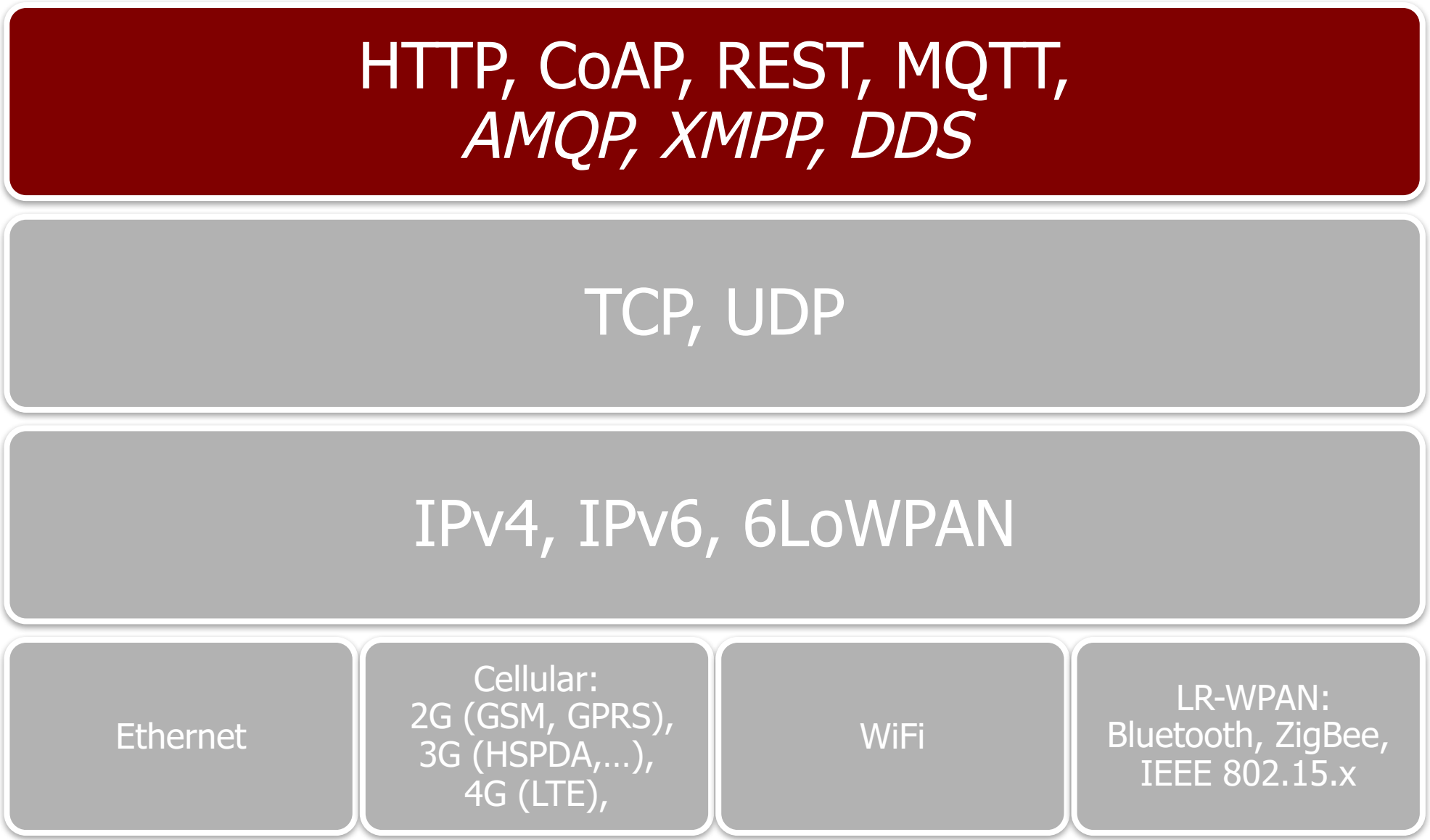
REST and MQTT

An IoT oriented comparison

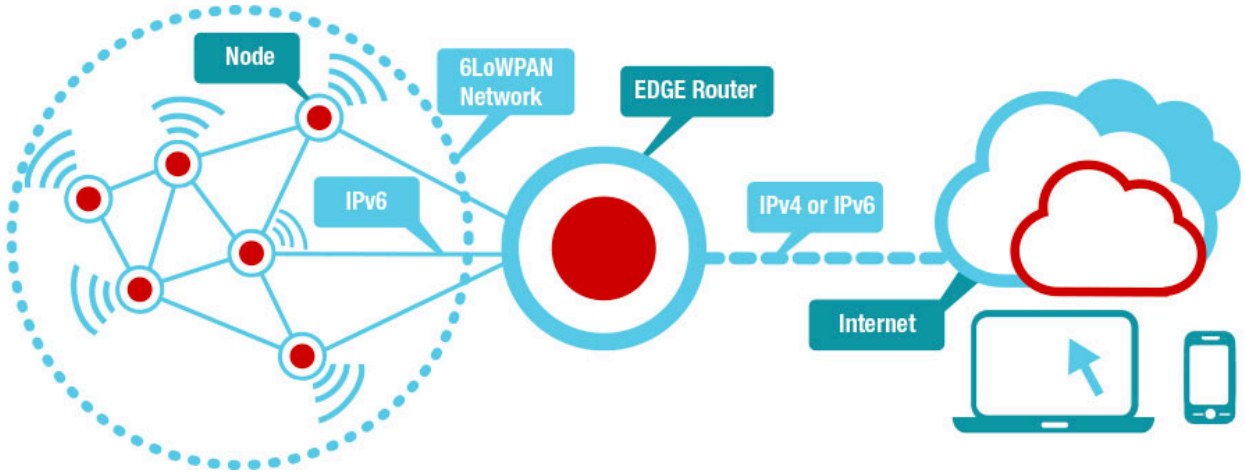
Credits: Prof.Pietro Manzoni, University of Valencia, Spain

IoT Protocols



Network layer protocols

- The network (or OSI Layer 3 abstraction) provides an abstraction of the physical world.
- Communication protocols
  - Most of the IP-based communications are based on the IPv4 (and often via gateway middleware solutions)
  - However, IP overhead makes it inefficient for embedded devices with low bit rate and constrained power.
    - IPv6 is increasingly being introduced for embedded devices ➔ 6LowPAN (*IPv6 over Low power Wireless Personal Area Networks*)



IoT standards: OASIS

OASIS
Advancing open standards for the information society

- ✓ OASIS is a nonprofit consortium that drives the development, convergence and adoption of open standards for the global information society.
- ✓ OASIS promotes industry consensus and produces worldwide standards for security, Internet of Things, cloud computing, energy, content technologies, emergency management, and other areas.

OASIS Committee Categories: IoT/M2M

Technical Committees:

OASIS Advanced Message Queuing Protocol (AMQP) Bindings and Mappings (AMQP-BINDMAP) TC
Defining bindings and mappings of AMQP wire-level messaging protocol for real-time data passing and business transactions

OASIS Advanced Message Queuing Protocol (AMQP) TC
Defining a ubiquitous, secure, reliable and open internet protocol for handling business messaging.

OASIS Classification of Everyday Living (COEL) TC
Developing transparent technical frameworks to support a business ecosystem aimed at providing personalized services

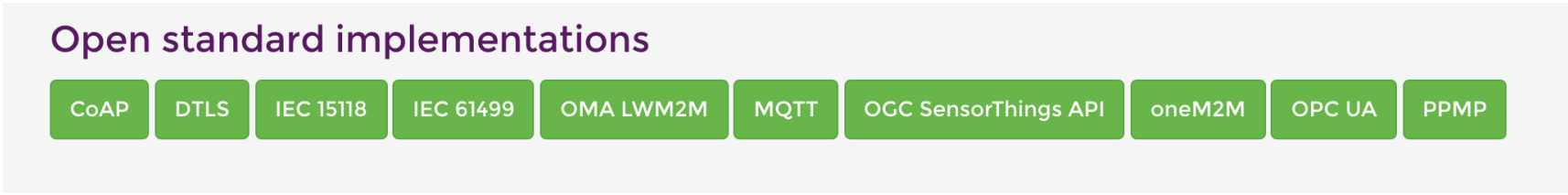
OASIS Message Queuing Telemetry Transport (MQTT) TC
Providing a lightweight publish/subscribe reliable messaging transport protocol suitable for communication in M2M/IoT contexts where a small code footprint is required and/or network bandwidth is at a premium.

OASIS OCPP Electric Vehicle Charging Equipment Data Exchange TC
Assuring interoperable exchange of electric vehicle (EV) and EV supply equipment (EVSE) charging station data

OASIS Open Building Information Exchange (oBIX) TC
Enabling mechanical and electrical control systems in buildings to communicate with enterprise applications

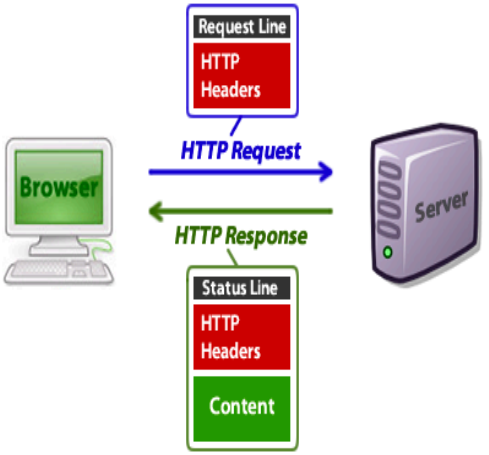
IoT standards: open-source implementations

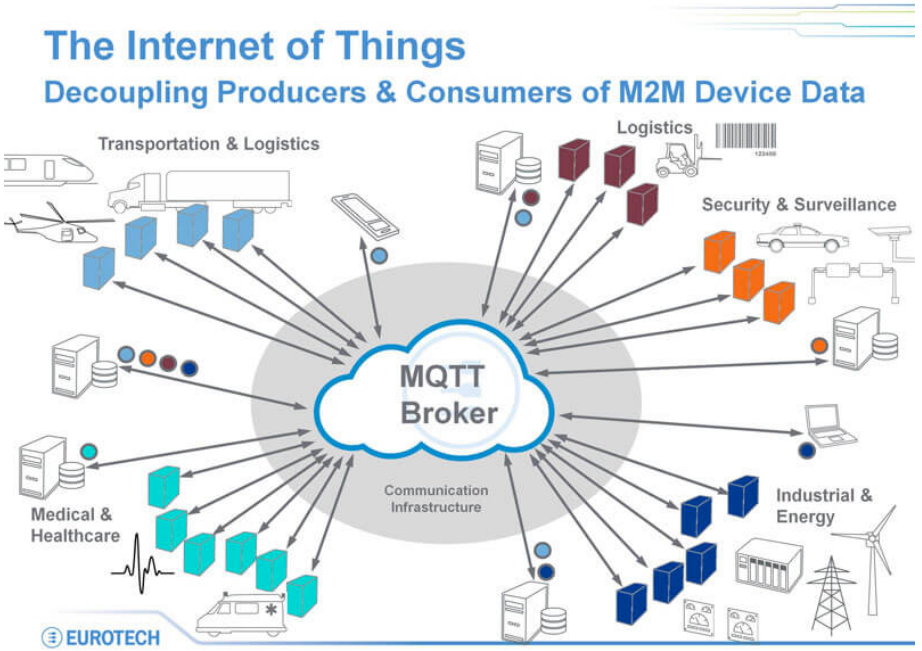
- While Open Standards are key, it is also important to make available open-source implementations of such standards, to encourage adoption of such standards both by IoT developers and the IoT industry at large.
- http://iot.eclipse.org/
  - Eclipse IoT is an ecosystem of companies and individuals that are working together to establish an Internet of Things based on open technologies.



"Message based" communications

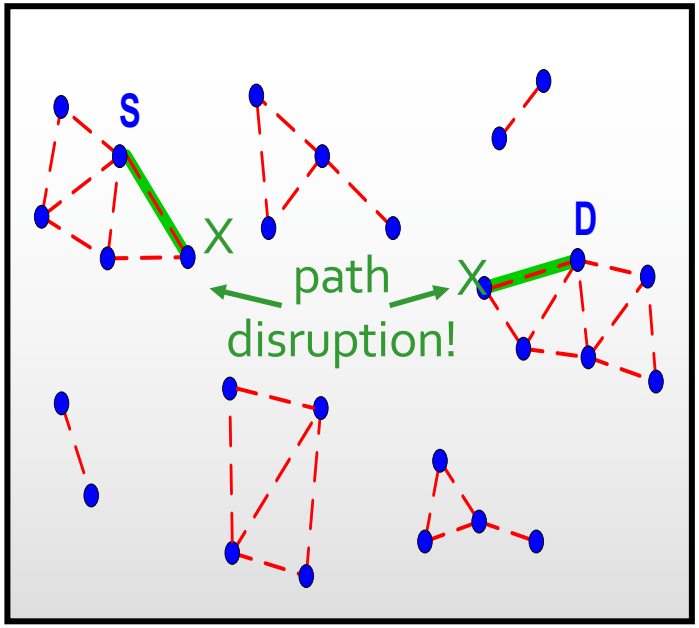
- Data bundles (i.e., messages) interchange is becoming everyday more common
  - E.g., Twitter, Whatsapp, Instagram, Snapchat, Facebook,…
- HTTP is becoming the new "transport" protocol
  - Request/response paradigm
  - Firewall friendly
  - Similar solutions: REST, CoAP
- Also… producer/consumer paradigm:
  - MQTT
  - AMQP, XMPP
- **Data Dissemination**

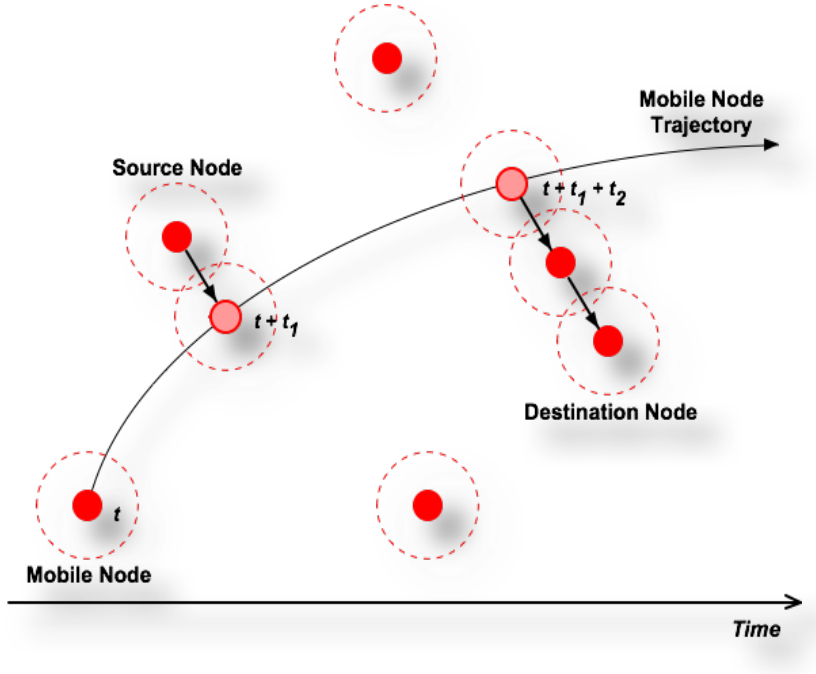




Opportunistic Data Dissemination

- Problem with partitioned networks
  - Next hop is not always present
- Opportunistic approach
  - Basic Idea: Store, Carry and Forward
  - Disruption Tolerant Networks



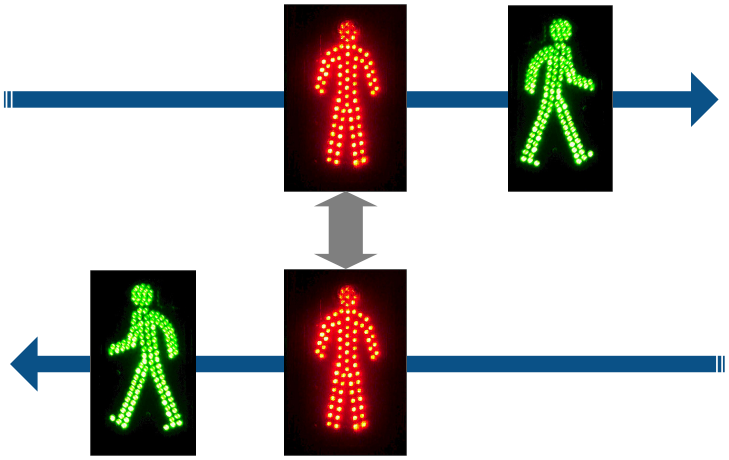


"Bundles"

- Bundle: Application-meaningful message
  - Contains all necessary info packed inside one bundle (atomic message)
  - Next hop has immediate knowledge of storage and bandwidth requirements
  - Remember SMTP+MIME?
- Optional ACKs
  - Depending on class service
- Goal: Avoid chattiness
  - Minimize number of propagation delays paid
- No more links!
  Now we have "contacts"

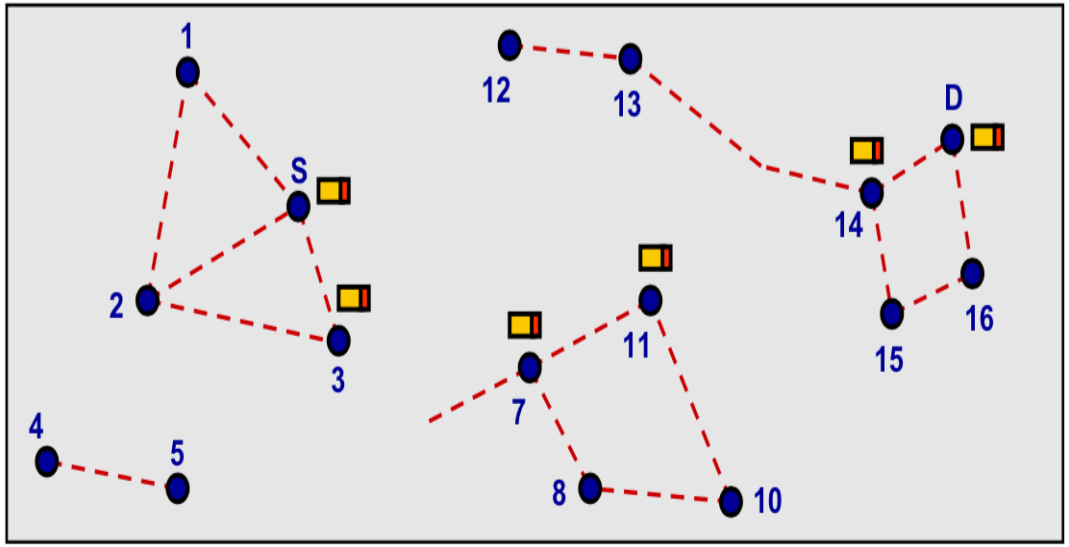
Encounters

- The number of “encounters” can be bigger than the number of “contacts”
- Not easy to efficiently transform an encounter into a contact
- Determine as fast as possible the “others”
- Infos availability through handshaking



Store-Carry-and-Forward

- Store a bundle for a long period of time.
- Forward when the next contact is available
  - Hours or even days until appropriate contact.
- How is this different from Internet routers store-and- forward?
  - **Persistent storage/local buffers: (hard disk, days) vs memory storage (few ms)**
  - Wait for next hop to appear vs. wait for table-lookup and available outgoing routing port



JSON and Python

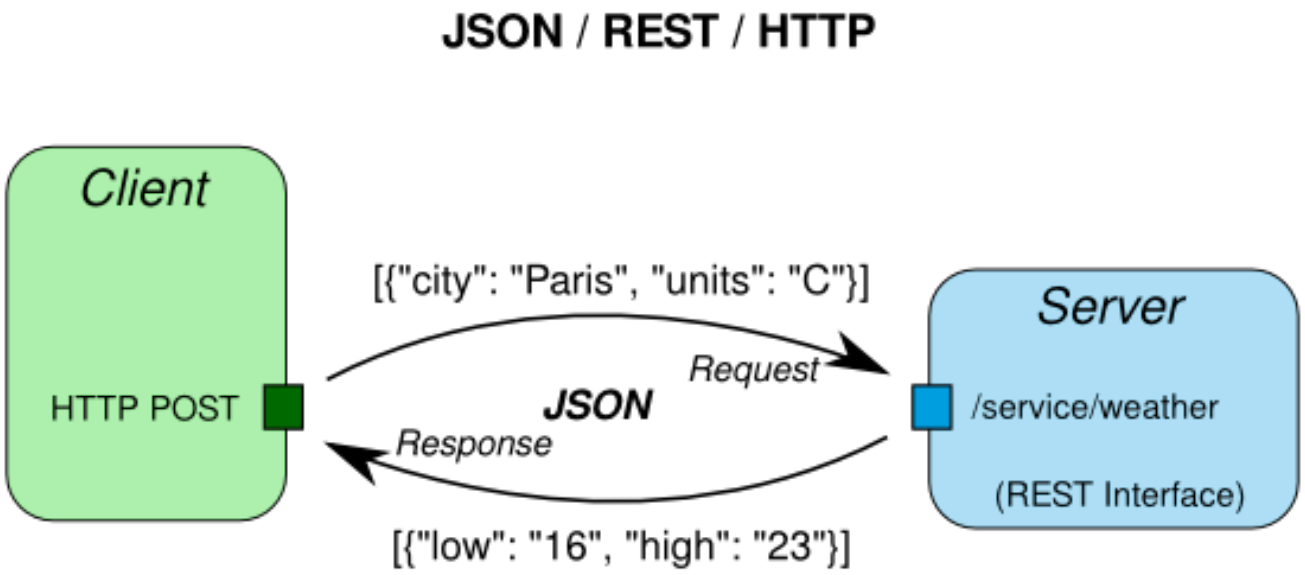
{"sensor": {"value": 237}}

```
<sensor>
        <value>237</value>
</sensor>
```

```
>>> import json

>>> jdata = json.loads('{"sensor": {"value": 237}}')
>>> jdata
{u'sensor': {u'value': 237}}
>>> jdata['sensor']
{u'value': 237}
>>> jdata['sensor']['value']= 666
>>> jdata
{u'sensor': {u'value': 666}}

>>> json.dumps(jdata)
'{"sensor": {"value": 666}}'

>>> jdata['sensor']= 'off'
>>> json.dumps(jdata)
'{"sensor": "off"}'
```

REST

- Basic concepts
- Basic programming



Data Representation

{“value": 237} vs. <value>237</value>

- Data-interchange format. Should be easy for humans to read and write. Should be easy for machines to parse and generate
- Two main formats:

JavaScript Object Notation (JSON) [http://www.json.org/]

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
      "menuitem": [
         {"value": "New", "onclick": ”NewDoc()"},
         {"value": "Open", "onclick": "OpenDoc()"},
         {"value": "Close", "onclick": "CloseDoc()"}
      ]
    }
  }
}
```

XML

```
<menu>
  <id>file</id>
  <value>File</value>
  <popup>
    <menuitem>
      <value>New</value>
      <onclick>NewDoc()</onclick>
    </menuitem>
    <menuitem>
      <value>Open</value>
      <onclick>OpenDoc()</onclick>
    </menuitem>
    <menuitem>
      <value>Close</value>
      <onclick>CloseDoc()</onclick>
    </menuitem>
  </popup>
</menu>
```

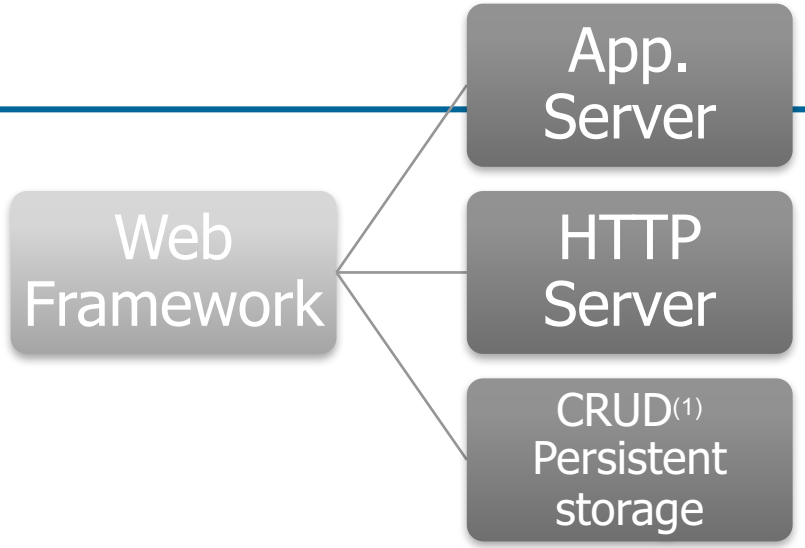
REST and HTTP

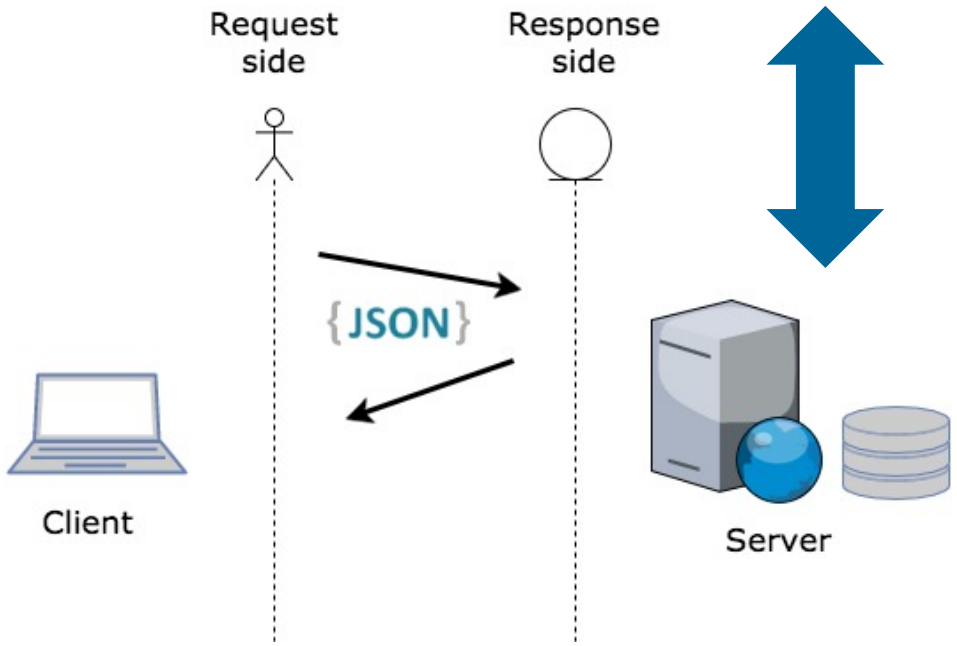
- REST stands for Representational State Transfer.
- It basically leverages the HTTP protocol and its related frameworks to provide data services.
- The motivation for REST was to capture the characteristics of the Web which made the Web successful.
- Make a Request – Receive Response – Display Response
- REST is not a standard… but it uses several standards:
  - HTTP
  - URL
  - XML/HTML/GIF/JPEG/etc (Resource Representations)
  - text/xml, text/html, image/gif, image/jpeg, etc (Resource Types, MIME Types)

Web basics

- The HTTP protocol is based on a request-response model.
  - The client (e.g., a browser) requests data from a web application that resides on a physical machine.
  - The web application in turn responds to the request with the data requested.
- Web application frameworks, or simply "web frameworks", are the de facto way to build web-enabled applications.

- Browsers download websites from **Wweb Servers** (Apache, Nginx, ...) using the HTTP protocol. They take care of :
  - inspect the requested URL and return the appropriate page
  - How to deal with POST requests in addition to simple GET requests
  - handle more advanced concepts like sessions and cookies
  - scale the application to handle thousands of concurrent connections





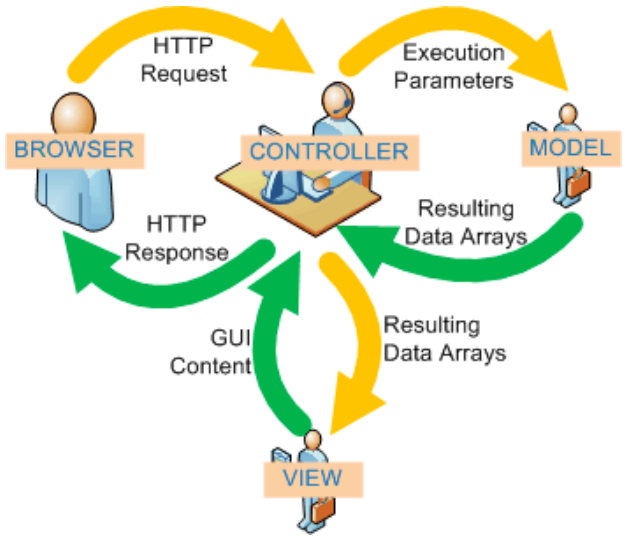
(1)CRUD: create, read, update, and delete

The simplest web application

```
import socket

HOST = ''
PORT = 8080
listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
listen_socket.bind((HOST, PORT))
listen_socket.listen(1)
connection, address = listen_socket.accept()
request = connection.recv(1024)
connection.sendall("""HTTP/1.1 200 OK
Content-type: text/html


<html>
    <body>
        <h1>Hello, World!</h1>
    </body>
</html>""")
connection.close()
```

Web applications: Routing and Templates

- Of all the issues surrounding building a web application, two stand out.
  1. How do we create the requested HTML dynamically, injecting calculated values or information retrieved from a database?
     - TEMPLETING
  2. How do we map a requested URL to the code that is meant to handle it?
     - ROUTING
- Every web framework solves these issues in some way, and there are many different approaches.



Routing

- **Routing is the process of mapping a requested URL to the code responsible for generating the associated HTML.**
  - In the simplest case, all requests are handled by the same code. Getting a little more complex, every URL could map 1:1 to a view function.
    - For example, we could record somewhere that if the URL www.foo.com/bar is requested, the function handle_bar() is responsible for generating the response.
- However, this approach fails when the URLs contain useful data, such as the ID of a resource (as is the case in www.foo.com/users/3/).
  - How do we map that URL to a view function, and at the same time make use of the fact that we want to display the user with ID 3?
- Every web framework offer a different solution
  - Bottle and Flask take a somewhat different approach. The canonical method for hooking up a function to a requested URL is through the use of the route() decorator.
  - Django maps URL to regular expressions

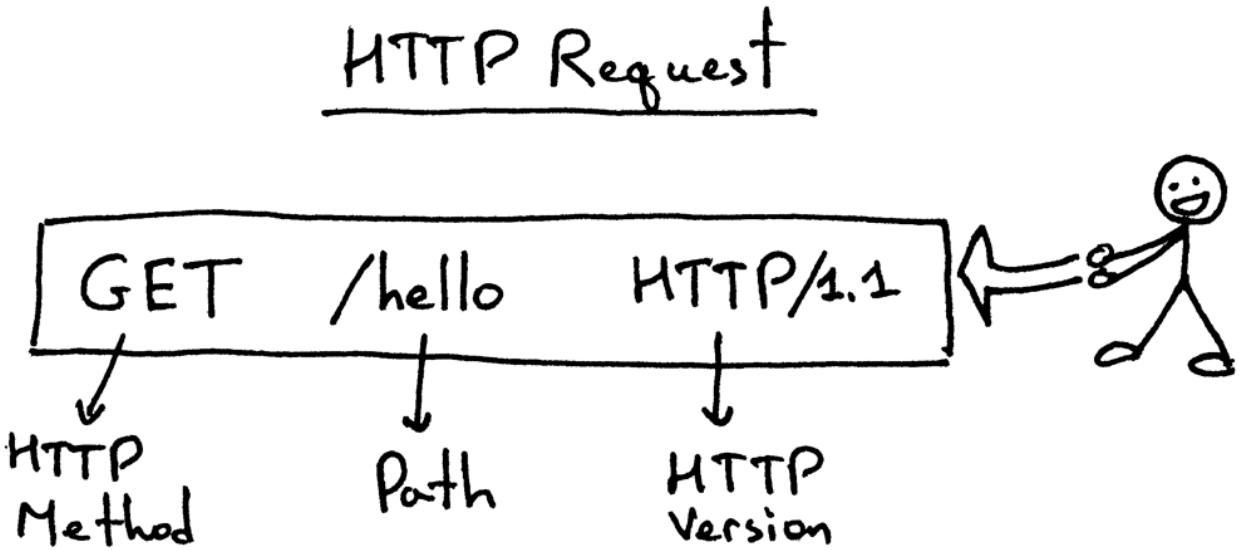
Back to REST: the resources

- The key abstraction of information in REST is a resource.
- A resource is a conceptual mapping to a set of entities
  - Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on
- Represented with a global identifier (URI in HTTP)
  - http://www.boeing.com/aircraft/747
  - http://www.library.edu/books/
  - http://www.library.edu/books/ISBN-0011
  - http://www.library.edu/books/ISBN-0011/authors
- As you traverse the path from more generic to more specific, you are navigating the data

Verbs

- Represent the actions to be performed on resources

- HTTP GET
- HTTP POST
- HTTP PUT
- HTTP DELETE



HTTP GET

- How clients ask for the information they seek.
- Issuing a GET request transfers the data from the server to the client in some representation (JSON, XML, …)

`GET http://www.library.edu/books`

- Retrieve all books

`GET http://www.library.edu/books/ISBN-0011021`

- Retrieve book identified with ISBN-0011021

`GET http://www.library.edu/books/ISBN-0011021/authors`

- Retrieve authors for book identified with ISBN-0011021

HTTP PUT, HTTP POST

- HTTP POST creates a resource
- HTTP PUT updates a resource

- POST http://www.library.edu/books/

```
Content: {title, authors[], …}
```

  - Creates a new book with given properties

- PUT http://www.library.edu/books/isbn-111

```
Content: {isbn, title, authors[], …}
```

  - Updates book identified by isbn-111 with submitted properties

HTTP PATCH

- PATCH is defined in RFC 5789. It requests that **a set of changes** described in the request entity be applied to the resource identified by the Request- URI. Let's look at one example.

```
{ "username": "skwee357", "email": "skwee357@domain.com" }
```

- If you POST this document to /users, then you get back an entity such as:

```
{
    "username": "skwee357",
    "email": "skwee357@domain.com"
}
```

- If you want to modify this entity later, you choose between PUT and PATCH. A PUT might look like this:

```
PUT /users/1
{
    "username": "skwee357",
    "email": "skwee357@gmail.com"       // new email address
}
```

- You can accomplish the same using PATCH:

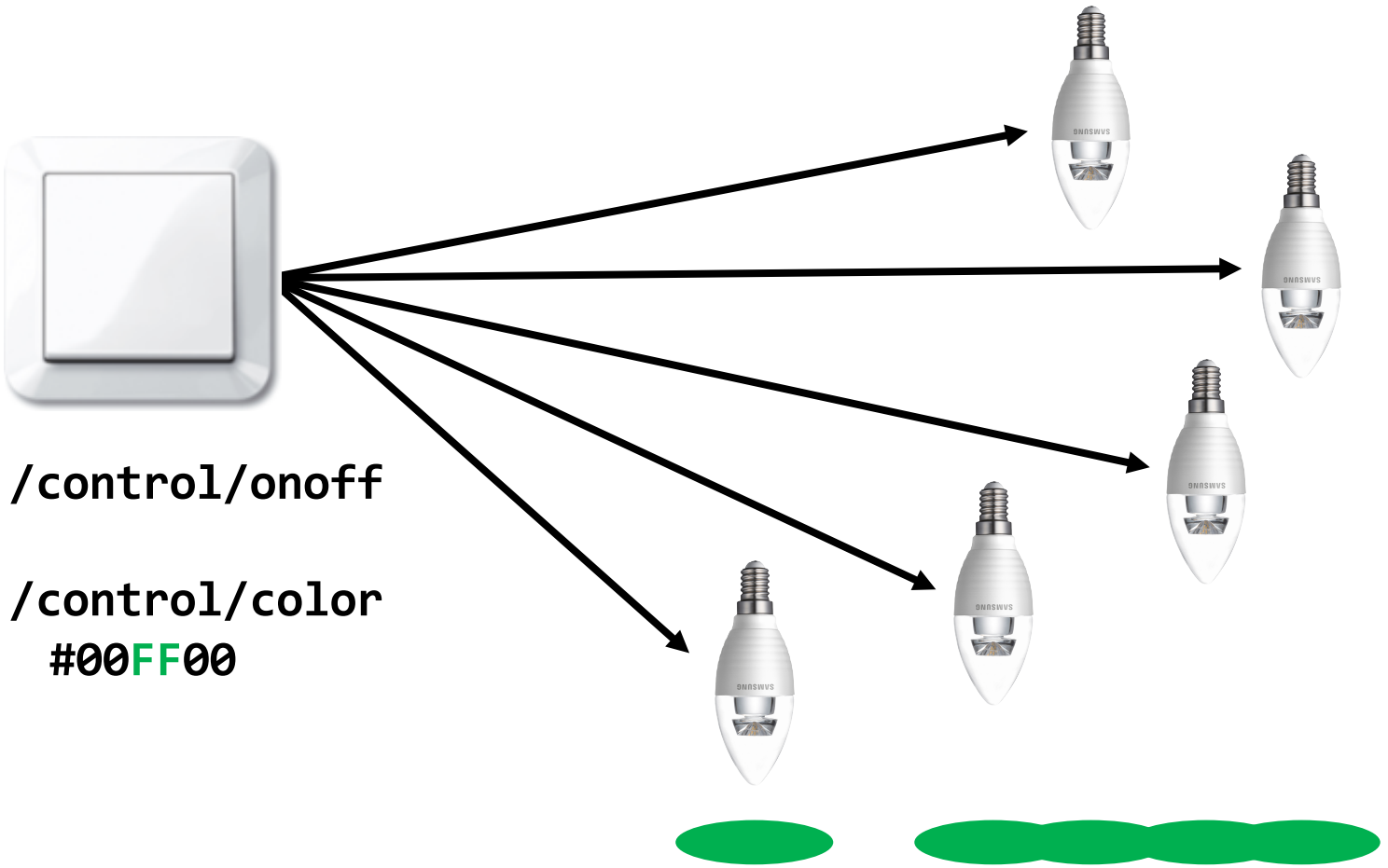
```
PATCH /users/1
{
    "email": "skwee357@gmail.com"       // new email address
}
```

- The PUT included all of the parameters on this user, but PATCH only included the one that was being modified (email).

REST for devices control communication

```
GET /status/power
```

```
all-lights.floor-d.example.com
```



```
PUT /control/onoff
```

```
PUT /control/color
      #00FF00
```

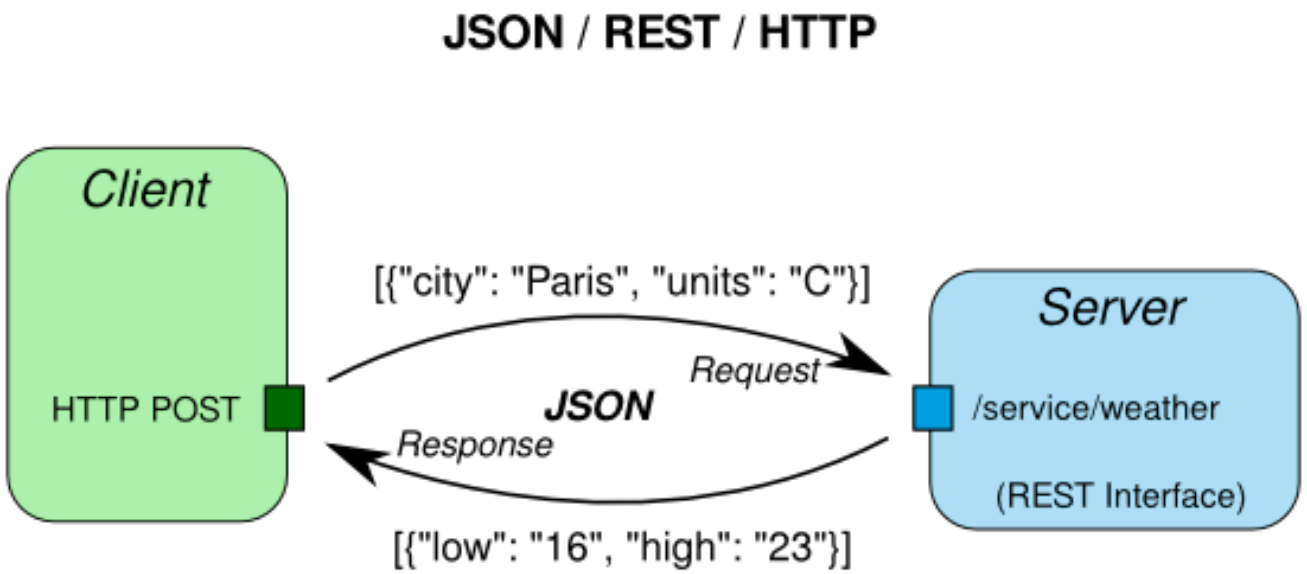
HTTP DELETE

- Removes the resource identified by the URI

- DELETE `http://www.library.edu/books/ISBN-0011`
  - Delete book identified by ISBN-0011

REST

- Basic concepts
- Basic programming



Talking about standards: the Java case

- JSR 311: JAX-RS: The Java API for RESTful Web Services
  - https://jcp.org/en/jsr/detail?id=311
- It's a Java programming language API spec that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern.
- JAX-RS uses annotations (see later "decorators" with Python), introduced in Java SE 5, to simplify the development and deployment of web service clients and endpoints.
- From version 1.1 on, JAX-RS is an official part of Java EE 6. A notable feature of being an official part of Java EE is that no configuration is necessary to start using JAX-RS.

Java REST

- Restlet:

  http://www.restlet.org/
- JBoss RESTeasy:

  http://resteasy.jboss.org/
- Spring by Pivotal

  https://spring.io/
- Jersey

  https://jersey.java.net/
- Apache Wink

  http://wink.apache.org/
- Jello

  http://jello-framework.com/

Let's talk Python

- big web frameworks
  - Django: http://www.django-rest-framework.org/
  - Pyramid: https://trypyramid.com/
  - Falcon: http://falconframework.org/
- minimalist web frameworks
  - **Bottle: http://bottlepy.org/docs/dev/**
  - Flask: http://flask.pocoo.org/
  - Pycnic: http://pycnic.nullism.com/

WSGI (Web Server Gateway Interface)

- It's a Python specification (standards) that describes how a web server communicates with web applications, and how web applications can be chained together to process one request.
  - http://wsgi.readthedocs.io/en/latest/
  - WSGI is not a server, a python module, a framework, an API or any kind of software. It is just an interface specification by which server and application communicate. Both server and application interface sides are specified in the PEP 3333.
  - If an application (or framework or toolkit) is written to the WSGI spec then it will run on any server written to that spec.
- A WSGI server (meaning WSGI compliant) only receives the request from the client, pass it to the application and then send the response returned by the application to the client. It does nothing else. All the gory details must be supplied by the application or middleware.
- Frameworks that run on WSGI:
  - http://wsgi.readthedocs.io/en/latest/frameworks.html
  - https://wiki.python.org/moin/WebFrameworks
- WSGI-compliant servers
  - uWSGI, Tornado, Gunicorn, Apache, Amazon Beanstalk, Google App Engine, and others.

Bottle: Python Web Framework

- The entire library is distributed as a one-file module
- The built-in default server is based on WSGIServer.
  - This non-threading HTTP server is perfectly fine for development and early production, but may become a performance bottleneck when server load increases.
  - There are three ways to eliminate this bottleneck: http://bottlepy.org/docs/0.12/deployment.html
- Bottle is database-agnostic and doesn't care where the data is coming from.
  - If you'd like to use a database in your app, the Python Package Index has several interesting options, like SQLAlchemy, PyMongo, MongoEngine, CouchDB…

- `sudo apt-get install python-bottle`

The server structure

```
import bottle
from bottle import request, response, route
from bottle import post, get, put, delete
import json
import time

_names = set()   # the set of sensors names

@post('/sensors')
def creation_handler():
  ...

@get('/sensors')
def listing_handler():
 ...

@get('/sensors/<sensor_name>')
def listing_with_name_handler(sensor_name):
 ...

@put('/sensors/<sensor_name>')
def update_handler(sensor_name):
 ...

@delete('/sensors/<sensor_name>')
def delete_handler(sensor_name):
 ...

@route('/')
def hello():
    return 'Hello World'

if __name__ == '__main__':
    bottle.run(host = '127.0.0.1', port = 8000)
```

The server side: POST

```
@post('/sensors')
def creation_handler():
    try:
        # parse input data
        try:
            data = json.loads(request.body.read())
        except:
            raise ValueError

        # extract the sensor name
        try:
            sensor_name = data['name']
        except (TypeError, KeyError):
            raise ValueError

        # check for the existence of the sensor name
        if len( _names ) > 0:
            if [ x for x in _names if x[0] == sensor_name]:
                raise KeyError

    except ValueError:
        response.status = 400
        return "Bad Request - input data with errors in POST"

    except KeyError:
        response.status = 409
        return "Bad Request - sensor name already exist in POST"

    # add element with default value set to 0 and current time
    new_sensor_t = (sensor_name, 0, time.ctime())
    _names.add( new_sensor_t )

    # return 200 Success
    response.status = 200
    response.headers['Content-Type'] = 'application/json'
    return json.dumps(new_sensor_t)
```

The server side: GET

```
@get('/sensors')
def listing_handler():
    response.headers['Content-Type'] = 'application/json'
    return json.dumps(list(_names))

@get('/sensors/<sensor_name>')
def listing_with_name_handler(sensor_name):
    response.headers['Content-Type'] = 'application/json'

    try:
        # check for the existence of sensor with name "sensor_name"
        the_sensor = [ x for x in _names if x[0] == sensor_name]
        if the_sensor:
            return json.dumps(the_sensor)
        else:
            raise KeyError

    except KeyError:
        response.status = 409
        return "Bad Request - sensor name does not exist in GET"
```

The server side: PUT

```
@put('/sensors/<sensor_name>')
def update_handler(sensor_name):
    try:
        # parse input data
        try:
            data = json.loads(request.body.read())
        except:
            raise ValueError
        # extract and validate new value
        try:
            if not data["value"].isdigit():
                raise ValueError
            newdata = int(data["value"])

        except (TypeError, KeyError):
            raise ValueError

        # check for the existence of sensor with name "sensor_name"
        the_sensor = [ x for x in _names if x[0] == sensor_name]
        if not the_sensor:
            raise KeyError



    except ValueError:
        response.status = 400
        return
    except KeyError:
        response.status = 409
        return "Bad Request - sensor name does not exist in GET”
```

```
# add new name and remove old name
 _names.remove(the_sensor[0])
 newdata_t = (sensor_name, newdata, time.ctime())
 _names.add(newdata_t)

 # return 200 Success
 response.headers['Content-Type'] = 'application/json'
 return json.dumps(newdata_t)
```

The server side: DELETE

```
@delete('/sensors/<sensor_name>')
def delete_handler(sensor_name):
    try:
        # check for the existence of sensor with name "sensor_name"
        the_sensor = [ x for x in _names if x[0] == sensor_name]
        if not the_sensor:
            raise KeyError

    except KeyError:
        response.status = 409
        return "Bad Request - sensor name does not exist in DELETE"

    # Remove name
    _names.remove(the_sensor[0])
    return
```

Client side tools

- Postman:

  https://www.getpostman.com/

- Advanced REST Client:

  https://advancedrestclient.com/

- Curl:

  https://curl.haxx.se/

  - Examples:

```
curl -X POST -H "Content-type: application/json" -d '{"name":"sensor1"}'\
      localhost:8000/sensors
curl -X POST -H "Content-type: application/json" -d '{"name":"sensor2"}'\
      localhost:8000/sensors

curl -X GET localhost:8000/sensors
curl -X GET localhost:8000/sensors/sensor2

curl -X PUT -H "Content-type: application/json" -d '{"value":"237"}'\
      localhost:8000/sensors/sensor2

curl -X DELETE localhost:8000/sensors/sensor2
```

Client side via Python

- https://pypi.python.org/pypi/requests
- http://docs.python-requests.org/en/master/
- Cite: "*Requests is the only Non-GMO HTTP library for Python, safe for human consumption.*" ☺

Periodic GET

```
import requests
import time

rest_s_url = 'http://localhost:8000'

while True:
    try:
        resp = requests.get( rest_s_url+'/sensors')
        if resp.status_code != 200:
            # This means something went wrong.
            raise ValueError

        for sdata in resp.json():
            print sdata[0]+' has value '+str(sdata[1])+' at '+sdata[2]

        time.sleep(3)

    except ValueError:
        print "Bad reply from GET"
```

Periodic PUT

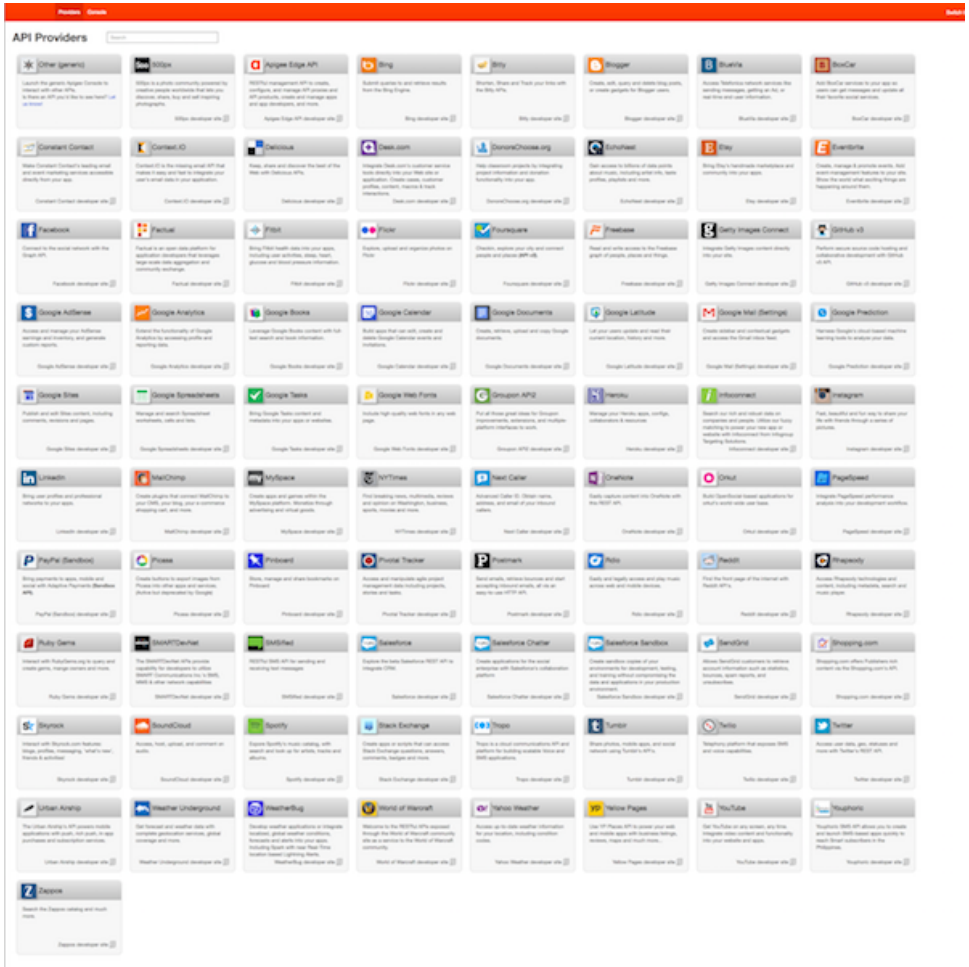
```
import requests, time, random, json

rest_s_url = 'http://localhost:8000/'
sensor_name = 'sensor1'

while True:
    try:

        # get the new data value
        new_s_value = random.randint(1, 1000)
        new_sensor_t = {'value': str(new_s_value)}

        resp = requests.put(rest_s_url+'sensors/'+sensor_name,\
            data=json.dumps(new_sensor_t),\
            headers={'Content-Type':'application/json'})

        if resp.status_code != 200:
            # This means something went wrong.
            raise ValueError

        time.sleep(3)

    except ValueError:
        print "Bad reply from PUT"
```

REST is widely used

- Google Maps:
  - https://developers.google.com/maps/web-services/
  - Try: `http://maps.googleapis.com/maps/api/geocode/json?address=lecco`
- Twitter:
  - https://dev.twitter.com/rest/public
- Facebook:
  - https://developers.facebook.com/docs/atlas-apis
- Amazon offers several REST services, e.g., for their S3 storage solution
  - http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html
- Also:
  - The Google Glass API, known as "Mirror API", is a pure REST API.
    - Here is (https://youtu.be/JpWmGX55a40) a video talk about this API. (The actual API discussion starts after 16 minutes or so.)
  - Tesla Model S uses an (undocumented) REST API between the car systems and its Android/iOS apps.
    - http://docs.timdorr.apiary.io/#reference/vehicles/state-and-settings

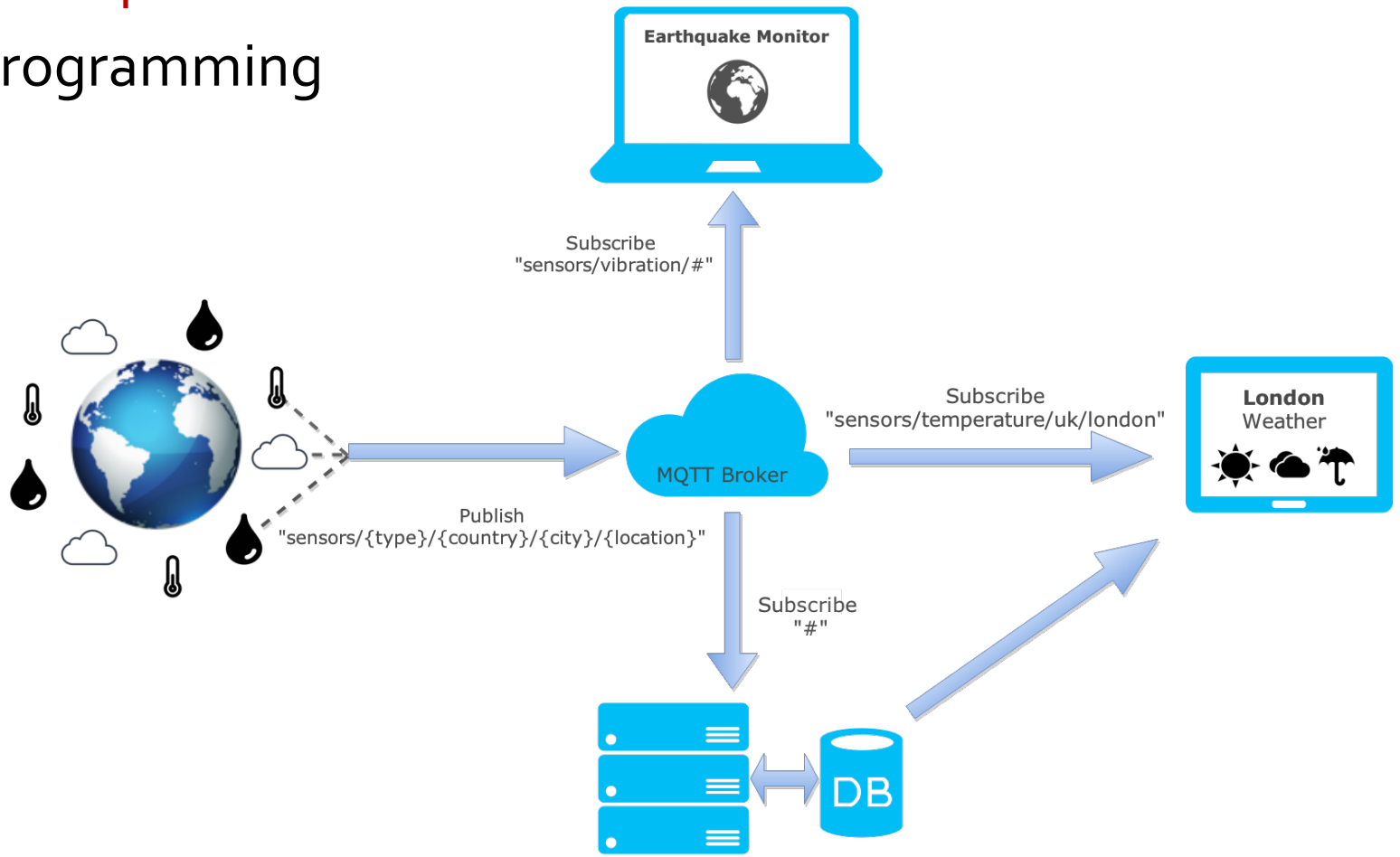
Very widely…

- https://apigee.com/providers
  - https://apigee.com/resources/instagram



MQTT

- Basic concepts
- Basic programming



Source: https://zoetrope.io/tech-blog/brief-practical-introduction-mqtt-protocol-and-its-application-iot

Message Queuing Telemetry Transport

MQTT

- A lightweight publish-subscribe protocol that runs on embedded devices and mobile platforms designed to connect the physical world devices with applications and middleware.
  - http://mqtt.org/
  - the MQTT community wiki: https://github.com/mqtt/mqtt.github.io/wiki
  - http://www.hivemq.com/mqtt-essentials/
- Designed to provide a low latency two-way communication channel and efficient distribution to one or many receivers.
- **Minimizes the amount of bytes flowing over the wire**
- **Low power usage.**

Message Queuing Telemetry Transport

MQTT

MQTT

designed for minimal network traffic
and constrained devices

small header size

PUBLISH	2-4 bytes
CONNECT	14 bytes
HTTP	0.1-1 KB

binary payload (not text)

small clients: 30 KB (C), 100 KB (Java)

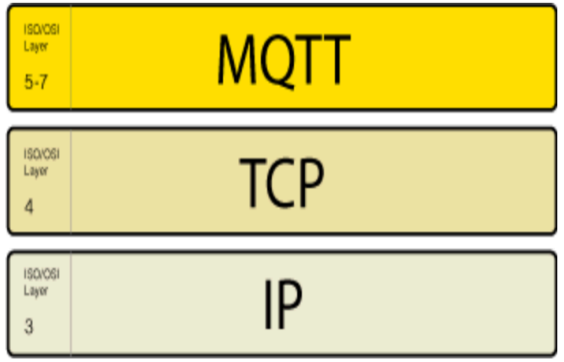
minimal protocol exchanges

MQTT has configurable keep alive
(2 byte PINGREQ / PINGRES)

efficient for battery life: http://stephendnicholas.com/archives/1217

MQTT overview

- MQTT is an open standard with a short and readable protocol specification.
  - `http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html`
- Works on top of the TCP protocol stack
- There is also the closely related MQTT for Sensor Networks (MQTT-SN)
  - TCP is replaced by UDP: to transfer small data pieces (measurements, remote commands, or user data) TCP stack is too complex for WSN
  - This has led to multiple, interoperable implementations.



- October 29th 2014: MQTT was officially approved as OASIS Standard.
- MQTT 3.1.1 is the current version of the protocol.
  - OASIS MQTT TC: `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt`

MQTT Version 5

- The next version of MQTT (autumn 2017) will be version 5.
  - If you are wondering what happened to 4 then see:

http://www.eclipse.org/community/eclipse_newsletter/2016/september/article3.php

- You can find a working draft:

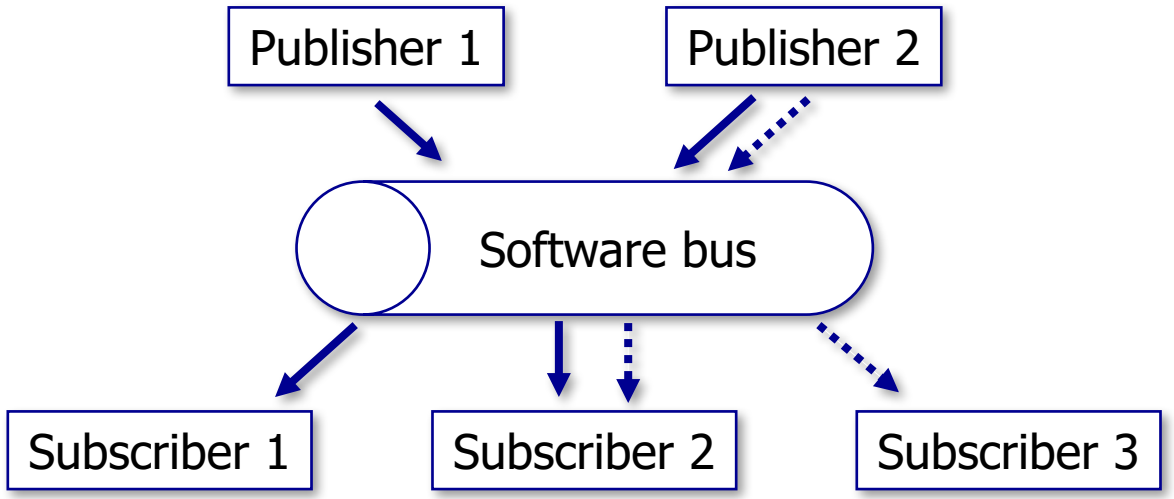
https://www.oasis-open.org/committees/download.php/59482/mqtt-v5.0-wd09.pdf

- And the key ideas in version 5 here.

https://www.oasis-open.org/committees/download.php/57616/Big%20Ideas%20for%20MQTT%20v5.pdf

Publish/subscribe pattern

- Pub/Sub decouples a client, who is sending a particular message (called publisher) from another client (or more clients), who is receiving the message (called subscriber). This means that the publisher and subscriber don't know about the existence of one another.
- There is a third component, called broker, which is known by both the publisher and subscriber, which filters all incoming messages and distributes them accordingly.



MQTT Connection

MQTT-Packet:

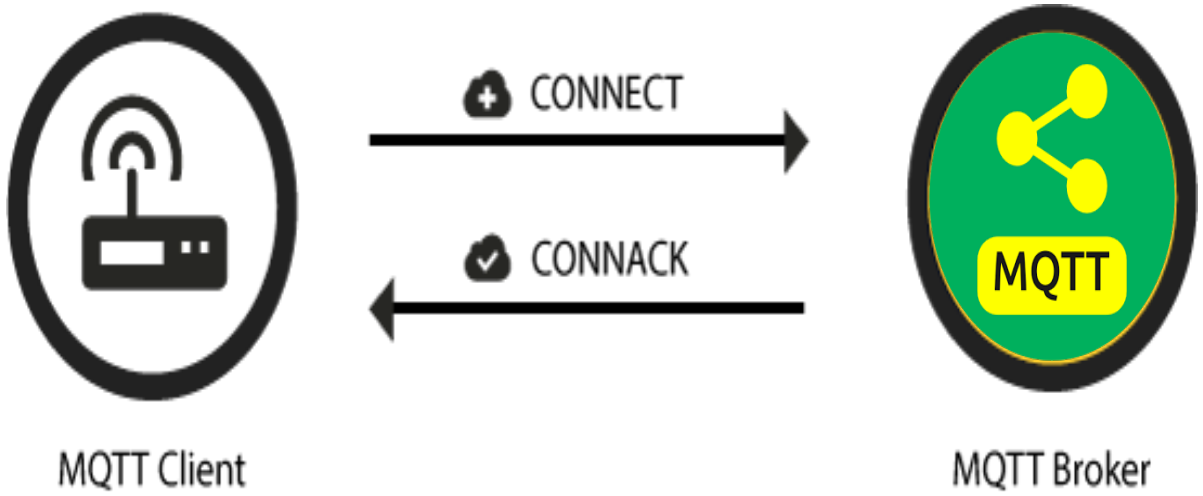
CONNECT

contains:	Example
clientId	"client-1"
cleanSession	true
username (optional)	"hans"
password (optional)	"letmein"
lastWillTopic (optional)	"/hans/will"
lastWillQos (optional)	2
lastWillMessage (optional)	"unexpected exit"
lastWillRetain (optional)	false
keepAlive	60

MQTT-Packet:

CONNACK

contains:	Example
sessionPresent	true
returnCode	0



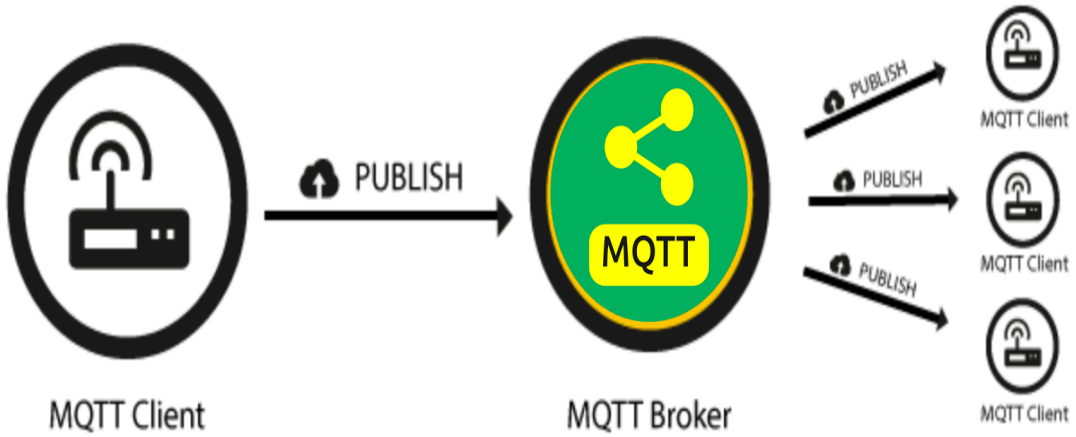
Publish

- MQTT is data-agnostic and it totally depends on the use case how the payload is structured. It's completely up to the sender if it wants to send binary data, textual data or even full-fledged XML or JSON.

MQTT-Packet:

PUBLISH

contains:	Example
packetId (always 0 for qos 0)	4314
topicName	"topic/1"
qos	1
retainFlag	false
payload	"temperature:32.5"
dupFlag	false

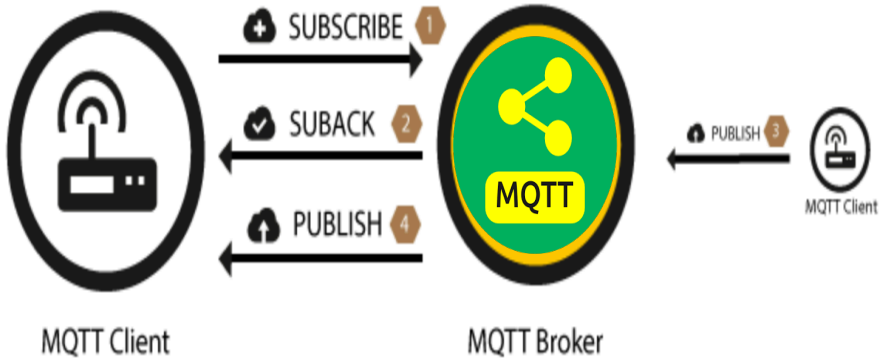


Subscribe

MQTT-Packet:

SUBSCRIBE

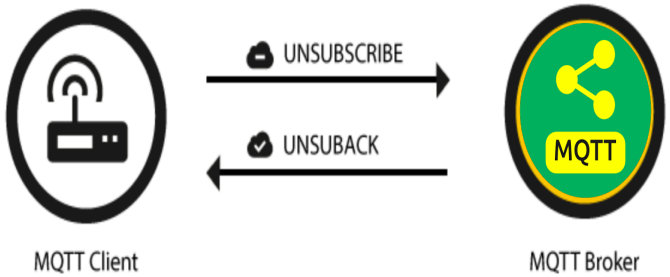
contains:		Example
packetId		4312
qos1	} (list of topic + qos)	1
topic1		"topic/1"
qos2	}	0
topic2		"topic/2"
...		...



MQTT-Packet:

SUBACK

contains:		Example
packetId		4313
returnCode 1	(one returnCode for each	2
returnCode 2	topic from SUBSCRIBE,	0
...	in the same order)	...



Topics

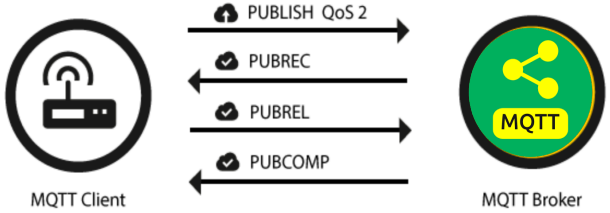
- Topic subscriptions can have wildcards
  - '+' matches anything at a given tree level so the topic "sensor/+/temp" would match "sensor/dev1/temp", "sensor/dev2/temp", etc.
  - '#' matches a whole sub-tree, so "sensor/#" would match all topics under "sensor/".
- These enable nodes to subscribe to groups of topics that don't exist yet, allowing greater flexibility in the network's messaging structure.

Topics best practices

- Don’t use a leading forward slash
- Don’t use spaces in a topic
- Keep the topic short and concise
- Use only ASCII characters, avoid non printable characters
- Embed a unique identifier or the ClientId into the topic
- Don’t subscribe to #
- Don’t forget extensibility
- Use specific topics, instead of general ones

Quality of Service (QoS)

- Messages are published with a Quality of Service (QoS) level, which specifies delivery requirements.
- A QoS-0 ("at most once") message is fire-and-forget.
  - For example, a notification from a doorbell may only matter when immediately delivered.
- With QoS-1 ("at least once"), the broker stores messages on disk and retries until clients have acknowledged their delivery.
  - (Possibly with duplicates.) It's usually worth ensuring error messages are delivered, even with a delay.
- QoS-2 ("exactly once") messages have a second acknowledgement round-trip, to ensure that non-idempotent messages can be delivered exactly once.



QoS: good to know

- Downgrade of QoS
  - The QoS flows between a publishing and subscribing client are two different things as well as the QoS can be different. That means the QoS level can be different from client A, who publishes a message, and client B, who receives the published message.
  - Between the sender and the broker the QoS is defined by the sender.
  - If client B has subscribed to the broker with QoS 1 and client A sends a QoS 2 message, it will be received by client B with QoS 1. And of course it could be delivered more than once to client B, because QoS 1 only guarantees to deliver the message at least once.
- Packet identifiers are unique per client
  - Also important to know is that each packet identifier (used for QoS 1 and QoS 2) is unique between one client and a broker and not between all clients.
  - If a flow is completed the same packet identifier can be reused anytime. That's also the reason why the packet identifier doesn't need to be bigger than 65535, because it is unrealistic that a client sends a such large number of message, without completing the flow.

QoS: Best Practice

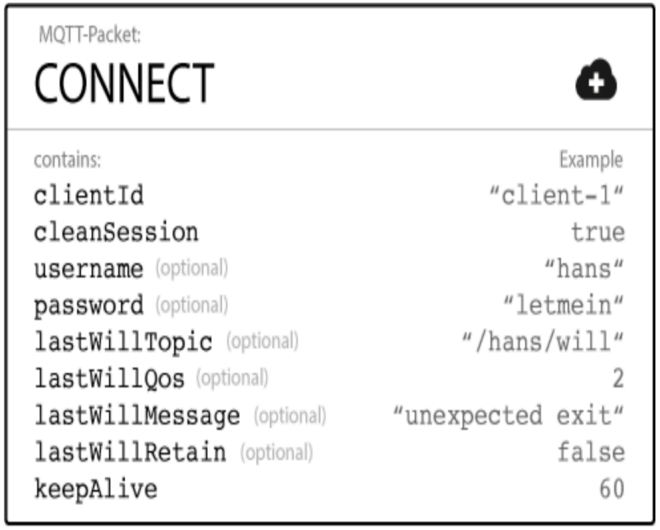
- Use QoS 0 when …
  - You have a complete or almost stable connection between sender and receiver. A classic use case is when connecting a test client or a front end application to a MQTT broker over a wired connection.
  - You don't care if one or more messages are lost once a while. That is sometimes the case if the data is not that important or will be send at short intervals, where it is okay that messages might get lost.
  - You don't need any message queuing. Messages are only queued for disconnected clients if they have QoS 1 or 2 and a persistent session.
- Use QoS 1 when …
  - You need to get every message and your use case can handle duplicates. The most often used QoS is level 1, because it guarantees the message arrives at least once. Of course your application must be tolerating duplicates and process them accordingly.
  - You can't bear the overhead of QoS 2. Of course QoS 1 is a lot fast in delivering messages without the guarantee of level 2.
- Use QoS 2 when …
  - It is critical to your application to receive all messages exactly once. This is often the case if a duplicate delivery would do harm to application users or subscribing clients. You should be aware of the overhead and that it takes a bit longer to complete the QoS 2 flow.
- Queuing of QoS 1 and 2 messages
  - All messages sent with QoS 1 and 2 will also be queued for offline clients, until they are available again. But queuing is only happening, if the client has a persistent session.

Retained Messages

- A retained message is a normal MQTT message with the retained flag set to true. **The broker will store the last retained message and the corresponding QoS for that topic**
  - Each client that subscribes to a topic pattern, which matches the topic of the retained message, will receive the message immediately after subscribing.
    - For each topic only one retained message will be stored by the broker.
- Retained messages can help newly subscribed clients to get a status update immediately after subscribing to a topic and don't have to wait until a publishing clients send the next update.
- The subscribing client doesn't have to match the exact topic, it will also receive a retained message if it subscribes to a topic pattern including wildcards.
  - For example client A publishes a retained message to myhome/livingroom/temperature and client B subscribes to myhome/# later on, client B will receive this retained message directly after subscribing.
  - In other words a retained message on a topic **is the last known good value**, because it doesn't have to be the last value, but it certainly is the last message with the retained flag set to true.
- It is important to understand that a retained message has nothing to do with a persistent session of any client.

"Will" message

- When clients connect, they can specify an optional "will" message, to be delivered if they are unexpectedly disconnected from the network.
  - (In the absence of other activity, a 2-byte ping message is sent to clients at a configurable interval.)
- This "last will and testament" can be used to notify other parts of the system that a node has gone down.



MQTT Keep alive

- The keep alive functionality assures that the connection is still open and both broker and client are connected to one another. Therefore the client specifies a time interval in seconds and communicates it to the broker during the establishment of the connection. The interval is the longest possible period of time, which broker and client can endure without sending a message.
  - Problem of half-open TCP connections
- Good to Know
  - If the broker doesn't receive a PINGREQ or any other packet from a particular client, it will close the connection and send out the last will and testament message (if the client had specified one).
  - The MQTT client is responsible of setting the right keep alive value. For example, it can adapt the interval to its current signal strength.
  - The maximum keep alive is 18h 12min 15 sec.
  - If the keep alive interval is set to 0, the keep alive mechanism is deactivated.

Persistent session

- A persistent session saves all information relevant for the client on the broker. The session is identified by the clientId provided by the client on connection establishment
- So what will be stored in the session?
  - Existence of a session, even if there are no subscriptions
  - All subscriptions
  - All messages in a Quality of Service (QoS) 1 or 2 flow, which are not confirmed by the client
  - All new QoS 1 or 2 messages, which the client missed while it was offlne
  - All received QoS 2 messages, which are not yet confirmed to the client
  - That means even if the client is offline all the above will be stored by the broker and are available right after the client reconnects.
- Persistent session on the client side
  - Similar to the broker, each MQTT client must store a persistent session too. So when a client requests the server to hold session data, it also has the responsibility to hold some information by itself:
  - All messages in a QoS 1 or 2 flow, which are not confirmed by the broker
  - All received QoS 2 messages, which are not yet confirmed to the broker

Common Questions

- Q- What happens to messages that get published to topics that no one subscribes to?
- A- They are discarded by the broker.

- Q-How can I find out what topics have been published?
- A- You can't do this easily as the broker doesn't seem to keep a list of published topics as they aren't permanent.

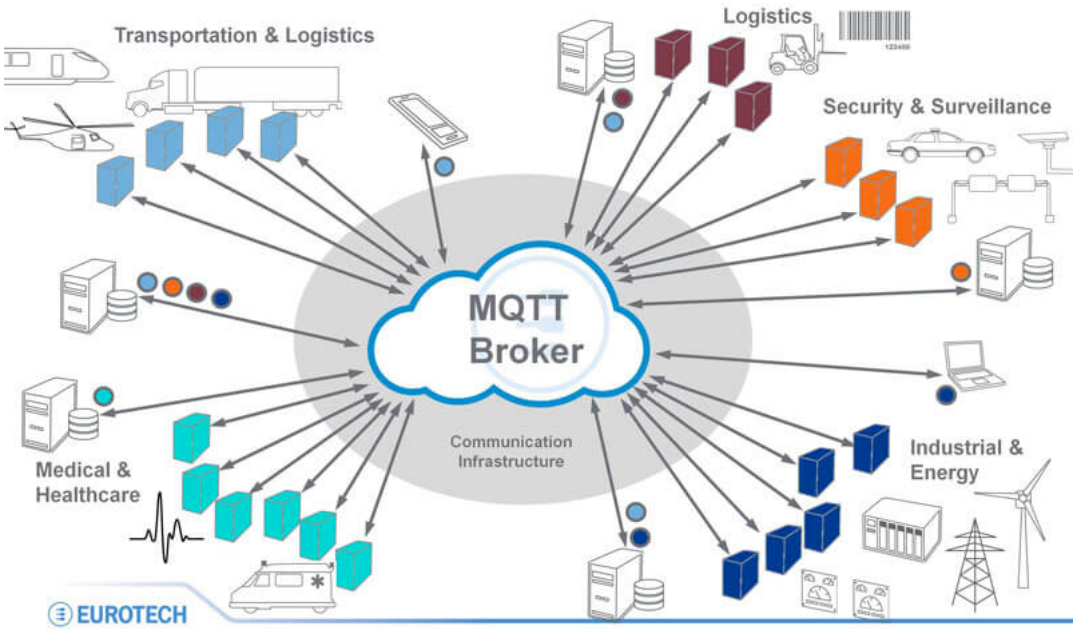
- Q- Can I subscribe to a topic that no one is publishing to?
- A- Yes

- Q- Are messages stored on the broker?
- A- Yes but only temporarily. Once they have been sent to all subscribers they are then discarded. But see next question.

- Q- What are retained messages?
- A- When you publish a message you can have the broker store the last published message. This message will be the first message that new subscribers see when they subscribe to that topic. MQTT only retains 1 message.

MQTT

- Basic concepts
- Basic programming



Software available

- Brokers (https://github.com/mqtt/mqtt.github.io/wiki/servers):
  - **http://mosquitto.org/**
  - http://www.hivemq.com/
  - https://www.rabbitmq.com/mqtt.html
  - http://activemq.apache.org/mqtt.html
  - https://github.com/mcollina/mosca
- Clients
  - **http://www.eclipse.org/paho/**
    - Source: https://github.com/eclipse/paho.mqtt.python
  - http://www.hivemq.com/demos/websocket-client/
- Tools
  - https://github.com/mqtt/mqtt.github.io/wiki/tools

The suggested working environment

- Installing the Python client


```
sudo pip install paho-mqtt
```

  - or

```
git clone https://github.com/eclipse/paho.mqtt.python.git
cd org.eclipse.paho.mqtt.python.git
python setup.py install
```

- Installing the whole Mosquitto environment


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

sudo wget http://repo.mosquitto.org/debian/mosquitto-jessie.list
sudo apt-get update sudo apt-get install mosquitto

sudo apt-get install mosquitto mosquitto-clients python-mosquitto
```

Controlling the broker

- Starting / stopping the broker

```
sudo /etc/init.d/mosquitto stop
sudo /etc/init.d/mosquitto start
```

- To avoid that it restarts automatically

```
sudo stop mosquitto
```

- Running verbose mode:

```
sudo mosquitto -v
```

- To check whether is running:

```
sudo netstat -tanlp | grep 1883
ps -ef | grep mosquitto
```

- man page:

```
https://mosquitto.org/man/mosquitto-8.html
```

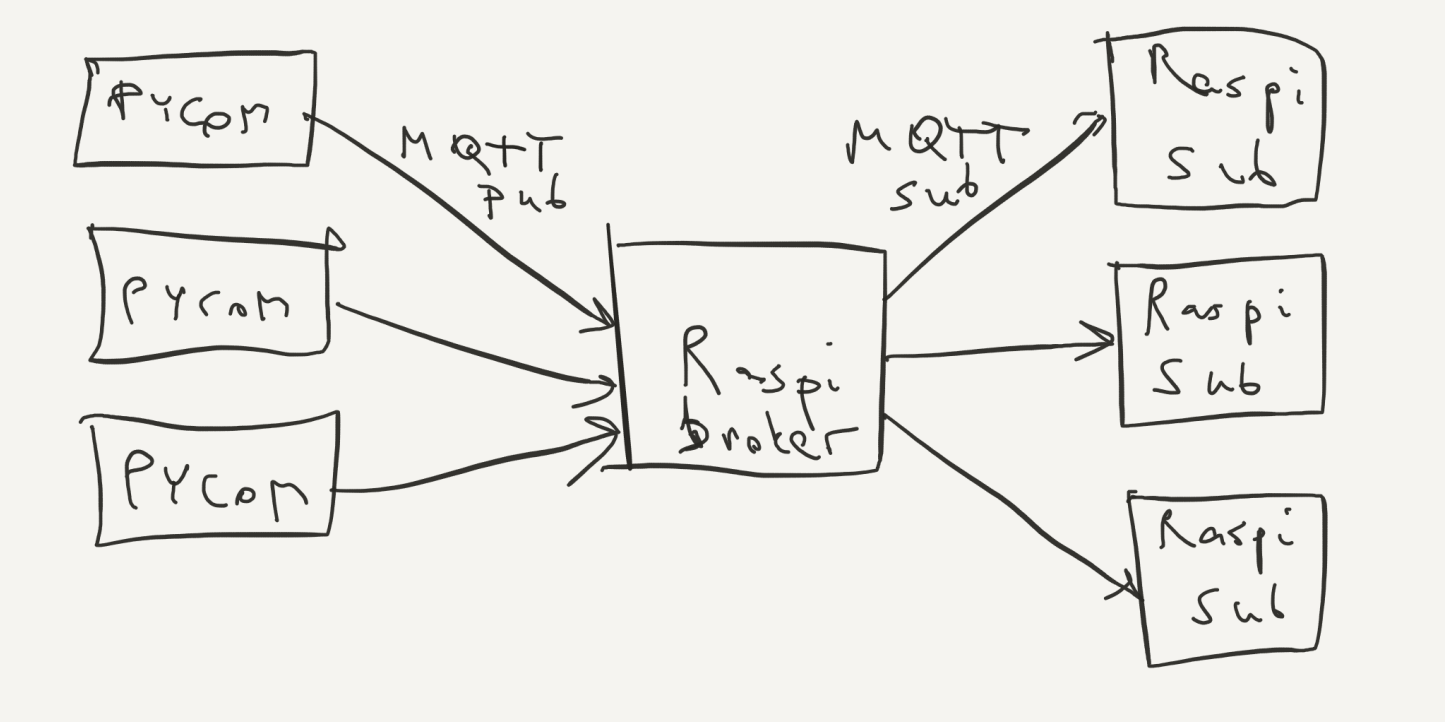
Sandboxes

- https://iot.eclipse.org/getting-started#sandboxes
  - hostname iot.eclipse.org and port 1883.
  - encrypted port 8883
  - http://test.mosquitto.org/
  - http://www.hivemq.com/try-out/
  - http://www.mqtt-dashboard.com/

Server	Broker	Port	Websocket
iot.eclipse.org	Mosquitto	1883 / 8883	n/a
broker.hivemq.com	HiveMQ	1883	8000
test.mosquitto.org	Mosquitto	1883 / 8883 / 8884	8080 / 8081
test.mosca.io	mosca	1883	80
broker.mqttdashboard.com	HiveMQ	1883	

- Google CLOUD PUB/SUB A global service for real-time and reliable messaging and streaming data
  - https://cloud.google.com/pubsub/

Excercise


PyCom
PyCom
PyCom
MQTT
Pub
Raspi
Broker
MQTT
Sub
Raspi
Sub
Raspi
Sub
Raspi
Sub

Excercise

- Broker information:

- WiFi SSID: mqtt
- WiFi password: brokerpi
- Broker IP address: 172.24.1.1

Excercise

1. Start with the MQTT example. It will send a text message via MQTT to the broker. Does it show up?

2. Next develop a MQTT + Pysense example. It will read sensor values from Pysense and send them to the broker. Can you read the values in the broker?

3. Install a MQTT client on your PC/smartphone and subscribe to the topics of interest.

Example 1: subscriber with `loop_forever`

```
import paho.mqtt.client as mqtt

# The callback for when the client receives a CONNACK response from the server.
def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))

    # Subscribing in on_connect() means that if we lose the connection and
    # reconnect then subscriptions will be renewed.
    client.subscribe("$SYS/#")

# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
    print(msg.topic+" "+str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect("iot.eclipse.org", 1883, 60)

# Blocking call that processes network traffic, dispatches callbacks and
# handles reconnecting.
# Other loop*() functions are available that give a threaded interface and a
# manual interface.
client.loop_forever()
```

Example 1: output

```
…
$SYS/broker/load/publish/dropped/5min 4080.10
$SYS/broker/load/publish/received/5min 2113.46
$SYS/broker/load/publish/sent/5min 8473.40
$SYS/broker/load/bytes/received/5min 384447.30
$SYS/broker/load/bytes/sent/5min 1114325.30
$SYS/broker/load/sockets/5min 929.54
$SYS/broker/load/connections/5min 727.84
$SYS/broker/load/messages/received/15min 20789.77
$SYS/broker/load/messages/sent/15min 27349.78
$SYS/broker/load/publish/dropped/15min 3932.77
$SYS/broker/load/publish/received/15min 2152.81
$SYS/broker/load/publish/sent/15min 8699.76
$SYS/broker/load/bytes/received/15min 398937.27
$SYS/broker/load/bytes/sent/15min 1170197.30
$SYS/broker/load/sockets/15min 897.06
$SYS/broker/load/connections/15min 686.45
$SYS/broker/messages/stored 2018275
$SYS/broker/subscriptions/count 15864
…
```

Network loop

`loop(timeout=1.0)`

- Call regularly to process network events. This call waits in select() until the network socket is available for reading or writing, if appropriate, then handles the incoming/outgoing data.
- This function **blocks** for up to **`timeout`** seconds.
- `timeout` must not exceed the `keepalive` value for the client or your client will be regularly disconnected by the broker.
- **Better to use the following two methods**

`loop_start() / loop_stop()`

- These functions implement a threaded interface to the network loop. Calling `loop_start()` once, before or after connect (), runs a thread in the background to call `loop()` automatically. This frees up the main thread for other work that may be blocking. This call also handles reconnecting to the broker. For example:

```
mqttc.connect("iot.eclipse.org")
mqttc.loop_start()
while True:
    temperature = sensor.blocking_read()
    mqttc.publish("paho/temperature", temperature)
```

- Call loop_stop() to stop the background thread.

`loop_forever()`

- This is a **blocking** form of the network loop and will not return until the client calls `disconnect()`. It automatically handles reconnecting.

Example 2: subscriber with loop

```
import sys
import paho.mqtt.client as mqtt

def on_connect(mqttc, obj, flags, rc):
    print "Connected to %s:%s" % (mqttc._host, mqttc._port)

def on_message(mqttc, obj, msg):
    print(msg.topic+" "+str(msg.qos)+" "+str(msg.payload))

def on_publish(mqttc, obj, mid):
    print("mid: "+str(mid))

def on_subscribe(mqttc, obj, mid, granted_qos):
    print("Subscribed: "+str(mid)+" "+str(granted_qos))

def on_log(mqttc, obj, level, string):
    print(string)

# If you want to use a specific client id, use
# mqttc = mqtt.Client("client-id")
# but note that the client id must be unique on the broker.
# Leaving the client id parameter empty will generate a random id.
mqttc = mqtt.Client()
mqttc.on_message = on_message
mqttc.on_connect = on_connect
mqttc.on_publish = on_publish
mqttc.on_subscribe = on_subscribe

# Uncomment to enable debug messages
# mqttc.on_log = on_log
mqttc.connect("test.mosquitto.org", keepalive=60)
mqttc.subscribe("$SYS/broker/load/bytes/#", 0)

rc = 0
while rc == 0:
    rc = mqttc.loop()

print("rc: "+str(rc))
```

```
$>: python code4.py
Connected to test.mosquitto.org:1883
Subscribed: 1 (0,)
$SYS/broker/load/bytes/received/1min 0 2895441.54
$SYS/broker/load/bytes/received/5min 0 2886222.81
$SYS/broker/load/bytes/received/15min 0 2904064.82
$SYS/broker/load/bytes/sent/1min 0 6326013.29
$SYS/broker/load/bytes/sent/5min 0 5450954.94
$SYS/broker/load/bytes/sent/15min 0 4253739.61
$SYS/broker/load/bytes/received/1min 0 2907633.84
$SYS/broker/load/bytes/sent/1min 0 6260546.57
$SYS/broker/load/bytes/received/5min 0 2889175.18
$SYS/broker/load/bytes/sent/5min 0 5468388.62
$SYS/broker/load/bytes/received/15min 0 2904844.26
$SYS/broker/load/bytes/sent/15min 0 4274165.58
…
```

Example 3: subscriber with `loop_start/loop_stop`

```
import sys, time
import paho.mqtt.client as mqtt

def on_connect(mqttc, obj, flags, rc):
    print "Connected to %s:%s" % (mqttc._host, mqttc._port)

def on_message(mqttc, obj, msg):
    global msg_counter
    print(msg.topic+" "+str(msg.qos)+" "+str(msg.payload))
    msg_counter+=1

def on_publish(mqttc, obj, mid):
    print("mid: "+str(mid))

def on_subscribe(mqttc, obj, mid, granted_qos):
    print("Subscribed: "+str(mid)+" "+str(granted_qos))

def on_log(mqttc, obj, level, string):
    print(string)

msg_counter = 0

mqttc = mqtt.Client()
mqttc.on_message = on_message
mqttc.on_connect = on_connect
mqttc.on_publish = on_publish
mqttc.on_subscribe = on_subscribe

mqttc.connect("test.mosquitto.org", keepalive=60)
mqttc.subscribe("$SYS/broker/load/bytes/#", 0)

mqttc.loop_start()
while msg_counter < 10:
    time.sleep(0.1)
mqttc.loop_stop()
```

Example 4: very simple periodic producer

```
import paho.mqtt.client as mqtt
import time

mqttc=mqtt.Client()
mqttc.connect("localhost", 1883, 60)

mqttc.loop_start()
while True:
    mqttc.publish("pietro/test","Hello")
    time.sleep(10)  # sleep for 10 seconds before next call
mqttc.loop_stop()
```

```
1482241224: New client connected from 127.0.0.1 as paho/D00DA652C1C18AA67D (c1, k60).
1482241224: Sending CONNACK to paho/D00DA652C1C18AA67D (0, 0)
1482241224: Received PUBLISH from paho/D00DA652C1C18AA67D (d0, q0, r0, m0, 'pietro/test', ... (5 bytes))
1482241234: Received PUBLISH from paho/D00DA652C1C18AA67D (d0, q0, r0, m0, 'pietro/test', ... (5 bytes))
1482241245: Received PUBLISH from paho/D00DA652C1C18AA67D (d0, q0, r0, m0, 'pietro/test', ... (5 bytes))
1482241255: Received PUBLISH from paho/D00DA652C1C18AA67D (d0, q0, r0, m0, 'pietro/test', ... (5 bytes))
```

Example 5: Pub/Sub with JSON

Producer

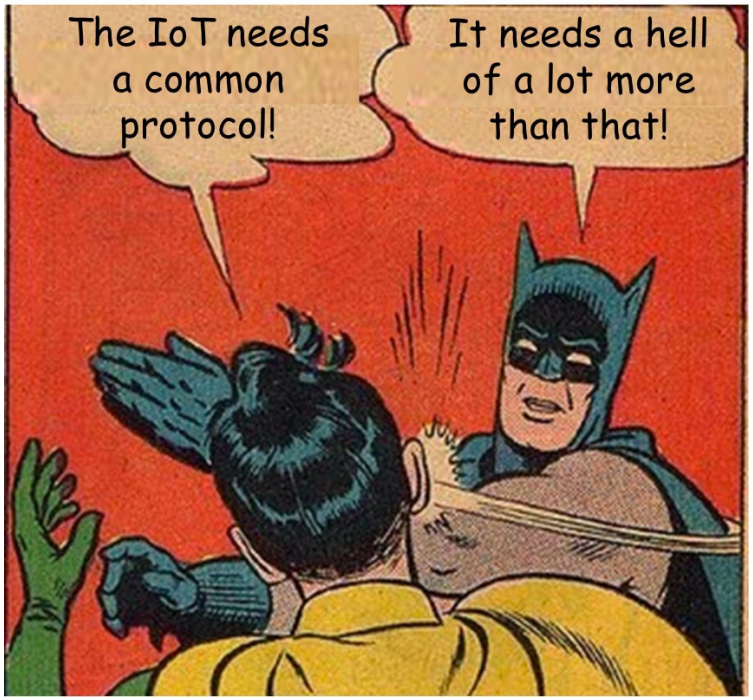
```
import paho.mqtt.client as mqtt
import time, random, json

mqttc=mqtt.Client()
mqttc.connect("localhost", 1883, 60)

mqttc.loop_start()

while True:
    # Getting the data
    the_time = time.strftime("%H:%M:%S")
    the_value = random.randint(1,100)
    the_msg={'Sensor': 1, 'C_F': 'C', 'Value': the_value,
'Time': the_time}
    the_msg_str = json.dumps(the_msg)

    print the_msg_str

    mqttc.publish("pietro/test",the_msg_str)
    time.sleep(5)# sleep for 5 seconds before next call

mqttc.loop_stop()
```

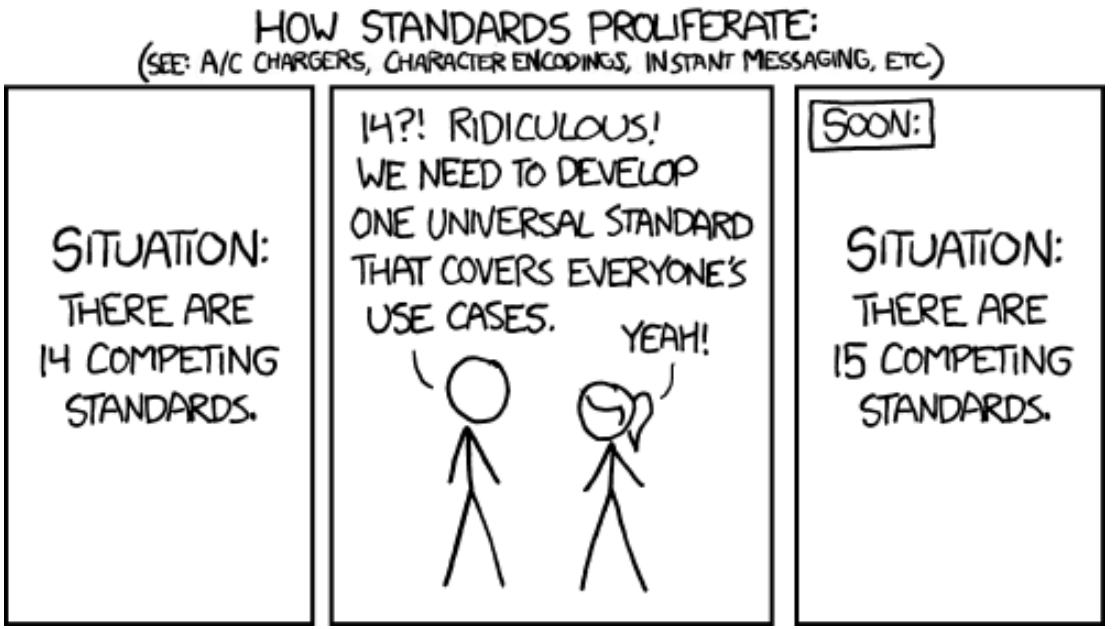
Consumer

```
import paho.mqtt.client as mqtt
import json

# The callback for when the client receives a CONNACK response
from the server.
def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))

# The callback for when a PUBLISH message is received from the
server.
def on_message(client, userdata, msg):
    print(msg.topic+" "+str(msg.payload))
    themsg = json.loads(str(msg.payload))
    print "Sensor "+str(themsg['Sensor'])+" got value ",
    print str(themsg['Value'])+" "+themsg['C_F'],
    print " at time "+str(themsg['Time'])

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect("localhost", 1883, 60)
client.subscribe("pietro/test")

client.loop_forever()
```

```
pietro/test {"Time": "22:31:11", "Sensor": 1, "Value": 86, "C_F": "C"}
Sensor 1 got value  86 C  at time 22:31:11
pietro/test {"Time": "22:31:16", "Sensor": 1, "Value": 90, "C_F": "C"}
Sensor 1 got value  90 C  at time 22:31:16
pietro/test {"Time": "22:31:21", "Sensor": 1, "Value": 24, "C_F": "C"}
Sensor 1 got value  24 C  at time 22:31:21
```

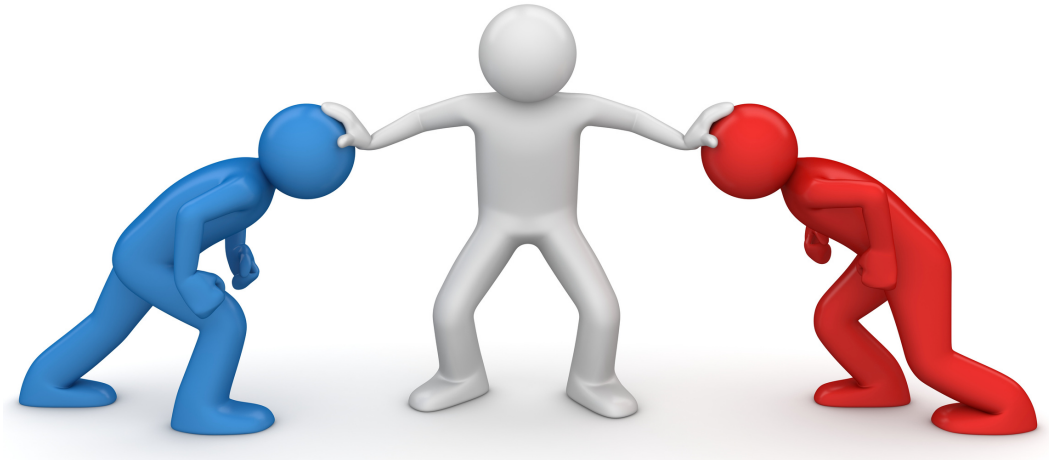
REST vs MQTT


The IoT needs a common protocol!
It needs a hell of a lot more than that!


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

MQTT vs REST

- Can they really be compared?!?!?
  - MQTT was created basically as a lightweight messaging protocol for lightweight communication between devices and computer systems
  - REST stands on the shoulders of the almighty HTTP
- So it's better to understand their weak and strong points and build a system taking the best of both worlds… if required



REST advantages

- The business logic is decoupled from presentation. So you can change one without impacting the other.
  - The uniform interface means that we don't have to document on a per-resource or per-server basis, the basic operations of the API.
- The universal identifiers embodied by URIs mean again that there is no resource or server specific usage that has to be known to refer to our resources
  - This assures that any tool that can work with HTTP can use the service.
- It is always independent of the type of platform or languages
  - The only thing is that it is indispensable that the responses to the requests should always take place in the language used for the information exchange, normally XML or JSON.

REST advantages

- It is stateless ➔ This allows for scalability, by adding additional server nodes behind a load balancer
  - Forbids conversational state. No state can be stored on servers: "keep the application state on the client."
  - All messages exchanged between client and server have all the context needed to know what to do with the message.
- It is cacheable ➔ you save bandwidth by caching responses from the server.
  - By using either expiry or validation (`Etag`). Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
  - This also provides an optimistic locking paradigm (also known as *Optimistic concurrency control - OCC*) through the use use of conditional requests. The `GET` method returns an `ETag` for a resource and subsequent `PUTs` use the `ETag` value in the `If-Match` headers; while the first PUT will succeed, the second will not, as the value in If-Match is based on the first version of the resource.

REST: HATEOAS

- HATEOAS, an abbreviation for Hypermedia As The Engine Of Application State, is a constraint of the REST application architecture that distinguishes it from most other network application architectures.
- The principle is that a client interacts with a network application entirely through hypermedia provided dynamically by application servers.
- A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia.
- A REST client enters a REST application through a simple fixed URL. All future actions the client may take are discovered within resource representations returned from the server.
  - The media types used for these representations, and the link relations they may contain, are standardized.
  - The client transitions through application states by selecting from the links within a representation or by manipulating the representation in other ways afforded by its media type.
  - In this way, RESTful interaction is driven by hypermedia, rather than out-of-band information.

REST: HATEOAS, an example with JSON

- For example, the following code represents a `Customer` object.

```
class Customer {
  String name;
}
```

- A HATEOAS-based response would look like this:

```
{
    "name": "Alice",
    "links": [ {
        "rel": "self",
        "href": "http://localhost:8080/customer/1"
    } ]
}
```

- This response not only has the person's name, but includes the self-linking URL where that person is located.
  - `rel` means relationship. In this case, it's a self-referencing hyperlink. More complex systems might include other relationships. For example, an order might have a `"rel":"customer"` relationship, linking the order to its customer.
  - `href` is a complete URL that uniquely defines the resource.

REST: HATEOAS, an example with XML

- Here is a GET request to fetch an Account resource, requesting details in an XML representation:

```
GET /accounts/12345 HTTP/1.1
Host: bank.example.com
Accept: application/xml
...
```

- Here is the response:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length:
 ...
<?xml version="1.0"?>
<account>
       <account_number>12345</account_number>
       <balance currency="usd">100.00</balance>
       <link rel="deposit" href="https://bank.example.com/accounts/12345/deposit" />
       <link rel="withdraw" href="https://bank.example.com/accounts/12345/withdraw" />
       <link rel="transfer" href="https://bank.example.com/accounts/12345/transfer" />
       <link rel="close" href="https://bank.example.com/accounts/12345/close" />
</account>
```

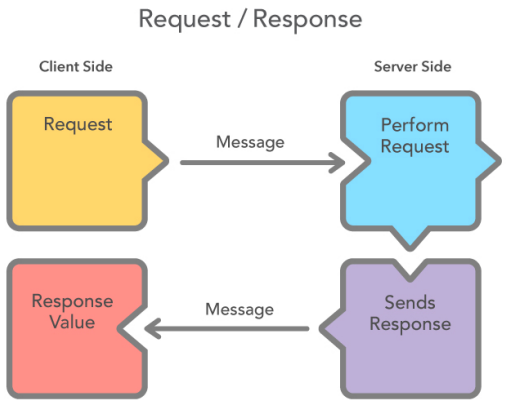
- Note the response contains 4 possible follow-up links - to make a deposit, a withdrawal, a transfer or to close the account.
- Some time later the account information is retrieved again, but now the account is overdrawn:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length:
...
<?xml version="1.0"?>
<account>
       <account_number>12345</account_number>
       <balance currency="usd">-25.00</balance>
       <link rel="deposit" href="https://bank.example.com/account/12345/deposit" />
</account>
```

- Now only one link is available: to deposit more money. In its current state, the other links are not available. What actions are possible vary as the state of the resource varies.

REST disadvantages

- Today's real world embedded devices for IoT usually lacks the ability to handle high-level protocols like HTTP and they may be served better by lightweight binary protocols.
- It is PULL based. This poses a problem when services depend on being up to date with data they don't own and manage.
  - Being up to date requires polling, which quickly add up in a system with enough interconnected services.
  - Pull style can produce heavy unnecessary workloads and bandwidth consumption due to for example a request/response polling-based monitoring & control systems
- It is based on one-to-one interactions



Push style in HTTP

- Attempts to improve non-polling HTTP communication have come from multiple sides:
  - The HTML 5 WebSocket API specifies a method for creating a persistent connection with a server and receiving messages via an **`onmessage`** callback.
  - Server-sent events (SSE) is a technology where a browser receives automatic updates from a server via HTTP connection. The Server-Sent Events EventSource API is standardized as part of HTML5
  - The BOSH protocol by the XMPP standards foundation. It emulates a bidirectional stream between browser and server by using two synchronous HTTP connections.
    - https://xmpp.org/extensions/xep-0124.html
  - Comet is a web application model in which a long-held HTTP request allows a web server to push data to a browser, without the browser explicitly requesting it.
    - Comet is known by several other names, including Ajax Push, Reverse Ajax, Two-way-web, HTTP Streaming, and HTTP server push among others.

Advantages of MQTT

- **Push based**: no need to continuously look for updates
- It has built-in function useful for reliable behavior in an unreliable or intermittently connected wireless environments.
  1. "last will & testament" so all apps know immediately if a client disconnects ungracefully,
  2. "retained message" so any user re-connecting immediately gets the very latest information, etc.
- Useful for one-to-many, many-to-many applications
- It's binary, so lower use of bandwidth… but see later
- Small memory footprint protocol, with reduced use of battery

Energy usage: some number

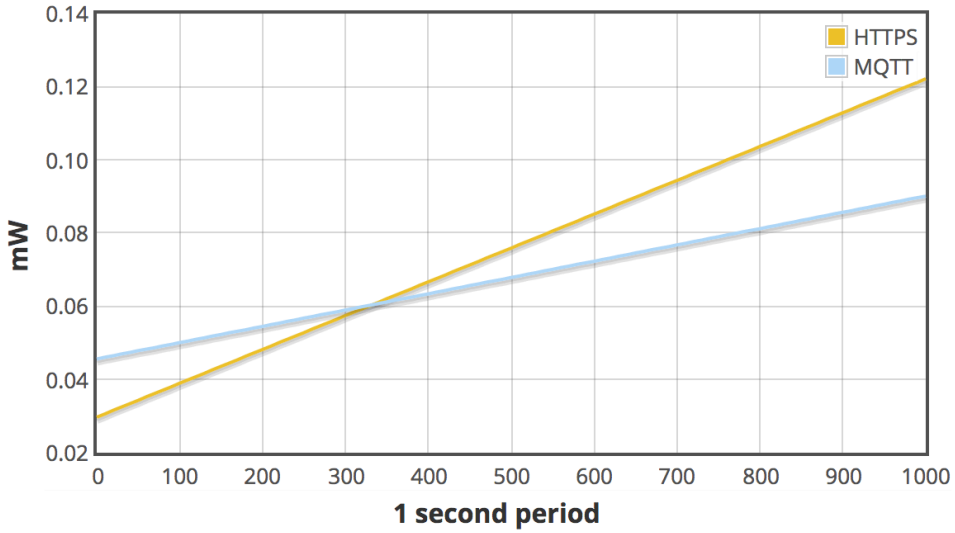
amount of power taken to establish the initial connection to the server:

% Battery Used			
3G		Wifi	
HTTPS	MQTT	HTTPS	MQTT
0.02972	0.04563	**0.00228**	0.00276

cost of 'maintaining' that connection (in % Battery / Hour):

	% Battery / Hour			
	3G		Wifi	
Keep Alive (Seconds)	HTTPS	MQTT	HTTPS	MQTT
60	1.11553	**0.72465**	0.15839	**0.01055**
120	0.48697	**0.32041**	0.08774	**0.00478**
240	0.33277	**0.16027**	0.02897	**0.00230**
480	0.08263	**0.07991**	0.00824	**0.00112**

3G – 240s Keep Alive – % Battery Used Creating and Maintaining a Connection

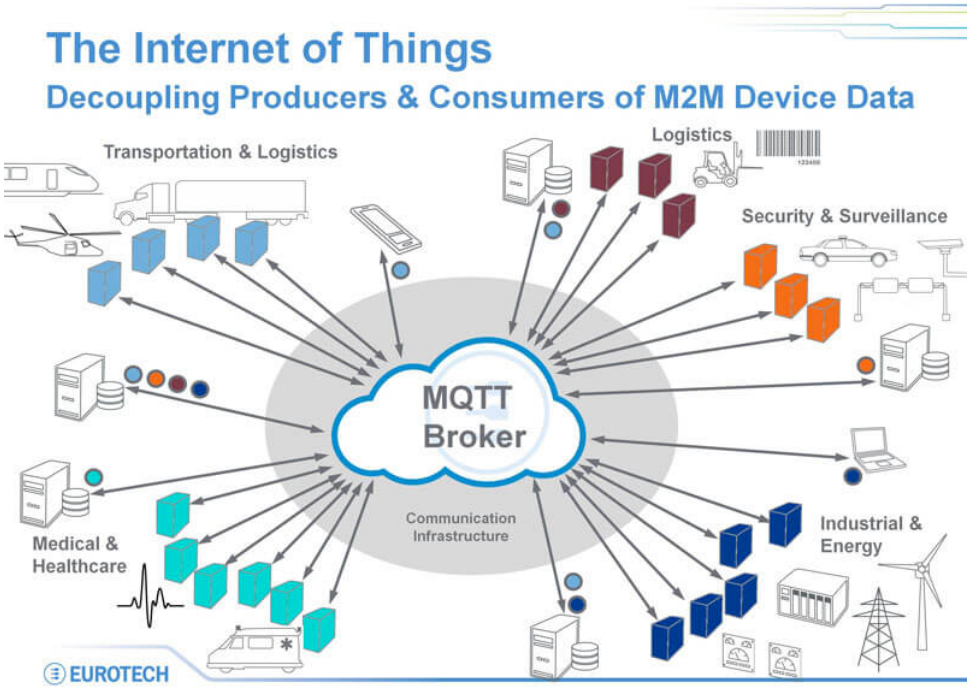


you'd save ~4.1% battery per day just by using MQTT over HTTPS to maintain an open stable connection.

http://stephendnicholas.com/posts/power-profiling-mqtt-vs-https

MQTT advantages: decoupling and filtering

- Decoupling of publisher and receiver can be differentiated in more dimensions:
  - **Space decoupling**: Publisher and subscriber don't need to know each other (by IP address and port for example)
  - **Time decoupling**: Publisher and subscriber do not need to run at the same time
  - **Synchronization decoupling**: Operations on both components are not halted during publish or receiving
- **Filtering** of the messages makes possible that only certain clients receive certain messages.
  - MQTT uses subject-based filtering of messages. So each message contains a topic, which the broker uses to find out, if a subscribing client will receive the message or not.



MQTT disadvantages

- Does not have a point-to-point (aka queues) messaging pattern
  - Point to Point or One to One means that there can be more than one consumer listening on a queue but only one of them will be get the message
- Does not define a standard client API, so application developers have to select the best fit.
- Does not include message headers and other features common to messaging platforms.
  - developers frequently have to implement these capabilities in the message payload to meet application requirements thus increasing the size of the message and increasing the BW requirements.
- Does not include many features that are common in Enterprise Messaging Systems like:
  - expiration, timestamp, priority, custom message headers, …
- Maximum message size 256MB
- If the broker fails…

So, trying to summarize

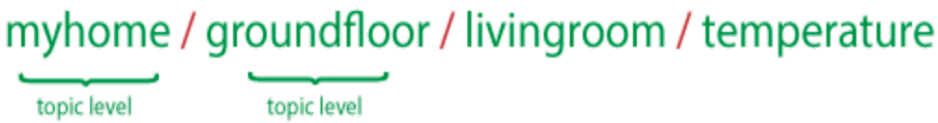
- REST is not the best option even though it's an established and great protocol for request/response solutions, when:
  - Dealing with low-power devices that cannot handle a web server
  - Push and quality of service channels are necessary
  - Text-based requires more bandwidth
  - Battery duration is a critical issue

- In these cases MQTT offers:
  - Publish and subscribe with quality of service
  - Binary format that requires less bandwidth
  - Small implementation footprint
  - Lower battery requirements

Combining the two: Topics and URLs

- A very useful characteristic of MQTT topic structure is that it can contain delimiters ('/'), which opens up a possibility to sync up REST URLs and topics.
  - See: *MQTT Version 3.1.1*, OASIS Standard, 2014. For instance, the resource */publish/room/237/temperature* is mapped to the topic *room/237/temperature*
- This prompted some developers to go for full parity
  ➔ essentially using the REST URL as a topic.
  - This is probably not necessary: as long as the segments that matter match, we don't need to have the same root.
  - In generally the entire URL is not required as a topic other than symbolically, unless you are writing a 'superbroker' – a server that is both a broker and a REST server.

http://www.boeing.com/aircraft/747

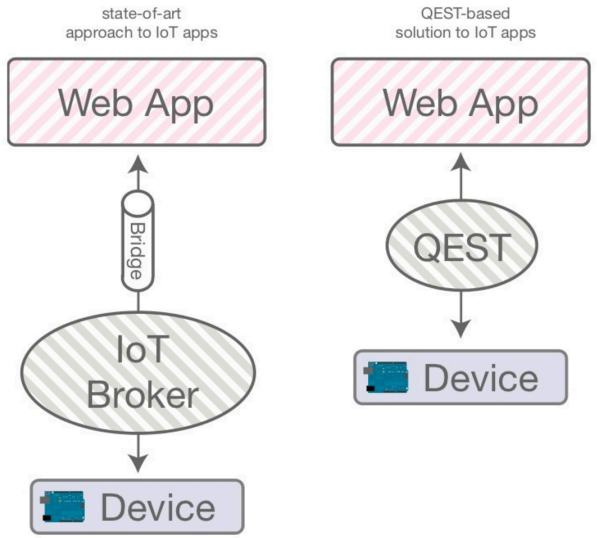


Combining the two: limitations

- Since HTTP uses a request/response mechanism, it cannot fully support a publish/subscribe mechanism.
  - WebSockets could be one possible solution to solve this issue.
  - But some author argues that WebSockets do not implement the concept of URI after opening the communication and, therefore, do not support the pure REST approach (resource-topic mapping).
  - They enhanced the implementation by a Long-Polling approach for retrieving real-time updates in the browser without using WebSockets.

Combining the two: QEST

- Project QEST, an attempt to bridge the world of apps speaking REST and the world of devices speaking MQTT with one bilingual broker.
  - "a mashup of MQTT and REST"
- https://github.com/mcollina/qest
- *M. Collina, G. E. Corazza and A. Vanelli-Coralli, "Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST," 2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications - (PIMRC), Sydney, NSW, 2012, pp. 36-41. doi: 10.1109/PIMRC.2012.6362813*



The QEST broker

Exposing MQTT topics as REST resources

- MQTT topics do not retain state. To expose them as a REST resource, we have to publish the last payload seen on the topic as the resource value
- Exposing the topics is possible by manipulating a single value.
- Following a pure REST style, we can create a topic just by issuing an HTTP POST request at `/topics` containing both the topic's name and its original payload.
- For example, if the topic's name is `light_bulb` we can:
  - read the last published value by doing an HTTP `GET` request at `/topics/light_bulb`
  - publish a value in the topic by doing an HTTP `PUT` request at `/topics/light_bulb`
- This approach can guarantee only a best-effort level of Quality of Service, because that is the true nature of the HTTP protocol.

Exposing REST resources as MQTT topics

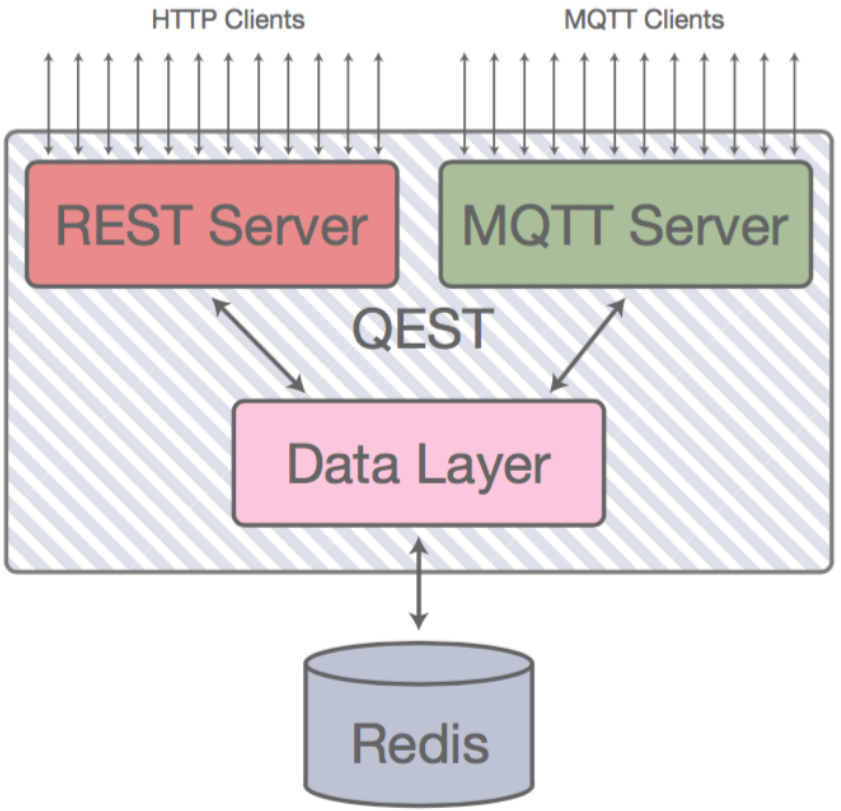
- The MQTT protocol is stateless in regard of payloads. We need to store the latest published value of a topic for its publication.
- In MQTT, when a machine subscribes to a topic, it does not know the value of the topic until a new payload is published. This is very far from HTTP publishing model.
  - Therefore the broker sends the last published value to the newly connect client.
  - This is an important change because it deviates from the pub/sub nature of the MQTT protocol.

QEST system architecture

- QEST revolves around the fast key-value store Redis, which acts as the internal broker of the application.
- A single QEST node consists of two main components, the REST (web) server, and the MQTT server: both provide a representation of the data stored in Redis, which is accessed through the data abstraction layer.
- QEST is a multiprocol broker with a common semantics, and its main responsibility is to notify subscribers on the update of a particular topic, which is achieved as follows:
  1. the client publishes the update either through the REST or the MQTT front-end;
  2. the front-end writes the new value on the data layer;
  3. the data layer stores the value and publishes an update on a specific Redis key.
- In order to receive updates on a topic, the client acts as follows:
  1. the client subscribes for updates to either the REST or the MQTT front-end, according to its nature;
  2. the front-end registers for changes to the data layer;
  3. the data layer subscribes to changes to a particular Redis key;
  4. whenever is published a value on the Redis key, it is forwarded to the data layer;
  5. the data layer notifies the subscribed clients through the front-ends.

⇦ https://redis.io/

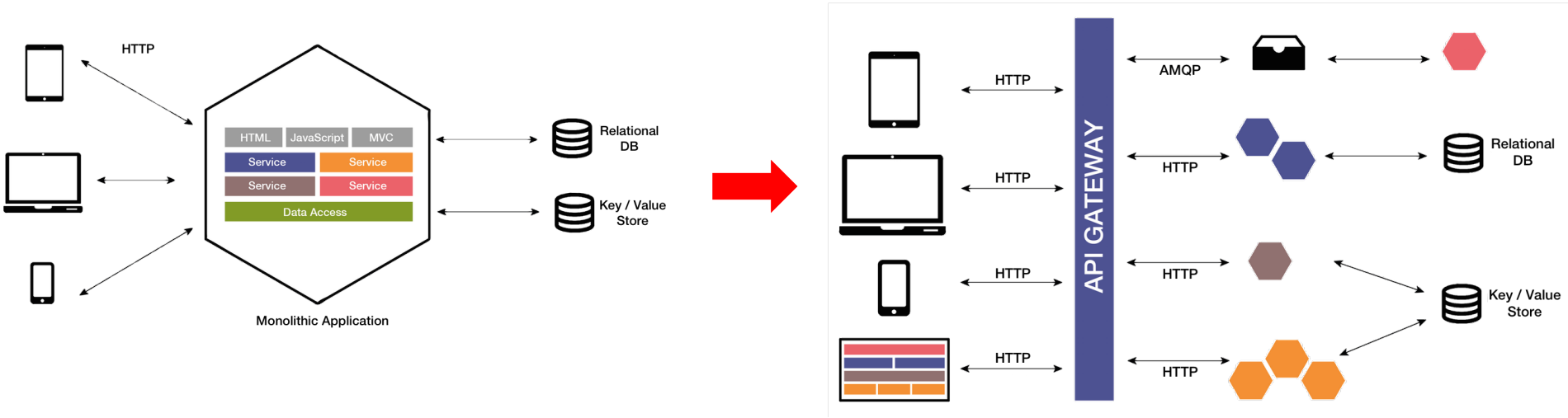
Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker.



Combining the two: Dejan Glozic (IBM) proposal

- Proposal that combines REST and MQTT in the context of micro-service-based distributed systems.
  - *Microservices* are small, autonomous services that work together

- Warning regarding terminology: even simple entities are called “services”

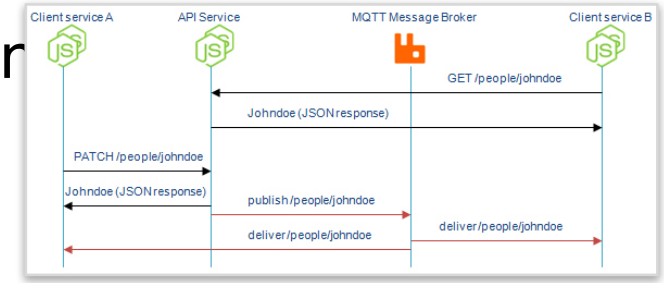


Combining the two: Dejan Glozic (IBM) proposal

- REST poses a problem when services depend on being up to date with data they don't own and manage.
- Being up to date requires polling, which quickly add up in a system with enough interconnected services.
- However, the problem with this data flow reversal is that the responsibility of storing the data is put on the event recipients.
  - Storing the data in event subscribers allows them to be self-sufficient and resilient – they can operate even if the link to the event publisher is temporarily severed.
  - Basic pragmatic assumption: the message broker can fail!

DG proposal: how it works

- Augmenting REST with messaging results in a very powerful combination.
- This is how it works:



1. The service with a REST API will field requests by client services as expected. This establishes the **baseline state**.
2. All the services will simultaneously connect to a message broker.
3. API service will fire messages notifying about data changes (essentially for all the verbs that can cause the change, in most cases POST, PUT, PATCH and DELETE).
4. Clients interested in receiving data updates will react to these changes according to their functionality.
5. In cases where having the correct data is critical, client services will forgo the built-up baseline + changes state and make a new REST call to establish a new baseline before counting on it.

DG proposal: how it works

- Messages are used to augment, not replace REST.
- While message brokers are reliable and there are ways to further increase this durability, REST is still counted on establishing a 'clean slate'.
  - Of course, REST can fail too, but if it does, you have no data, as opposed to old and therefore incorrect data.
- Client services are not required to store data. For this to work, they still need to track the baseline data they obtained through the REST call and be able to correlate messages to this baseline.

DG proposal: REST/MQTT API in action

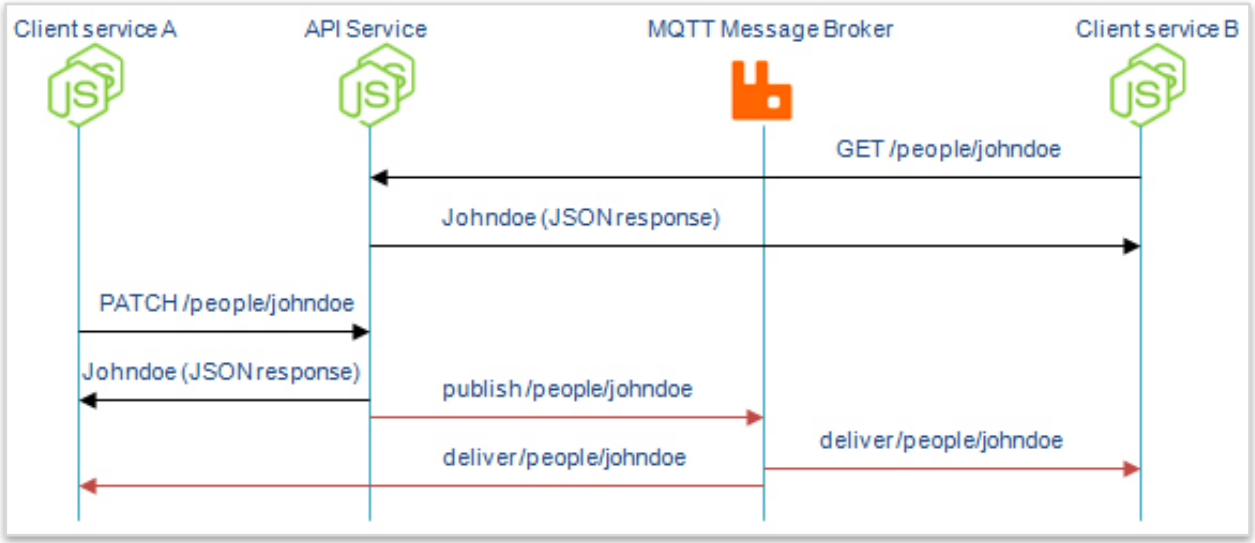
- So how does this approach look in practice? Essentially, you start with a normal REST API and add MQTT messages for REST endpoints that result in a state change (POST/PUT/PATCH/DELETE).
- For example, let's say we have an API service responsible for serving people profiles. The REST endpoints may look something like this:
- **GET /people** – this returns an array of JSON objects, one per person
- **GET /people/:id** – this returns a single JSON object of a person with the provided id, something like:

```
{
  "id": "johndoe",
  "name": "John Doe",
  "email": "jdoe@example.com"
}
```

- **PATCH /people/:id** – this updates select fields of the person.

DG proposal: REST/MQTT API in action

- The sequence diagram of using such an API service starts with the client service B making an HTTP GET request to fetch a resource for John Doe.
- API service will return JSON for the requested person as expected.
- After that, another service (client A) issues a PATCH request to update John Doe's email address. API service will execute the request, return updated JSON for John Doe in the response, then turn around and publish a message to notify subscribers that '/people/johndoe' has changed.
- This message is delivered to both clients that are subscribed to 'people/+' topics (i.e. changes to all people resources). This allows service B to react to this change.



DG proposal: Topics and message bodies

- Since MQTT is now part of the formal API contract, we must document it for each REST endpoint that causes state change. There is no hard and fast rule on how to do this, but we are using the following conventions:
- POST endpoints publish a message into the matching MQTT topic with the following shape:

```
{
  "event": "created",
  "state": { /* JSON returned in the POST response body */ }
}
```

- PUT and PATCH endpoints use the following shape:

```
{
  "event": "modified",
  "changes": { "email": "johndoe@example.com" }
}
```

- The shape above is useful when only a few properties have changed. If the entire object has been replaced, an alternative would be:

```
{
  "event": "modified",
  "state": { /* JSON returned in the PUT response body */ }
}
```

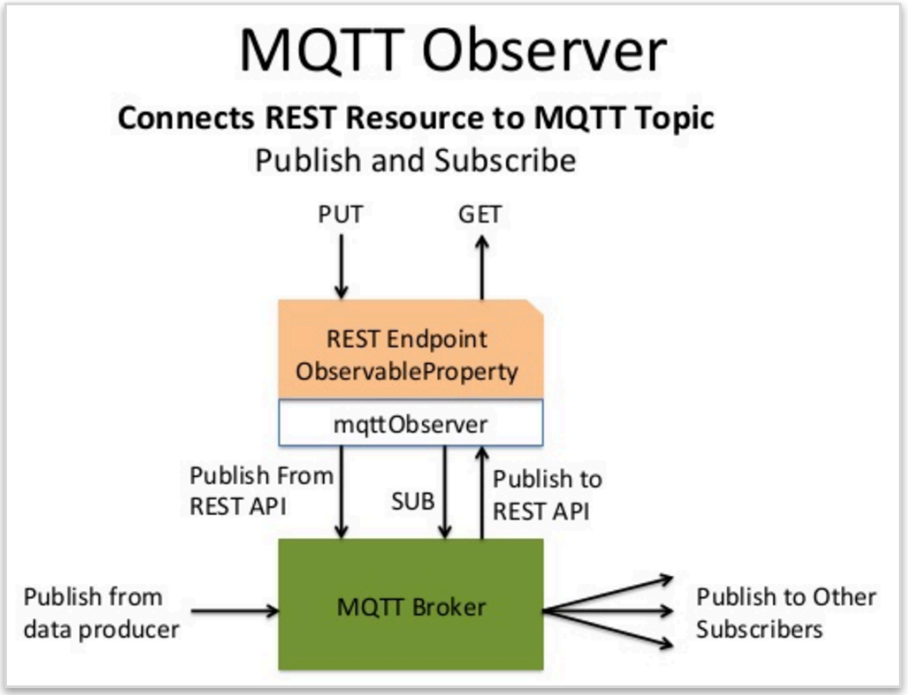
- Finally, a message published upon a DELETE endpoint looks like this:

```
{
  "event": "deleted"
}
```

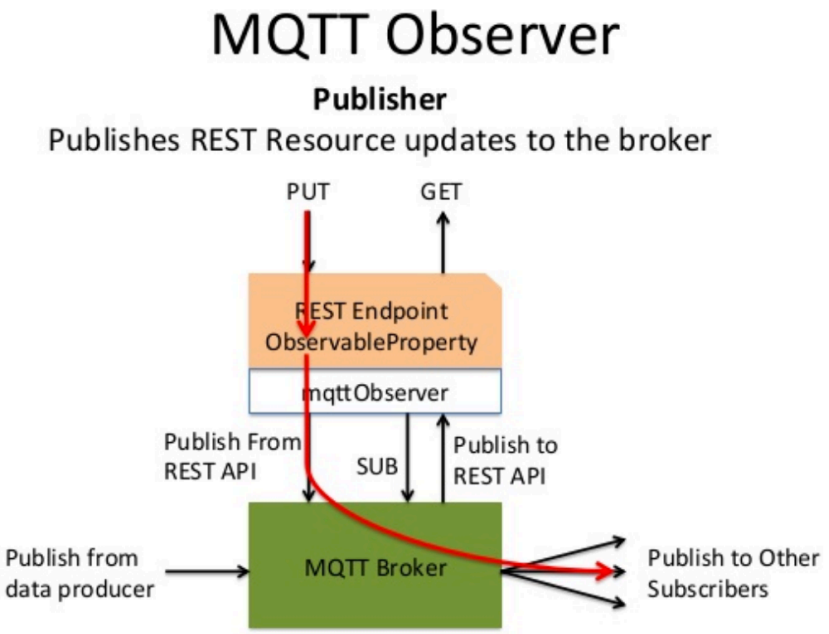
MQTT - REST Bridge using the Smart Object API

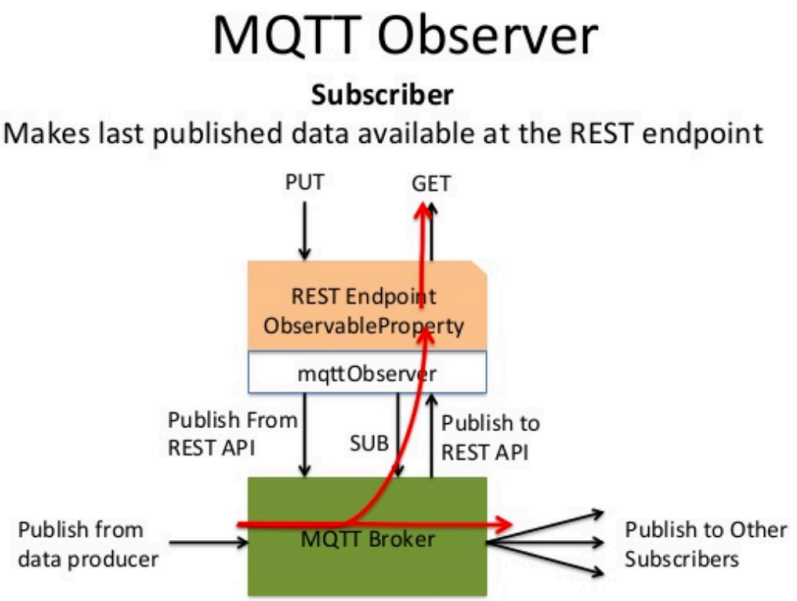
- Michael Koster:http://iot-datamodels.blogspot.com.es/
- Implements the MQTT abstraction layer
- Publish, Subscribe, or Pub+Sub using the mqttObserver resource class
- Prototype opens a connection for each REST endpoint

```
Observers.create({'resourceName': 'mqttTestObserver',\
                  'resourceClass': 'mqttObserver',\
                  'connection': 'smartobjectservice.com',\
                  'pubTopic': 'sealevel_pressure',\
                  'subTopic': None,\
                  'QoS': 0,\
                  'keepAlive': 60 })
```



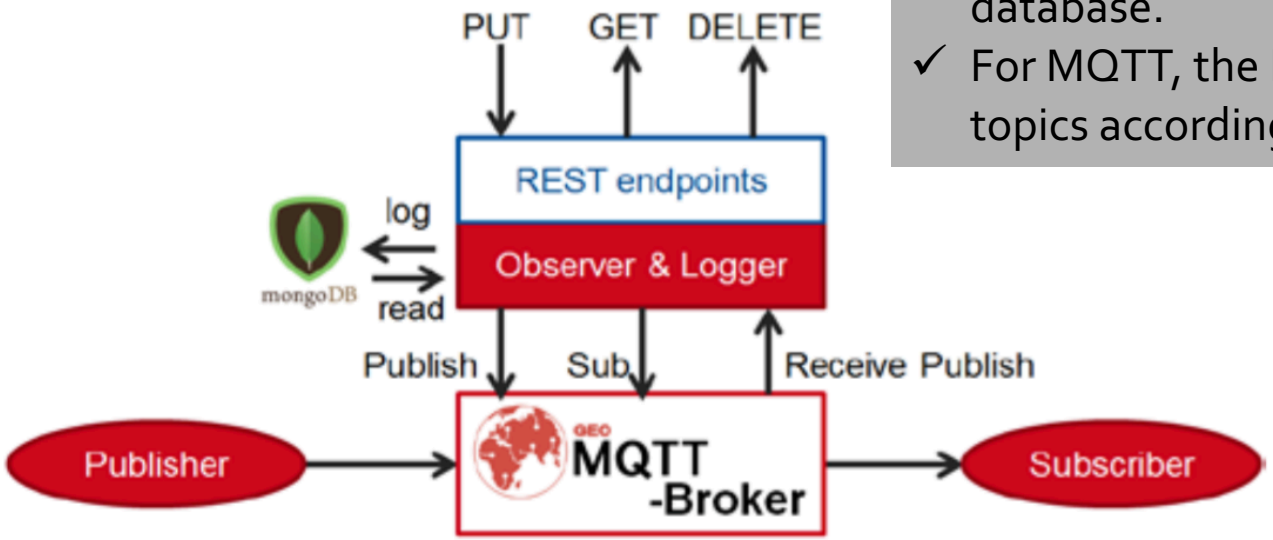
MQTT - REST Bridge using the Smart Object API





Similar approach: Bridging GeoMQTT and REST

- Work by Stefan Herle, Ralf Becker & Jörg Blankenbach, RWTH Aachen University, Germany herle@gia.rwth-aachen.de



- ✓ The observer & logger client subscribes to all MQTT (and GeoMQTT) messages at the broker.
- ✓ It logs the publish and GeoPublish messages received in separate collections in a MongoDB database.
- ✓ For MQTT, the REST resources are mapped to topics according to the approach seen before