 (core, 211 text pages), 6121 (IM, 114 text pages), 7622, ...
- Much more complex than MQTT
- Developed in 1999 (previously called Jabber) for near real-time instant messaging (IM), presence information, and contact list maintenance
- Based on XML (Extensible Markup Language)
- Enables the near-real-time exchange of structured yet extensible data between any two or more network entities
- No QoS currently

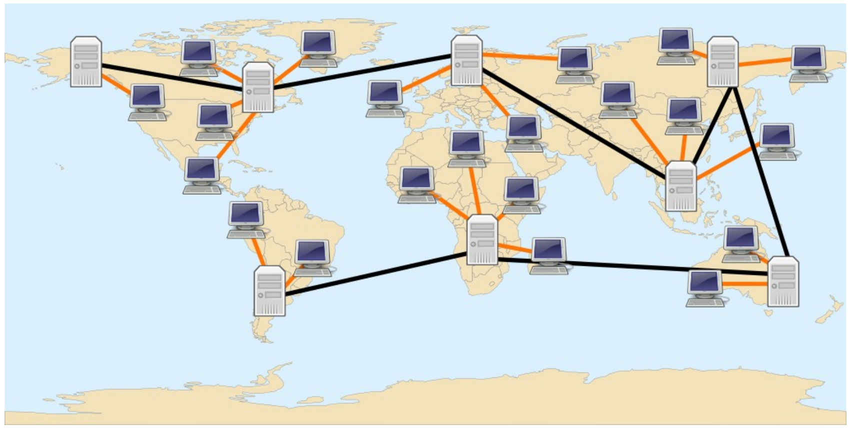
Instant messaging big picture

- Traditionally, IM applications have combined the following factors:
  - the central point of focus is a list of one's contacts or "buddies" (in XMPP this list is called a "roster")
  - the purpose of using such an application is to exchange relatively brief text messages with particular contacts in close to real time -- often relatively large numbers of such messages in rapid succession, in the form of a one-to-one "chat session"
  - the catalyst for exchanging messages is "presence" -- i.e. information about the network availability of particular contacts (thus knowing who is online and available for a one-to-one chat session)
  - presence information is provided only to contacts that one has authorized by means of an explicit agreement called a "presence subscription"
- Thus at a high level this document assumes that a user needs to be able to complete the following use cases:
  - manage items in one's contact list
  - exchange messages with one's contacts
  - exchange presence information with one's contacts
  - manage presence subscriptions to and from one's contacts

XMPP – architecture

- Client-server architecture
- Each client connect to a sever
- Clients communicate only with their server
- Distributed system, anyone can run its own server and there is no central server, like e-mail

```
 example.net <--------------> im.example.com
     ^                               ^
     |                               |
     v                               v
romeo@example.net           juliet@im.example.com
```



XMPP – applications

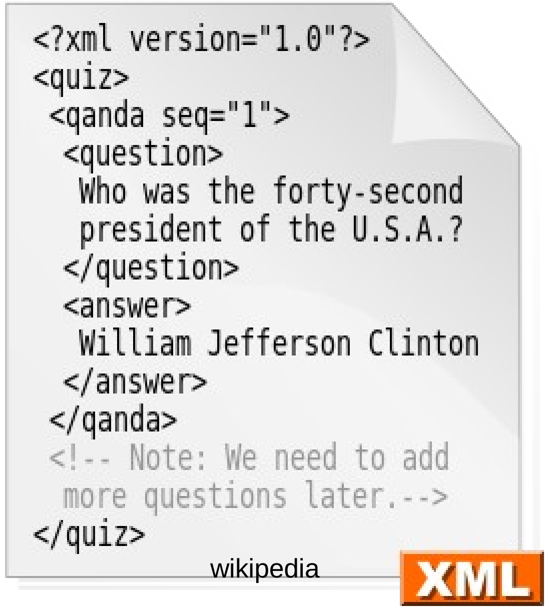
- Numerous libraries (e.g. loudmouth) and applications
- Designed to be extensible, it has been used also for publish-subscribe systems, signalling for VoIP, video, file transfer, gaming, the Internet of Things (IoT) applications such as the smart grid, and social networking services
- Several big companies integrated support for XMPP in their product, and dropped it later
  - “The WhatsApp protocol is a slightly modified version of XMPP; it is deliberately modified so that it can only be used with the WhatsApp program available from the company of the same name”
  - “XMPP can support all the features that WhatsApp does, without the spying and poor security. Not all XMPP clients support all features, but the popular ones will support photo sharing, VoiP etc.”
  - “Facebook Chat and Gtalk internally use the Jabber protocol. Unlike WhatsApp, these companies used to allow reguular XMPP clients to connect to their chat service, so you could use any Jabber program to chat with someone on Facebook or Gtalk.”
  - “Since I originally wrote this guide (early 2013; now mid 2017) this seems to be no longer the case; you can only connect to Facebook with their proprietary client”

XMPP messages

- Messages are based on XML
- On top of TCP, but can also on top of HTTP, allowing to bypass firewalls in Internet
- Three types of XML stanzas (complete XML fragment):
  - message – “push” mechanism, asynchronous messaging
  - presence (network availability), such as online/offline and available/busy etc. – publish-subscribe model with delayed delivery (like MQTT)
  - IQ (info/query or request-response) – HTTP-like
- Every user has a unique address, similar to an e-mail address: name@address, to which can be added a resource, e.g. /mobile, separating several clients belonging to the same user
- Each resource can have a priority; sending to name@address will send to its highest priority resource

XML

- Markup language (similar to HTML)
- Human-readable and machine-readable
- Goals: simplicity, generality, usability across Internet



```
<?xml version="1.0"?>
<quiz>
 <qanda seq="1">
  <question>
   Who was the forty-second
   president of the U.S.A.?
  </question>
  <answer>
   William Jefferson Clinton
  </answer>
 </qanda>
 <!-- Note: We need to add
  more questions later.-->
</quiz>
```

wikipedia

XMPP stream exchange, RFC 6120

- TCP
- TLS (through STARTTLS extension): secure the stream from tampering (modify) and eavesdropping (see)
- SASL: authentication
- XMPP

```
I: <?xml version='1.0'?>
   <stream:stream
       from='juliet@im.example.com'
       to='im.example.com'
       version='1.0'
       xml:lang='en'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'>

R: <?xml version='1.0'?>
   <stream:stream
       from='im.example.com'
       id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
       to='juliet@im.example.com'
       version='1.0'
       xml:lang='en'
       xmlns='jabber:client'
       xmlns:stream='http://etherx.jabber.org/streams'>
```

XMPP bindings

- Standard binding, using XML stanzas
- BOSH - Bidirectional streams over Synchronous HTTP – through HTTP
- EXI, Efficient XML Interchange – compressed XML, however its proposed specification has expired...

XMPP Debian packages

- prosody - Lightweight Jabber/XMPP server
- ejabberd - distributed, fault-tolerant Jabber/XMPP server
- python3-sleekxmpp - XMPP (Jabber) Library Implementing Everything as a Plugin (Python 3.x)
- libtaningia0 - Taningia is a generic communication library based on XMPP
- psi-plus - Qt-based XMPP client (basic version)
- ... and much more!

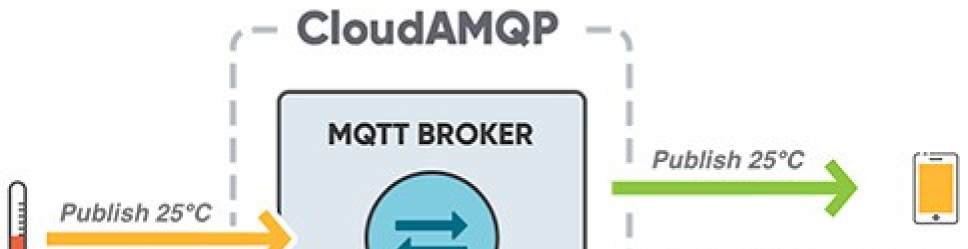
XMPP – questions

- How can someone use XMPP to get the status of the sensors in its home?
- What are the attributes which can be put in all the three XML stanzas?

Various interesting random slides

Pub-sub model

- HTTP is synchronous – the Web page does not complete until server has answered
- 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)
- Consider a temperature sensor in your greenhouse, which is measuring the temperature. Your sensor could send the temperature to a **message queue** in a 15-minute interval (about 96 times a day), instead of processing the data in the greenhouse and be connected all the time.

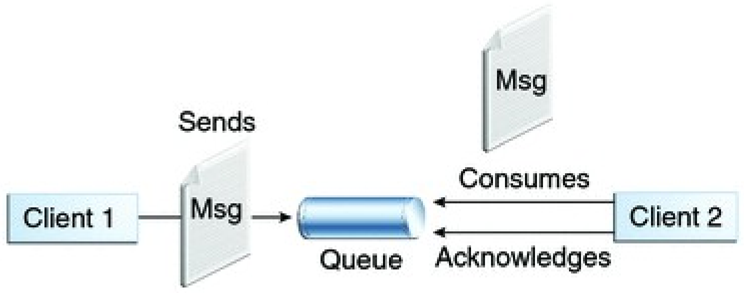


Publish/Subscribe

- each message can have multiple consumers
- publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages

Point-to-point

- Read page https://docs.oracle.com/javaee/6/tutorial/doc/bncdr.html
- A point-to-point (PTP) product or application is built on the concept of message queues, senders, and receivers
- Queues retain all messages sent to them until the messages are consumed or expire
  - each message has only one consumer
  - a sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message
  - the receiver acknowledges the successful processing of a message



- An enterprise application provider is likely to choose a messaging API over a tightly coupled API, such as a remote procedure call (RPC), under the following circumstances
  - The provider wants the components not to depend on information about other components’ interfaces, so components can be easily replaced
  - The provider wants the application to run whether or not all components are up and running simultaneously
  - The application business model allows a component to send information to another and to continue to operate without receiving an immediate response

- Messaging = a method of communication between software componets and applications, loosely coupled
  - it is different than e-mail, which is used by **people**
- On the contrary, RMI (Remote Method Invocation) is tightly coupled, since the sender must know recipient's methods, and both parties need to be available at the same time
- Sockets too need that both parties be available, whereas queuing takes the message and sends it to destination when it becomes available

Communication patterns in IoT – router-dealer (shared queue) pattern – questions

- How it works
- Find the difference compared to the simpler pub-sub pattern

What IoT 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:
  - allows only direct communication
  - heavy (e.g. see HTTP header)
  - etc.

AMQP – questions

- What are the AMQP features concerning flow control?
- How can packets get reordered when in transit?

AMQP TODO

- Provides:
  - pub-sub
  - point-to-point
  - request-response
  - fan-out ??
  - reliability

AMQP architecture

- The link protocol transfers messages between two nodes but assumes very little as to what those nodes are or how they are implemented
- A key category is those nodes used as a rendezvous point between senders and receivers of messages (e.g. queues or topics). The AMQP specification calls such nodes distribution nodes and codifies some common behaviors
- This includes:
  - some standard outcomes for transfers, through which receivers of messages can for example accept or reject messages
  - a mechanism for indicating or requesting one of the two basic distribution patterns, competing- and non-competing- consumers, through the distribution modes move and copy respectively
  - the ability to create nodes on-demand, e.g. for temporary response queues
  - the ability to refine the set of message of interest to a receiver through filters
- Though AMQP can be used in simple peer-to-peer systems, defining this framework for messaging capabilities additionally enables interoperability with messaging intermediaries (brokers, bridges etc.) in larger, richer messaging networks

Protocol comparison

MQTT

- It’s important to understand the class of use that each of these important protocols [mqtt, xmpp etc.] addresses
- MQTT targets device data collection (Fig. 3). As its name states, its main purpose is telemetry, or remote monitoring. Its goal is to collect data from many devices and transport that data to the IT infrastructure. It targets large networks of small devices that need to be monitored or controlled from the cloud.
- MQTT enables applications like monitoring a huge oil pipeline for leaks or vandalism. Those thousands of sensors must be concentrated into a single location for analysis. When the system finds a problem, it can take action to correct that problem. Other applications for MQTT include power usage monitoring, lighting control, and even intelligent gardening. They share a need for collecting data from many sources and making it available to the IT infrastructure.
- a lightweight broker-based publish/subscribe messaging protocol designed to be open, simple, lightweight and easy to implement
- It targets large networks of small devices that need to be monitored or controlled from the cloud

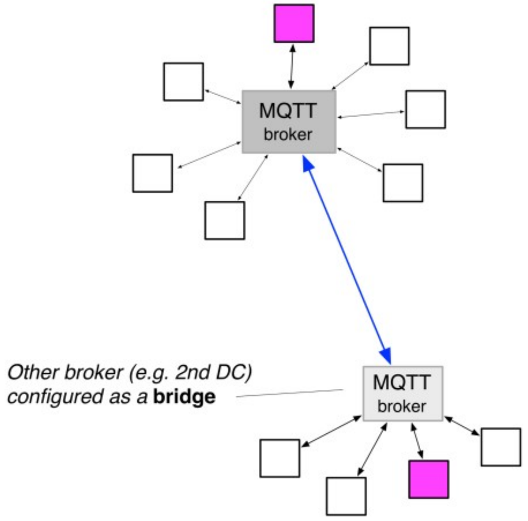
MQTT

- The design principles and aims of MQTT are much more simple and focused than those of AMQP—it provides publish-and-subscribe messaging (no queues, in spite of the name) and was specifically designed for resource-constrained devices and low bandwidth, high latency networks such as dial up lines and satellite links, for example. Basically, it can be used effectively in embedded systems.
- One of the advantages MQTT has over more full-featured “enterprise messaging” brokers is that its intentionally low footprint makes it ideal for today’s mobile and developing “Internet of Things” style applications.
- MQTT’s strengths are simplicity (just five API methods), a compact binary packet payload (no message properties, compressed headers, much less verbose than something text-based like HTTP), and it makes a good fit for simple push messaging scenarios such as temperature updates, stock price tickers, oil pressure feeds or mobile notifications. It is also very useful for connecting machines together, such as connecting an Arduino device to a web service with MQTT.

MQTT

- mainly used when a huge network of small devices needs to be monitored or managed via the Internet, i.e. parking sensors, underwater lines, energy grid, etc.
- Pros
- Lightweight for constrained networks
- Flexibility to choose Quality of Services with the given functionality
- Standardized by OASIS Technical Committee
- Easy and quick to implement
- Cons
- High power consumption due to the TCP-based connection
- Lack of encryption [??????????? The Mosquitto broker supports TLS out of the box]
- Use Case
- A parking lot where there are a number of parking sensors installed to identify the number and location of empty or vacant parking spots

MQTT



- Mosquitto [or MQTT ??] can be configured as a so-called “bridge”. I could imagine this being useful in, say, different data centers.
- In a bridge configuration, Mosquitto is configured to pass certain topics in certain directions. For example, I could configure a bridge to notify a “central” broker for messages of topic +/important.
- Mosquitto periodically publishes statistics which interested parties can subscribe to, e.g. for monitoring purposes.
- The Mosquitto project has a test server http://test.mosquitto.org/ you can use if you don’t want to set up your own (just launch mosquitto_sub at it),

```
$SYS/broker/version mosquitto version 1.1
$SYS/broker/clients/total 368
$SYS/broker/clients/active 91
$SYS/broker/clients/inactive 277
$SYS/broker/clients/maximum 368
$SYS/broker/messages/received 13358099
$SYS/broker/messages/sent 16381123
$SYS/broker/messages/dropped 414180
$SYS/broker/messages/stored 10806
$SYS/broker/messages/sent 16381123
$SYS/broker/messages/sent 16381123
$SYS/broker/bytes/received 761223497
$SYS/broker/bytes/sent 476065843
$SYS/broker/load/bytes/sent/1min 28745.93
$SYS/broker/load/bytes/sent/5min 15418.24
$SYS/broker/load/bytes/sent/15min 9980.69
[...]
```

XMPP

- Its key strength is a name@domain.com addressing scheme that helps connect the needles in the huge Internet haystack.
- In the IoT context, XMPP offers an easy way to address a device
- XMPP provides a great way, for instance, to connect your home thermostat to a Web server so you can access it from your phone. Its strengths in addressing, security, and scalability make it ideal for consumer-oriented IoT applications.
- Cons:
  - Text-based messaging, no end-to-end encryption provision
  - No Quality of Service provision
- Use cases:
  - A smart thermostat that can be accessed from a smartphone via a web server
  - A gaming console with instant messaging between the two online players

AMQP

- AMQP is all about queues
- AMQP is focused on not losing messages
- The standard also describes an optional transaction mode with a formal multiphase commit sequence
- AMQP is mostly used in business messaging
- True to its origins in the banking industry, AMQP focuses on tracking messages and ensuring each message is delivered as intended, regardless of failures or reboots.
- Pros
-   Messages can be sent over TCP and UDP
-   Provides end-to-end encryption
- Cons
-   Relatively high resource utilization, i.e. power and memory usage
- Use Cases
  - AMQP is mostly used in business messaging. It usually defines devices like mobile handsets, communicating with back-office data centers

AMQP

- Two of the most important reasons to use AMQP are reliability and interoperability. As the name implies, it provides a wide range of features related to messaging, including reliable queuing, topic-based publish-and-subscribe messaging, flexible routing, transactions, and security. AMQP exchanges route messages directly—in fanout form, by topic, and also based on headers.
- There’s a lot of fine-grained control possible with such a rich feature set. You 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. This protocol was designed for reliability at the many large companies who depend on messaging to integrate applications and move data around their organisation. In the case of RabbitMQ, there are many different language implementations and great samples available, making it a good choice for building large scale, reliable, resilient, or clustered messaging infrastructures.

AMQP

- Companies like JP Morgan use it to process 1 billion messages a day. NASA uses it for Nebula Cloud Computing. Google uses it for complex event processing. Here are a couple of additional AMQP examples and links:
- It is used in one of the world’s largest biometric databases India’s Aadhar project—home to 1.2 billion identities.
- It is used in the Ocean Observatories Initiative—an architecture that collects 8 terabytes of data per day.
- More examples and links are available at amqp.org.

STOMP

- The design principles here were to create something simple, and widely-interoperable. For example, it’s possible to connect to a STOMP broker using something as simple as a telnet client.
- STOMP does not, however, 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.
- However, STOMP is simple and lightweight (although somewhat verbose on the wire), with a wide range of language bindings. It also provides some transactional semantics. One of the most interesting examples is with RabbitMQ Web Stomp which allows you to expose messaging in a browser through websockets. This opens up some interesting possibilities—like updating a browser, mobile app, or machine in real-time with all types of information.

Comparison between MQTT and XMPP

- MQTT: publish-subscribe model
- XMPP: P-S model, IQ (request-response),

- The thing I don’t like about MQTT is that every device must be configured with the address of a broker. This is an extra configuration step, and the broker can become a single point of failure; if it goes down, all devices are unusable. I prefer point-to-point self configuring protocols such as ZeroMQ
- A high level overview of the protocols behind such communication include MQTT, XMPP, DDS, and AMQP. Although the four protocols do overlap somewhat in nature, each generally targets a specific function in IoT. MQTT targets device data collection. XMPP is ideal for consumer oriented applications as it focuses on addressing, security, and scalability. DDS concentrates on devices that use device data. Finally, AMQP is primarily a queueing protocol.
- https://www.electronicdesign.com/iot/understanding-protocols-behind-internet-things