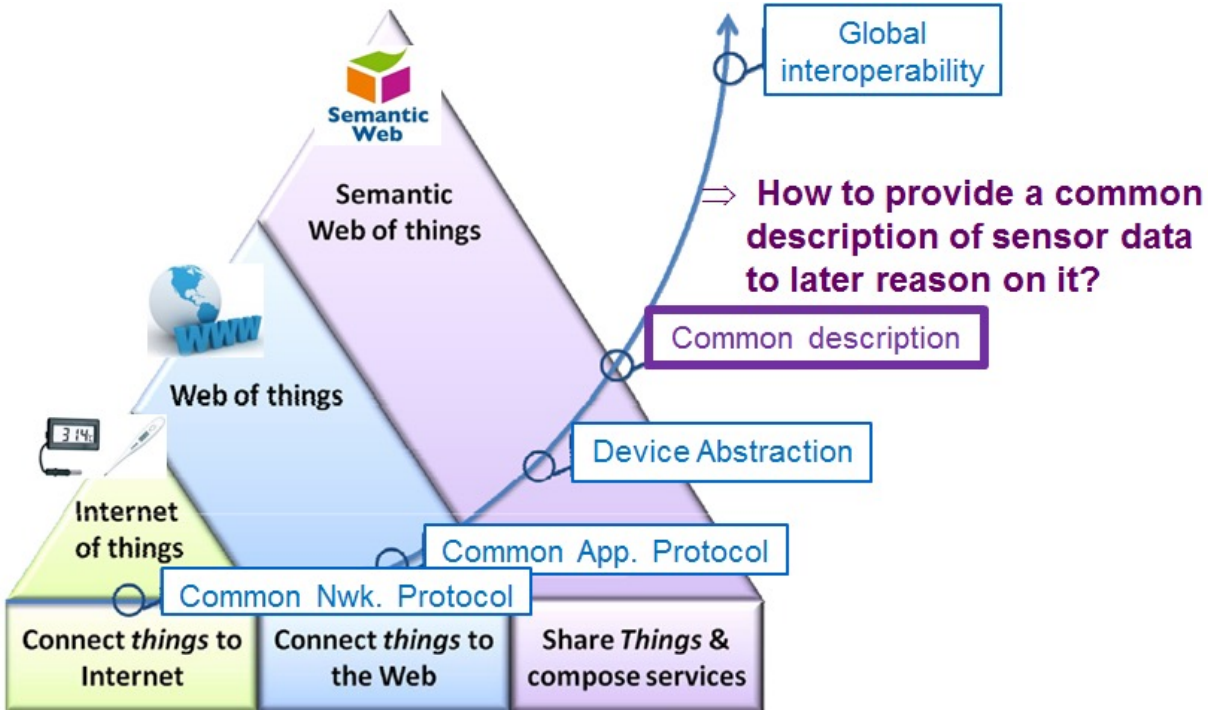
IoT Application Layer

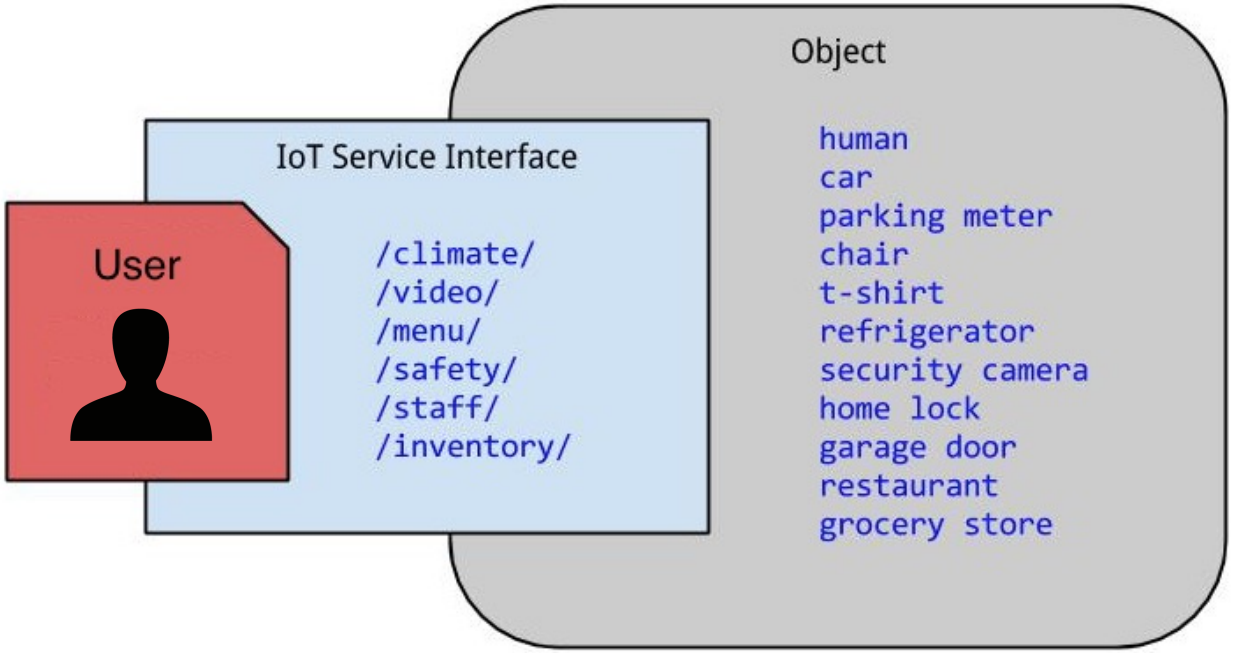
Connecting Things to the Web

- Meta-data models and schemas
- Linked data and Linked IoT data concepts
- Semantic technologies and semantic sensor networks
- Interoperability issues
- Web of Things



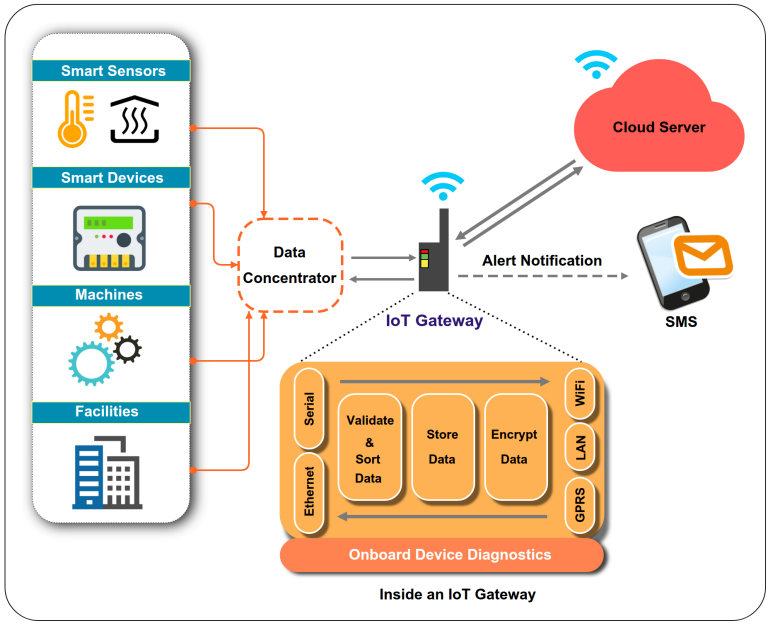
Service interfaces and Web connectivity

- IoT nodes are typically resource constrained
  - Memory, process and power limitations
  - Communication load
- Often do not support IP or use 6LowPAN
- Nodes may directly connect to the Web and support a service interface

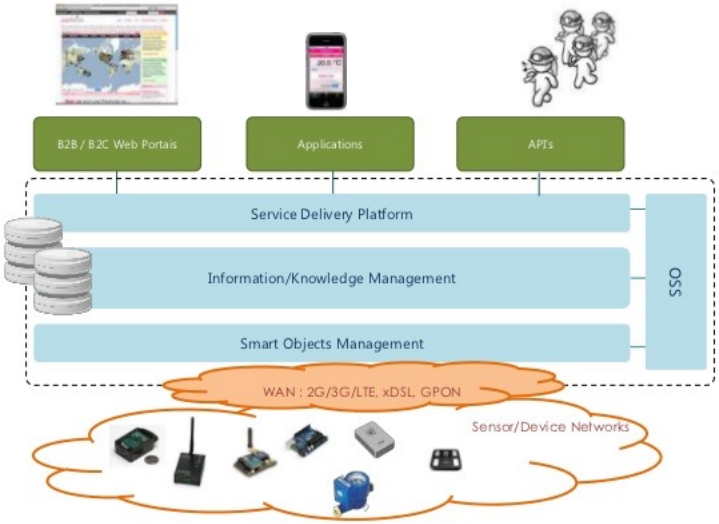


Using gateway and middleware

- It is unlikely that everything will be IP enabled and will run TCP/IP stack
- Gateway and middleware solutions can interface between low-level sensor island protocols and IP-based networks.
- The gateway can also provide other components such as QoS support, caching, mechanisms to address heterogeneity and interop

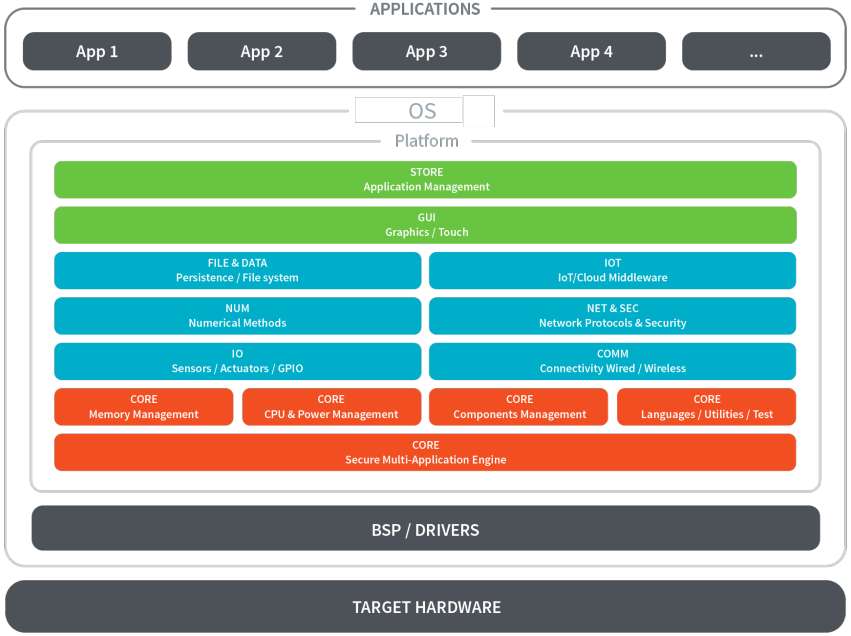


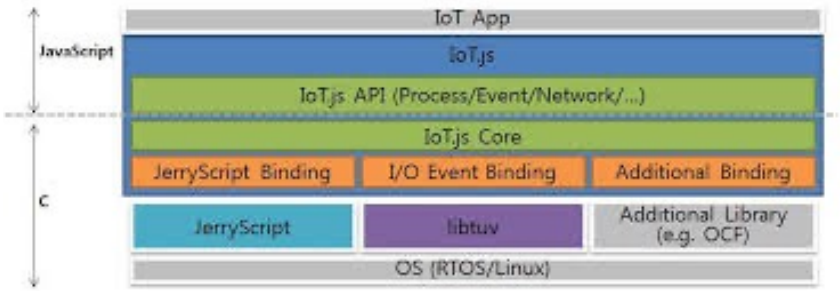
Inside an IoT Gateway



Service interfaces to IoT nodes

- Supporting high-level request/response interactions
- Async. event notifications
- Identifying and accessing data
  - By location, observed entity,
    - By semantically meaningful representations – “Room 35B”
- Accessibility of in-network processing functions
- Accessing node status (e.g., battery level, connectivity)
  - Security, management, …
- There are emerging solutions and standards in this domain supported by Semantic Web technologies and Linked-data





How to organize collaborating IoT objects?

- Principle of Least Knowledge:
  - Reveal only what's necessary.

- Problem: how does the information get to where it's needed?

- What happens in a stateful system?
- Objects may need to know each other's identity:
  - either to *pull* information from that object
  - or to *push* information to that object

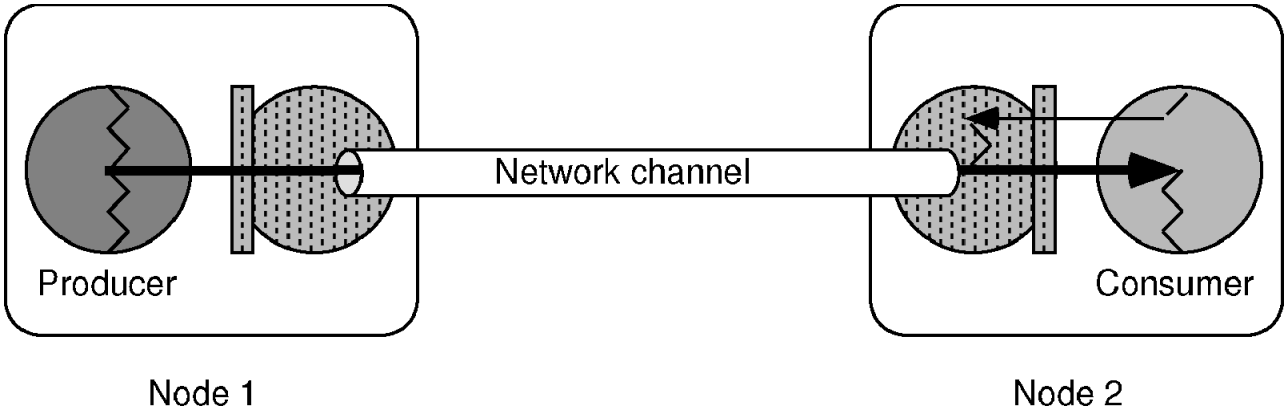
- We can structure the interface according to different service models

Alternative Communication Paradigms

- The options:
  - Message Passing
  - Remote Procedure Call (RPC)
  - Shared Spaces
  - Message Queuing
  - Publish Subscribe

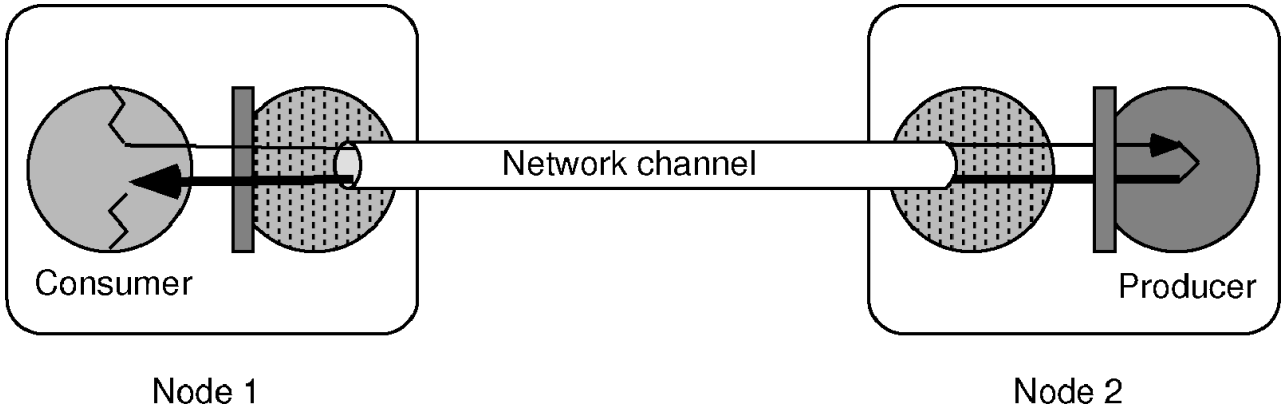
Message Passing

- Viewed as ancestor of distributed interactions.
- Participants communicate by sending and receiving messages asynchronously through a network channel.
- Producer and Consumer are coupled in time and space.
  - Asynchronous for producer but not consumer
  - Channel set up ahead of time, producer and consumer are active at same time
- Recipient of message known to sender



Remote Procedure Call (RPC)

- One of the most widely used forms of distributed interaction.
- Remote interactions appear the same as local interactions, make distributed programming easy.
- Each request is followed by a response.
- Strong time and synchronization (on the consumer side) coupling.
- Coupled in space - an invoking object holds a remote reference to each of its invokees.



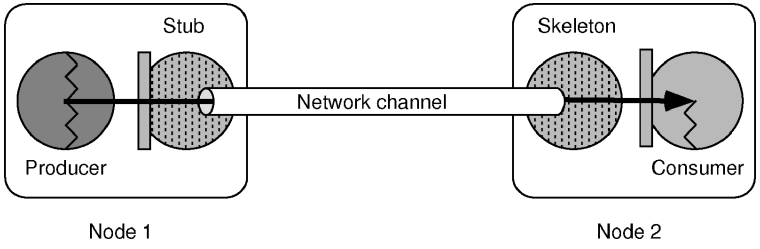
Attempts to remove synchronization in RPC

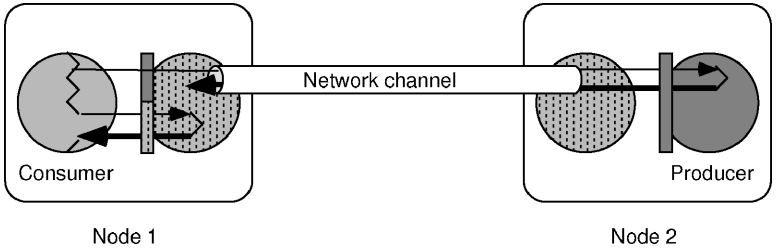
Asynchronous rpc

- Decouples synchronization by making the producer not expect a reply.

Future RPC

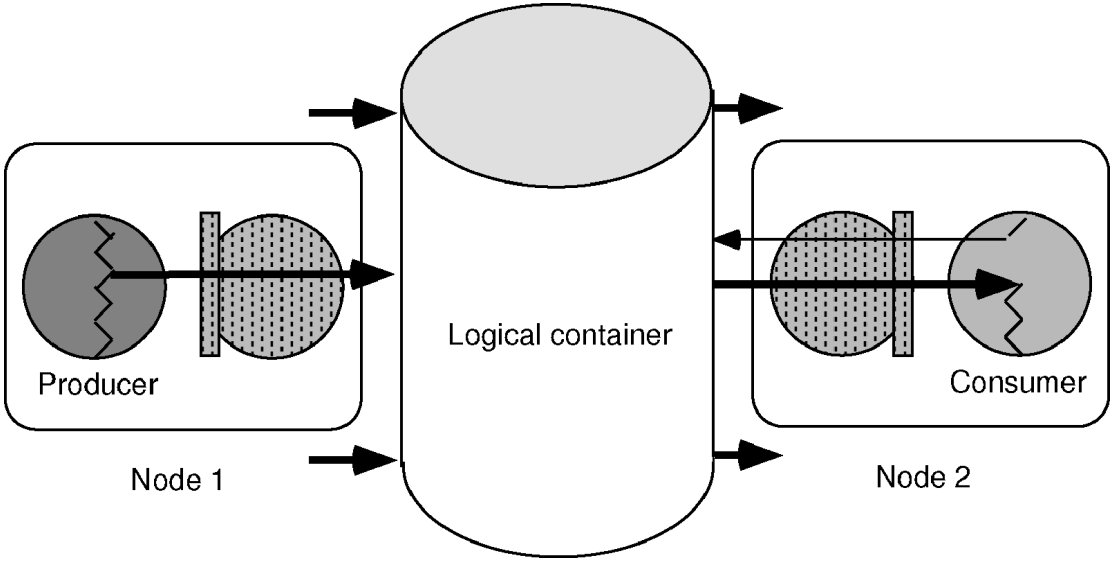
- Decouples synchronization by not blocking the producer, they can access the reply later when it becomes available.





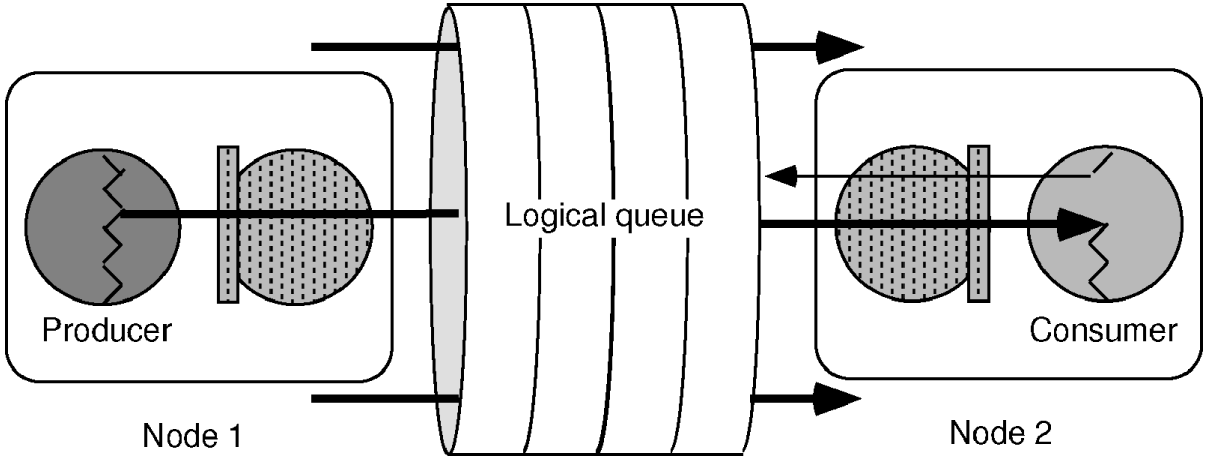
Shared Spaces

- Distributed shared memory, common to all participants who interact by reading and writing to it
- Many-to-many anonymous interaction – time and space decoupling
- Consumers not asynchronously notified of messages, but retrieve messages with a synchronous request.



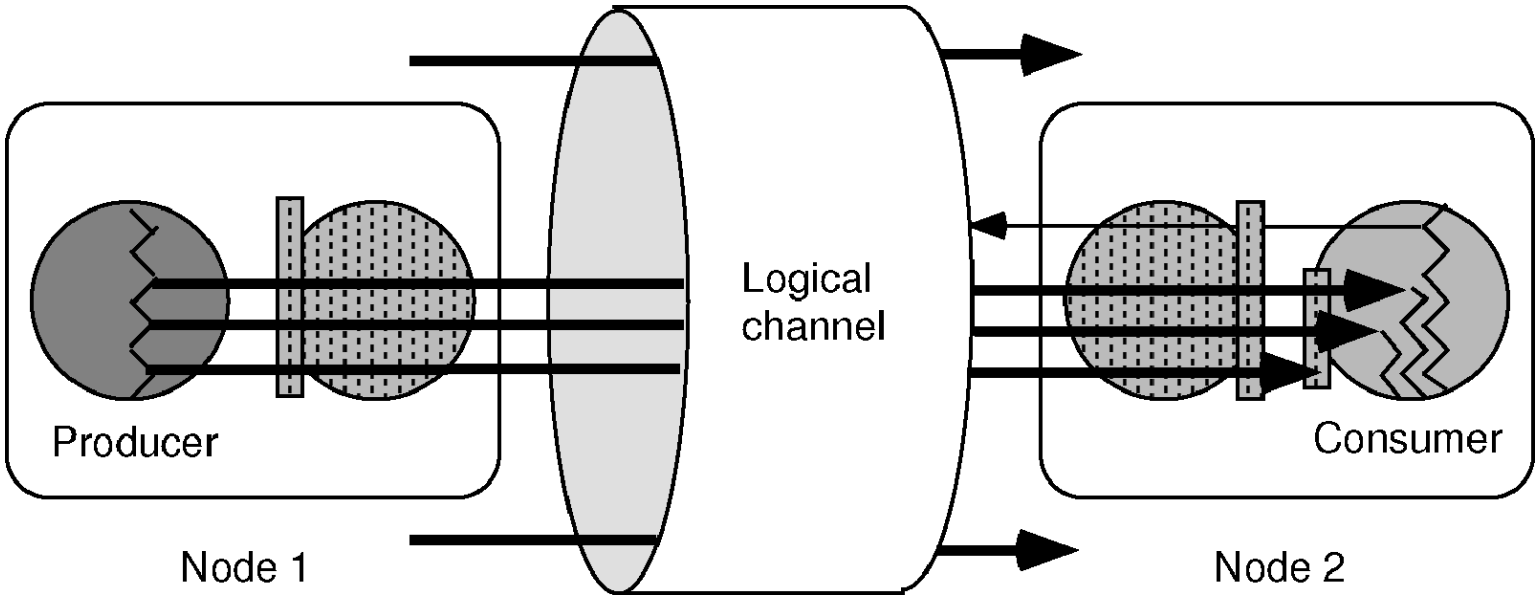
Message Queuing

- Messages are persistently stored within queues
- All communications filtered by the queue – similar to the Notification Service in Pub/Sub
- Consumers must explicitly pull messages from the queue – No synchronization decoupling on consumer side



The Publish/Subscribe Service Paradigm

- Publish-Subscribe is a service paradigm for implementing push-style communication between IoT objects over time.
- When information changes, the person who changes it pushes it out to the people who need to know.
- How does the information-changer know who to tell?
  - The information-needer must *register* with the information-changer.

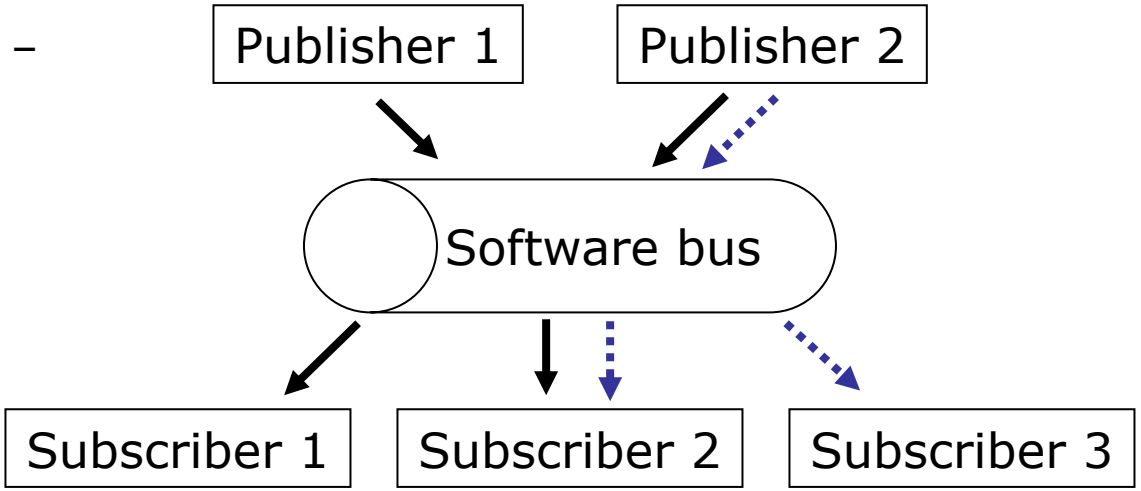


The Publish/Subscribe Service Paradigm

- The Publish/Subscribe service interface allows a message to be published once and multiple consumers (applications / devices) to receive the message providing decoupling between the producer and consumer(s)
  - A publisher keeps a list of subscribers.
  - When the publisher changes state, it sends a message notifying each of its subscribers about the state change.
  - Each subscriber changes its local state to take note of the messages it receives from the publisher.
  - Now, the subscriber can consult its local state instead of sending queries to the publisher.
  - Good if queries are much more frequent than state changes.

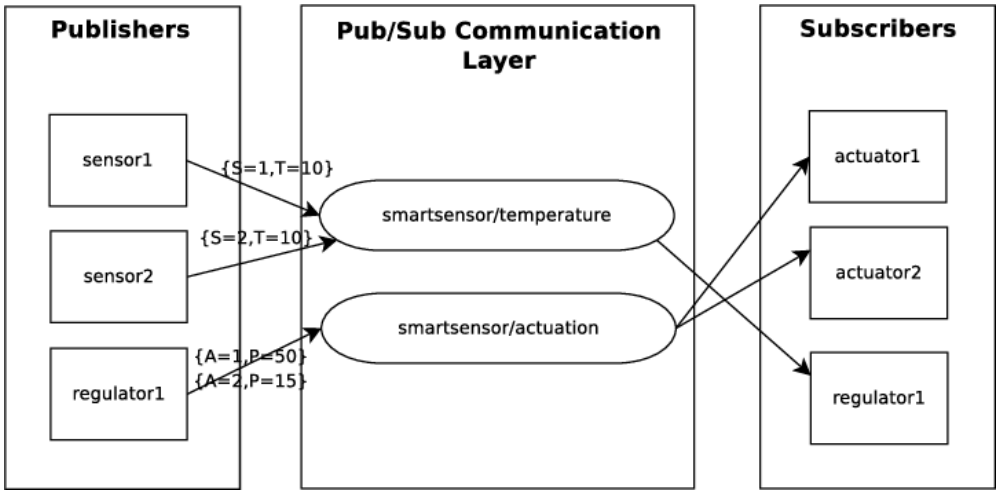
The Publish/Subscribe Service Paradigm

- Conceptually: Implemented by a software bus
  - Software bus stores subscriptions, published data; names used as filters; subscribers notified when values of named data changes
  - Variations
    - Topic-based P/S – inflexible
    - Content-based P/S – use general predicates over named data
    - Type-based P/S



Topic-Based Publish/Subscribe

- Earliest pub/sub schemes were based on the notion of *topics* or *subjects*
- *Topics* similar to the notion of *groups*
  - Subscribing to a topic *T* can be viewed as becoming a member of a group *T*, and publishing an event on topic *T* translates to broadcasting an event among members of *T*
- Programming abstraction which maps individual topics to distinct communication channels (many-to-many relationships)



Topic-Based Pub/Sub

- Hierarchy
  - A subscription made to some node in the hierarchy implicitly involves subscriptions to all the subtopics of that node.
- Wildcards
  - Offer the possibility to publish and subscribe to several topics that match a given set of keywords at the same time
    - E. g. an entire subtree or specific level of hierarchy

- ✔ myhome / groundfloor / livingroom / temperature
- ✔ myhome / groundfloor / kitchen / temperature
- ✘ myhome / groundfloor / kitchen / brightness
- ✘ myhome / firstfloor / kitchen / temperature
- ✘ myhome / groundfloor / kitchen / fridge / temperature

Content-Based Publish/Subscribe

- Subscriptions are related to specific information content
  - More flexible than Topic-Based
- Consumers subscribe to selective events by specifying filters to define constraints
  - Filters: *name-value* pairs, or comparison operators (=, <, ≤, >, ≥)
  - Logically combined to form complex subscription patterns
- Each combination of information items can be seen as a single dynamic logical channel (more of a many-to-one relationship)

Content-Based: Subscription Patterns

- Strings
  - Most frequently used
  - Must conform to the subscription grammar
    - SQL, Xpath, or some proprietary language
  - Parsed by the engine

- Template object
  - A participant provides an object *t*, which means they are interested in every event that conforms to type *t* and whose attributes all match the attributes of *t*.

- Executable code
  - Subscribers provide a predicate object able to filter events at runtime.
  - Implementation left to developer, hard to optimize

Type-Based Publish/Suscribe

- Events are filtered by their type
  - Example: Stock events can be split into two distinct types*: stock quotes* and *stock requests*
- Reuses type scheme of object-oriented languages, provides seamless integration between middleware and programming language
- Aims at providing guarantees such as type safety and encapsulation.

Publish/Subscribe Messaging

- A producer publishes a message (publication) on a topic (subject)
- A consumer subscribes (makes a subscription) for messages on a topic (subject)
- A Message-Oriented server/broker matches publications to subscriptions
    - If no matches the message is discarded
    - If one or more matches the message is delivered to each matching subscriber/consumer

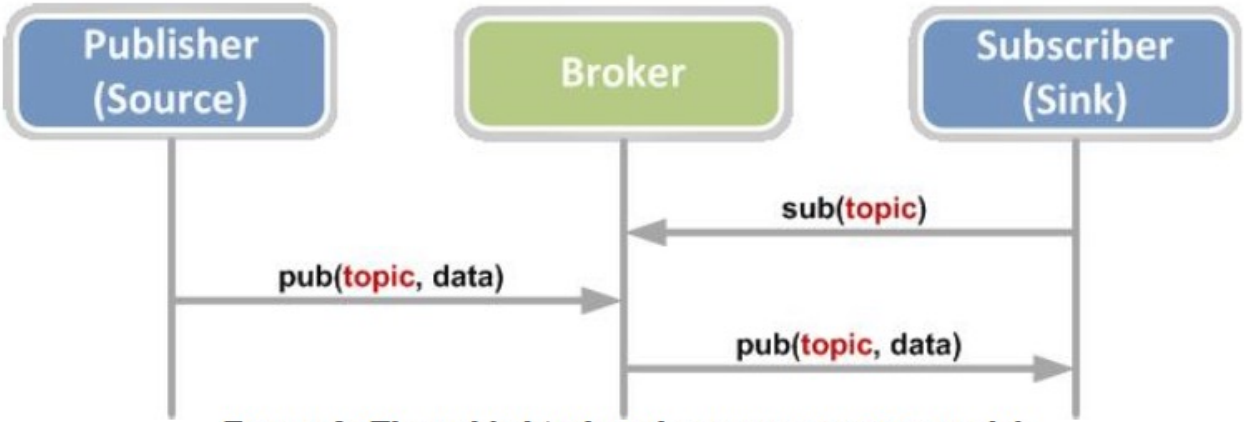
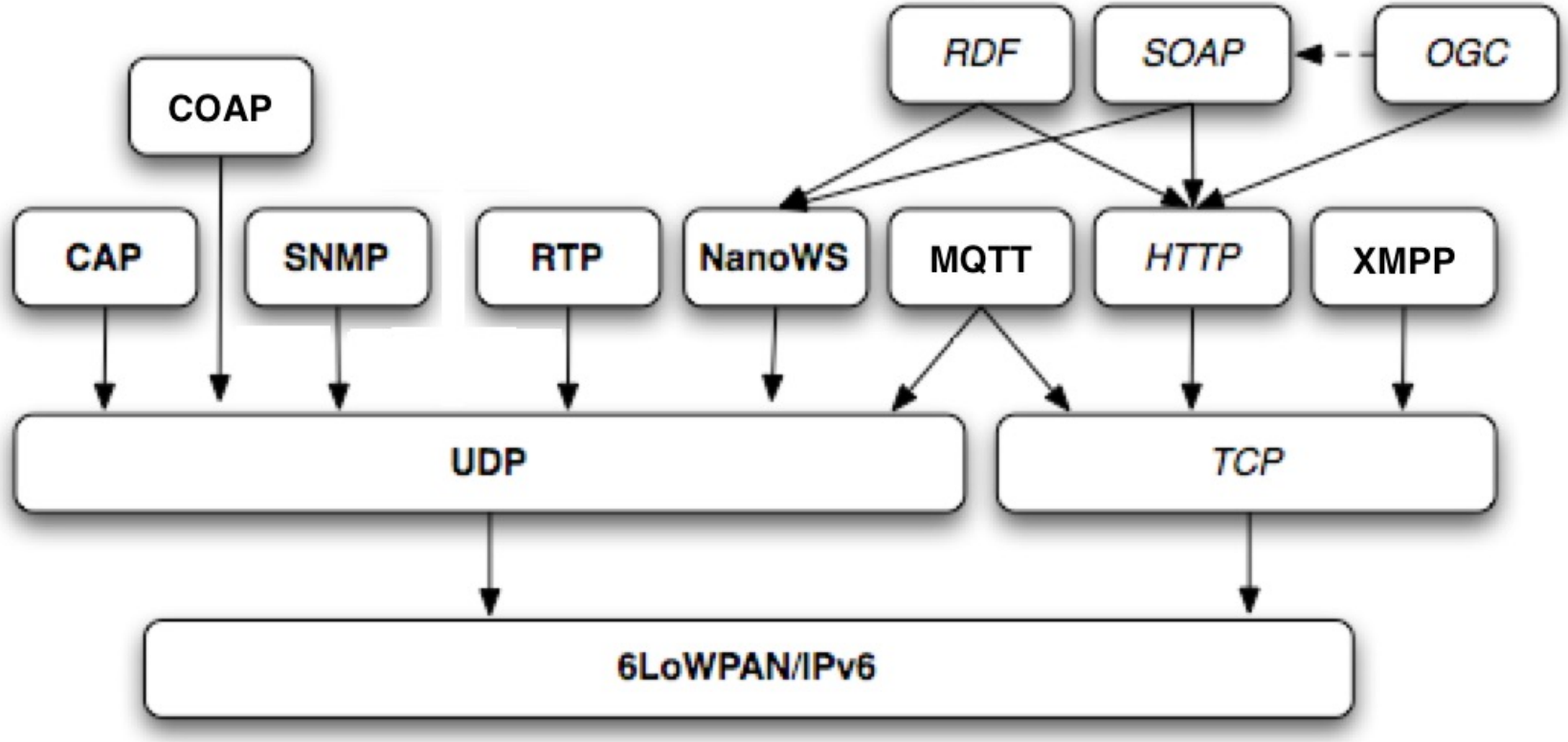


Figure 1: The publish/subscribe communication model

Decoupling Abilities of Interaction Paradigms

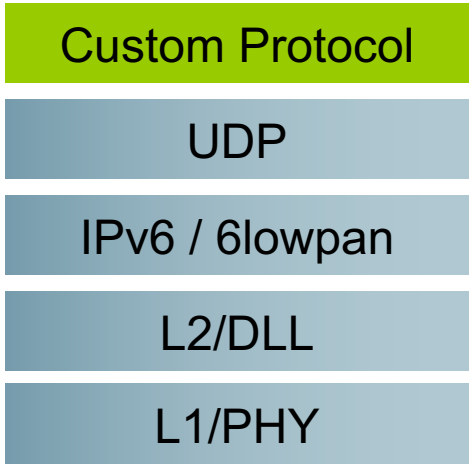
Abstraction	Space decoupling	Time decoupling	Synchronization decoupling
Message Passing	No	No	Producer-side
RPC/RMI	No	No	Producer-side
Asynchronous RPC/RMI	No	No	Yes
Future RPC/RMI	No	No	Yes
Tuple Spaces	Yes	Yes	Producer-side
Message queuing	Yes	Yes	Producer-side
Pub/Sub	**Yes**	**Yes**	**Yes**

Application Formats and Protocols



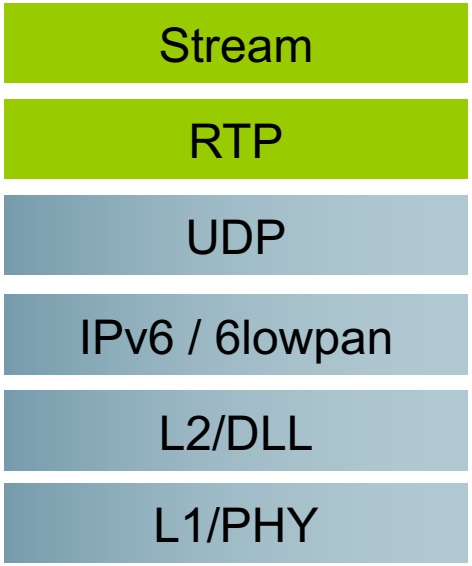
Custom Protocols

- The most common solution today
- Application data typically are binary encoded, often application specific
- Application protocols use UDP
- As 6LoWPAN allows end-to-end IPv6 communications, this is not a problem
- Advantage:
  - Compact, efficient, security can be integrated, end-to-end
- Disadvantage:
  - Custom server app needed, little re-use, learning curve, interoperability



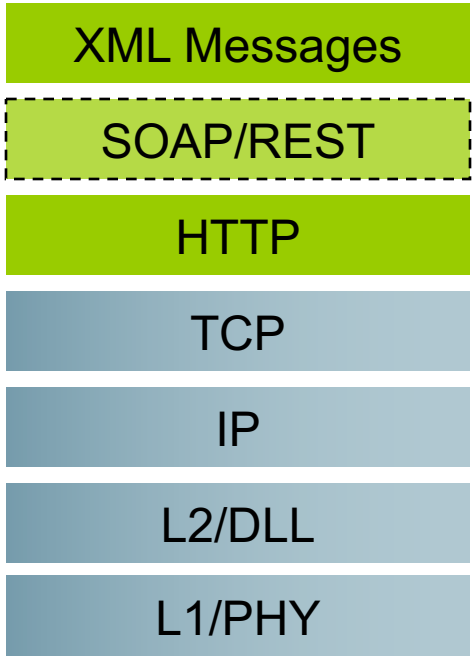
Streaming and RTP

- The correct streaming solution
- For audio or continuous sensor streaming
  - Audio over 802.15.4 needs good codec
- Advantages:
  - RTP can be used over 6LoWPAN
  - Provides end-to-end solution
  - No server modifications needed
  - Jitter control
- Disadvantages:
  - Headers could be more efficient for simple sensor data streaming



XML/HTTP

- De-facto for inter-server communications
- Well-known XML schema important
- All Internet servers speak HTTP/XML
- Useable for RPC, pub/sub and events
- SOAP or REST paradigm
- Advantages:
  - Well known XML schema
  - Formal message sequences
  - Internet-wide support
- Disadvantages:
  - Inefficient, complex
- Solution: Embedded web-services
  - See the IETF 6lowapp effort http://6lowapp.net



SOAP or REST requests (1)

- SOAP stands for Simple Object Access Protocol, a standards-based web services access protocol using XML to provide messaging services on standard protocols such as HTTP or SMTP

```
<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
 <soap12:Body>
 <GetTime xmlns="http://timecurrentime88.com/WebServices">
 <Request>EST</Request>
 </GetTime>
 </soap12:Body>
</soap12:Envelope>
```

- SOAP is complex and provides built-in error handling
- It requires a sophisticated parser on the target
- If there’s a problem with the request, a response with information about the error is sent

SOAP or REST requests (2)

- Representational State Transfer (REST) is an architectural style defining a set of constraints to be used for creating **stateless** web services termed RESTful web services
- It results in a network of Web resources identified by URIs where the user can select links, such as /water/temp, and operations such as GET, PUT, POST, DELETE

```
GET  /basement/water/temp      200 OK   application/text 40.5 F
GET  /basement/water/volume    200 OK   application/text 200 G
```

- HTTP associated with REST forms the REST HTTP protocol
- JSON povides a lightweight interface toward clients

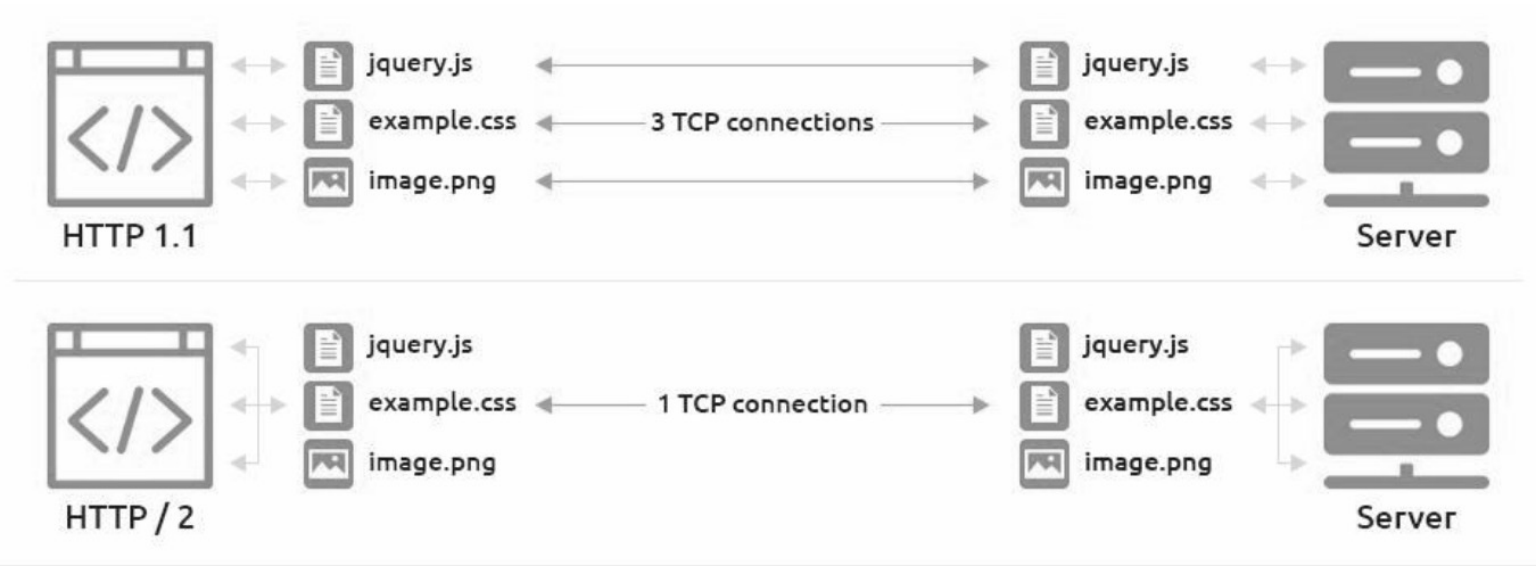
Using JSON for a Thing Description

```
GET http://myserver.com/things/pi
Accept: application/json

200 OK
{
"id": "https://myserver.com/things/pi",
"title": "WoT Pi",
"description": "A WoT-connected Pi",
"properties": {
    "temperature": {
    "title": "Temperature",
    "type": "number",
    "unit": "degree celsius",
    "readOnly": true,
    "description": ”A temperature sensor",
    "links":
       [{"href": "/things/pi/properties/temp"}]
    },
"actions": {
    "reboot": {
      "title": "Reboot",
      "description": "Reboot the device"
    }
  },
"events": {
    "reboot": {
      "description": "Going down for reboot"
    }
  },
"links": [
  {
    "rel": "properties",
    "href": "/things/pi/properties"
  },
  {
    "rel": "actions",
    "href": "/things/pi/actions"
  },
  {
    "rel": "events",
    "href": "/things/pi/events"
  },
  {
    "rel": "alternate",
    "href": "wss://myserver.com/things/pi"
  },
  {
    "rel": "alternate",
    "mediaType": "text/html",
    "href": "/things/pi"
  }
]
}
```

SOAP or REST requests (2)

- The IoT device acts as a Web Client.
- Since HTTP is a challenge-response protocol device will have to keep polling the server for new updates
- Three different versions:
  - HTTP 1.0 one TCP connection for each pair of request/response message;
  - HTTP 1.1 a TCP connection could be reused;
  - HTTP/2 multiplexing method, single TCP connection for multiple request/response messages.



REST is better for the IoT

- REST is better suited for lightweight IoT applications involving mobile and embedded devices being the most straightforward and simple way of achieving a global network of smart things. It:
  - Uses smaller message formats
  - Runs quickly because no extensive processing required
  - Looks similar to other web technologies in design philosophy
  - Proves low-cost in how it interacts with the web service

- REST acts as light-weight alternative and uses a simple URL in many cases instead of XML for requests.
- REST permits many different data formats, which by outward appearances adds complexity.
- Due to support for JSON, this allows REST to better support browser clients as JSON is a better fit for data and parses faster

REST disadvantages

- REST is HTTP-based
  - high power consumption
  - High complexity
  - not perfectly suited for constrained devices;
- TCP is used as transport protocol and for QoS
  - large overhead
  - not suited for push notification;
- TLS is used as security mechanism
  - The handshake process adds additional traffic

WebSocket Protocol

- HTTP is half-duplex and has too much overhead
- Websocket is a new protocol providing bi-directional, full duplex lightweight connections.
- It can reduce the most unnecessary data with the header to prevent wasting. Better for sending small packets

- Extension of HTTP(S) Protocols
  - Two way communication
  - Persistent connections
  - Layered over TCP or SSL/TLS connection

- Protocol Properties
  - Defined in detail in RFC 6455
  - Handshake is compatible with HTTP handshake
  - Simple message-based communication
    - User can specify custom sub-protocols (i.e., the contents and semantics of the messages)

Difference between REST/HTTP and WebSocket-based Communication

Comparison Based on	REST	Websocket
State	Stateless	Stateful
Direction	Unidirectional	Bidirectional
Req-Res/Full Duplex	Follow Request Response Model	Exclusive Pair Model
TCP Connections	Each HTTP request involves setting up a new TCP Connection	Involves a single TCP Connection for all requests
Header Overhead	Each request carries HTTP Headers, hence not suitable for real-time	Does not involve overhead of headers.
Scalability	Both horizontal and vertical are easier	Only Vertical is easier

WebSocket Protocol

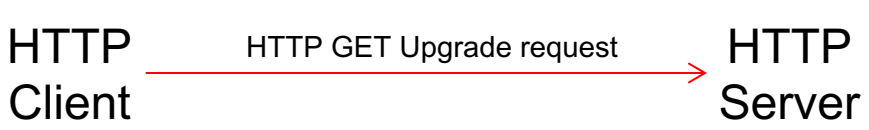
- Handshake
  - The connection is initiated by a HTTP request, which is a HTTP-upgrade request

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

The "upgrade" request

Security things

WebSocket specific information



WebSocket Protocol

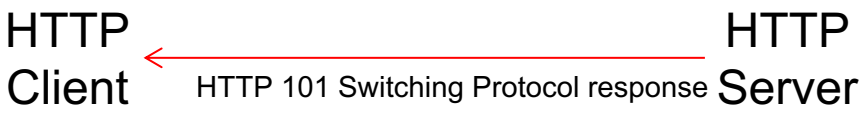
- Handshake
  - The server may respond by any HTTP-valid code
    - E.g. 404 if the URI is not valid or by 3xx redirection
    - It may also include things like auth-tokens, cookies…
  - Upgrade acceptation looks as follows

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGz...
Sec-WebSocket-Protocol: chat
```

Upgrade confirmation

Security verification

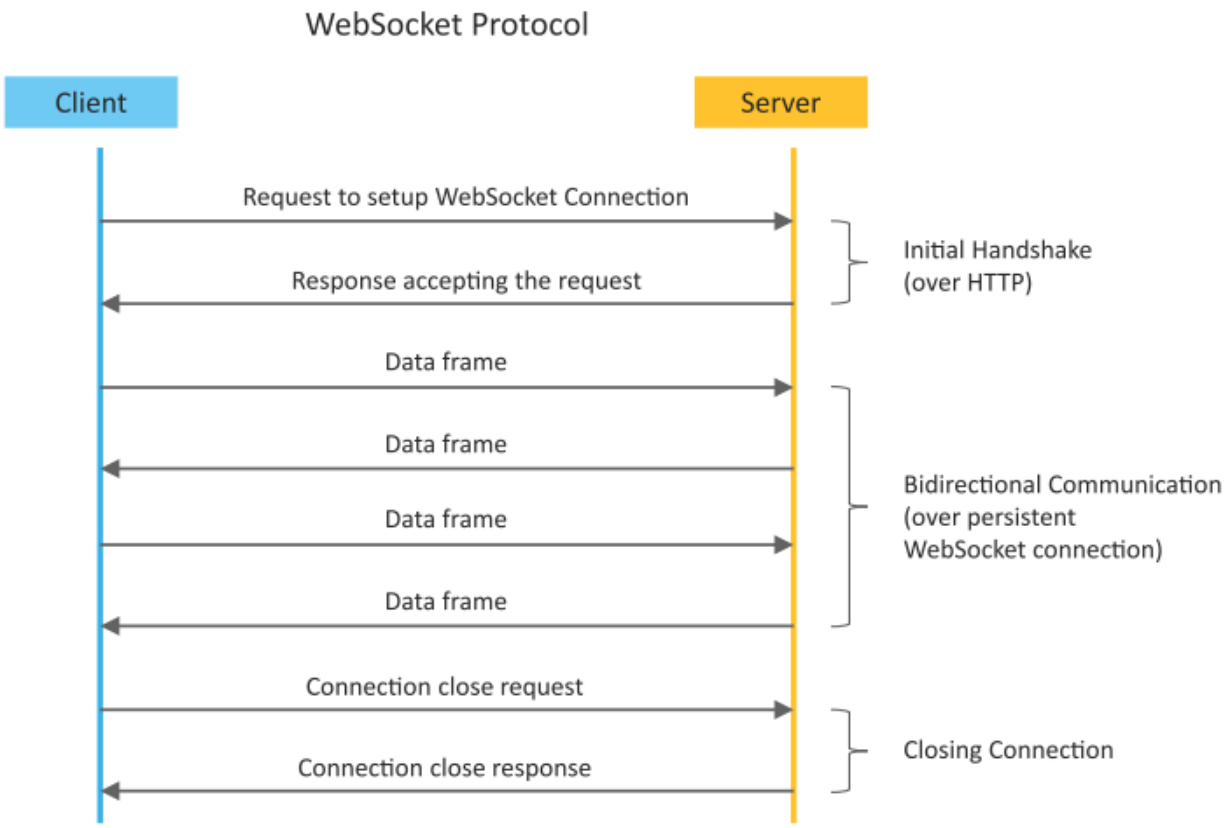
Accepted subprotocol



WebSocket Protocol

- Close Handshake
  - Simply a WebSocket frame with opcode 0x8
  - Not uncommon for WebSockets to close abruptly due to disconnection

- Data Transfer
  - Bidirectional communication using data frames

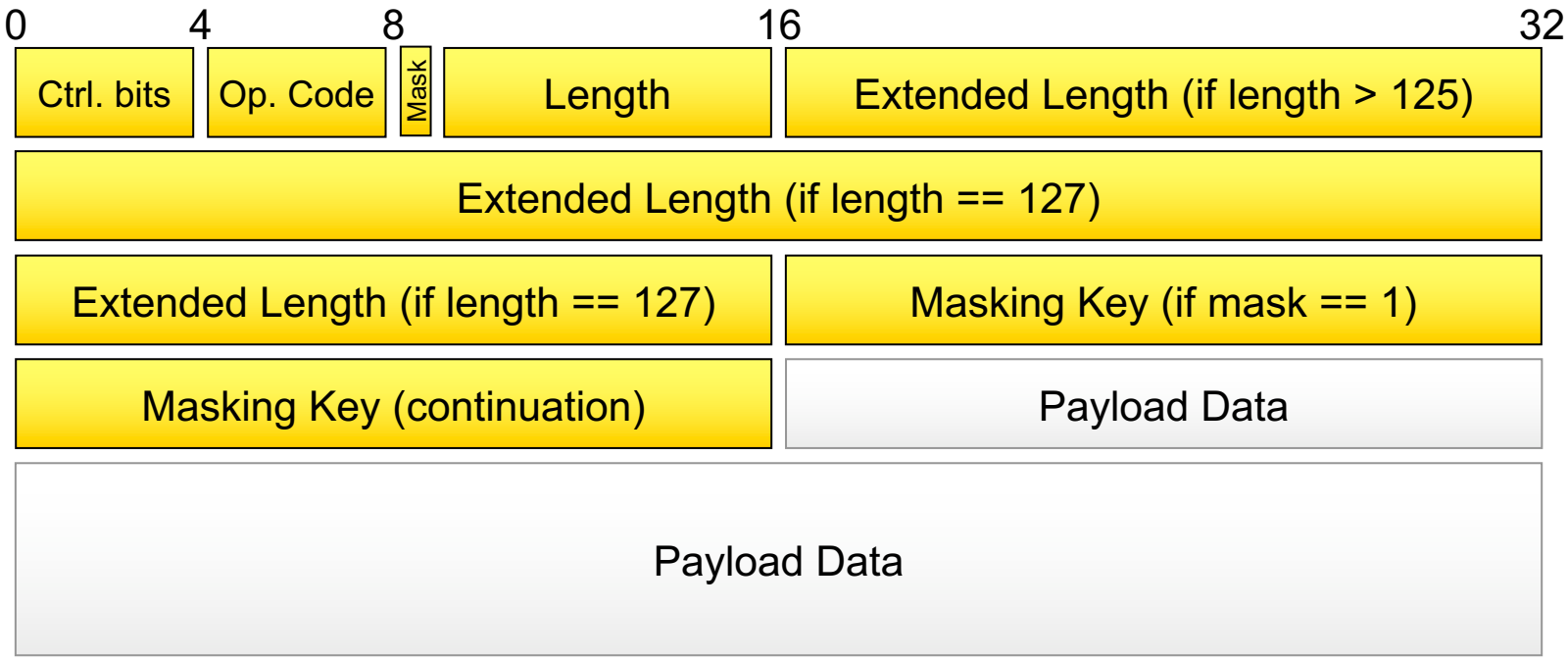


WebSocket Protocol

- Data Frames
  - Messages may be fragmented
    - First frame defines the type, other frames are defined as continuation frames
    - Last frame has the appropriate control bit set
  - Text messages must be in UTF8, binary messages are interpreted only by the application
- Control Frames
  - Close frame initiates graceful connection shutdown
  - Ping/pong frames verify the connection is still alive

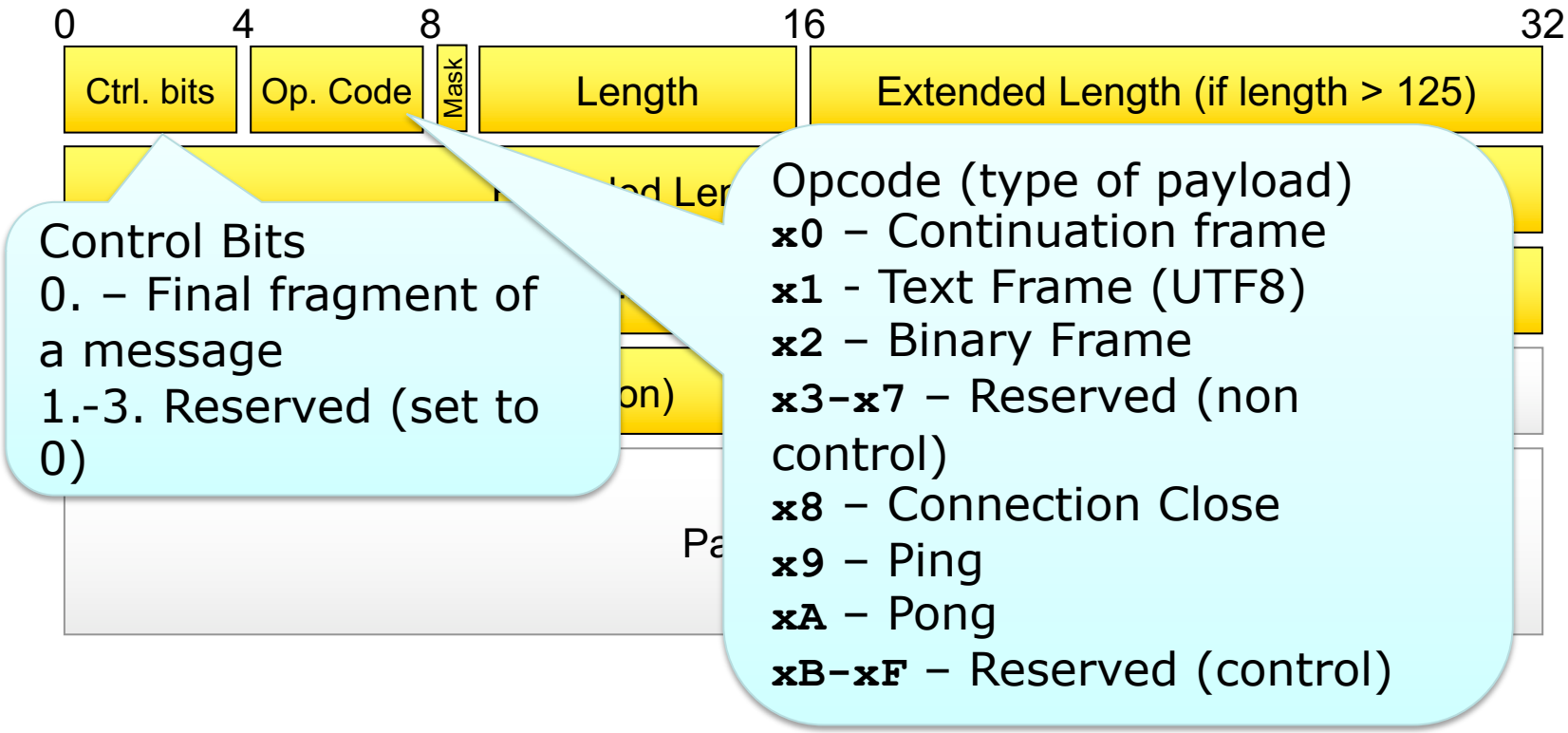
WebSocket Protocol

– Frame Format



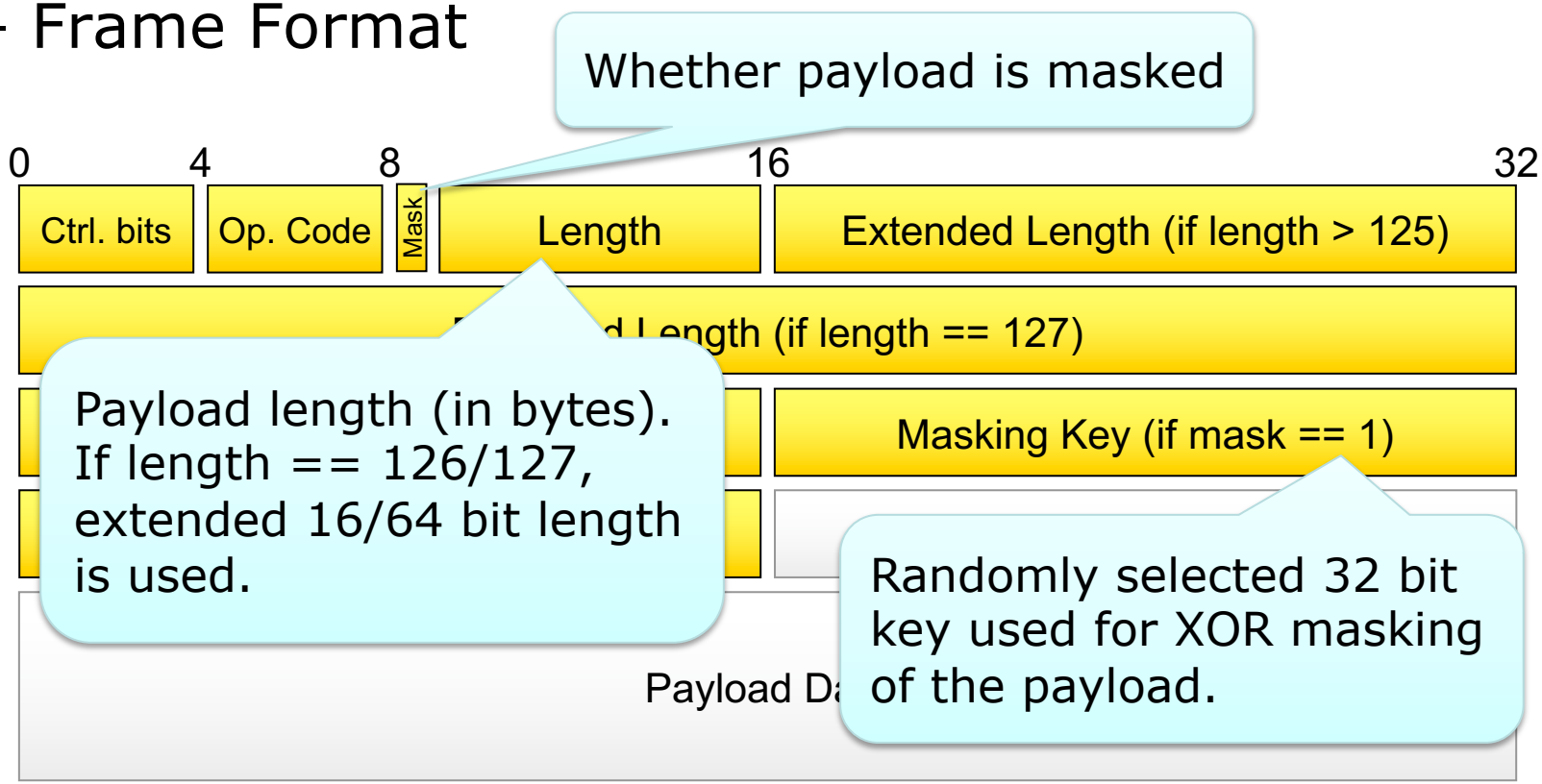
WebSocket Protocol

– Frame Format



WebSocket Protocol

– Frame Format

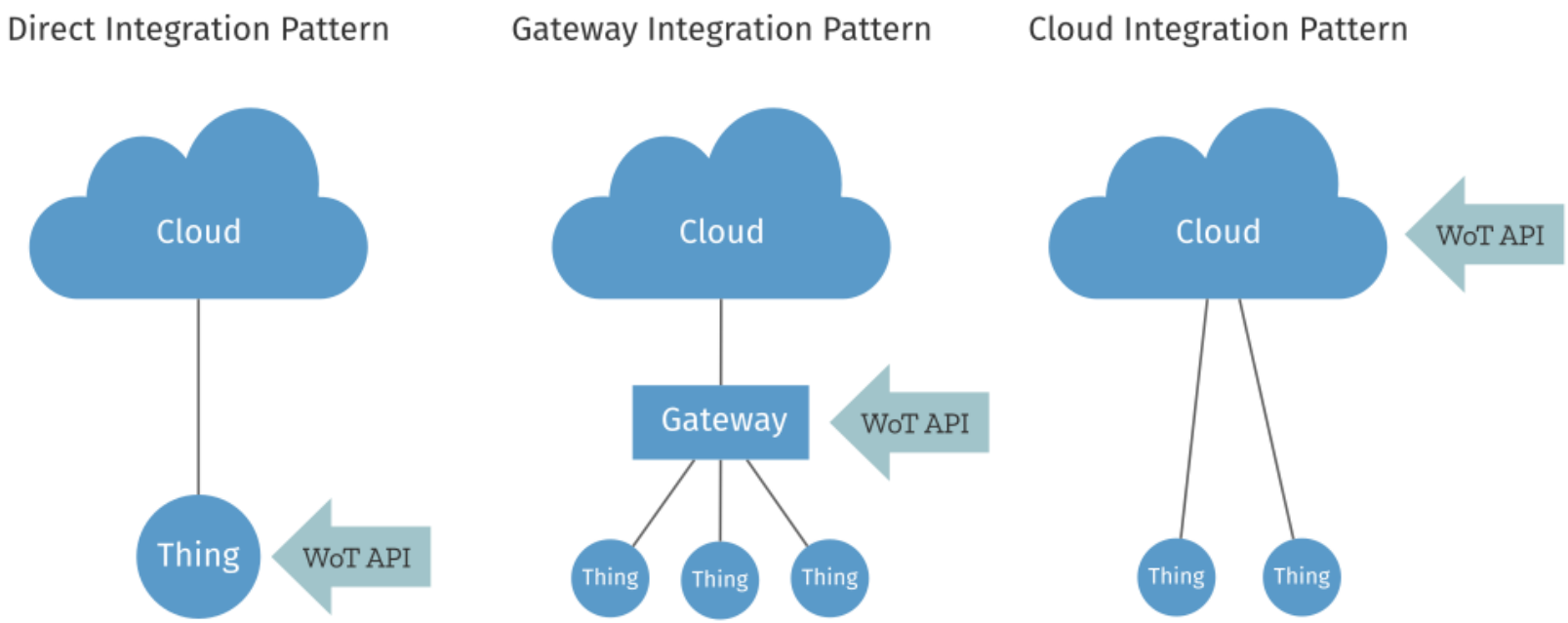


WebSocket Protocol

- Security
  - The connection may be established on a SSL/TLS channel instead of TCP
    - In the same way HTTP and HTTPS works
  - The client sends an origin that refers to a domain from which the client script was downloaded
  - The server must correctly transform given security key to verify it is really a WebSocket server
  - The client may use frame masking to defend against a specific cache-poisoning attack
    - When the attacker constructs frames that look like a HTTP request

The Web of Things (WoT) Integration

- Direct integration Pattern
- Gateway Integration Pattern
- Cloud Integration Pattern



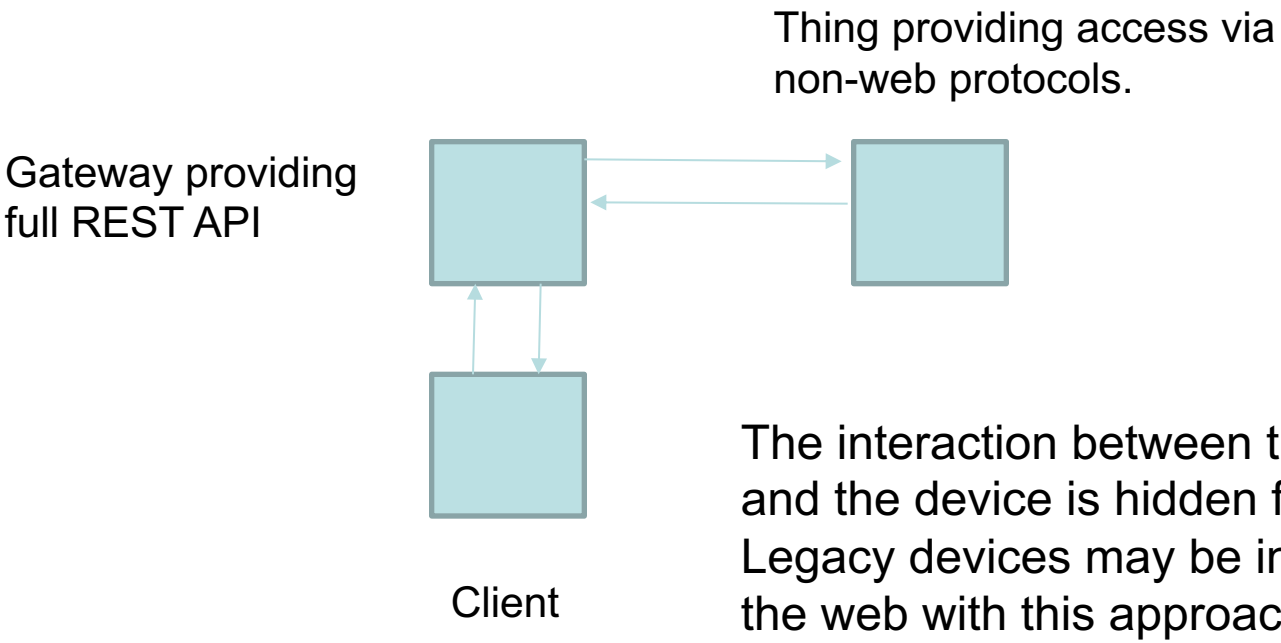
Direct Integration Pattern

- Some Things have full internet access.
- They provide an HTTP server running over TCP/IP and can directly connect to the internet via WiFi, Ethernet or cellular
- These may be used to implement a Direct Integration Pattern via REST on devices.

Example: Use a phone to communicate via WiFi (with WiFi router) to an HTTP server on a device.

Gateway Integration Pattern

- Some Things may not have full internet access.
- These Things may support only Bluetooth or 802.15.4.
- Suppose we cannot send IP packets to the device – it is constrained. This is the Gateway Integration Pattern.

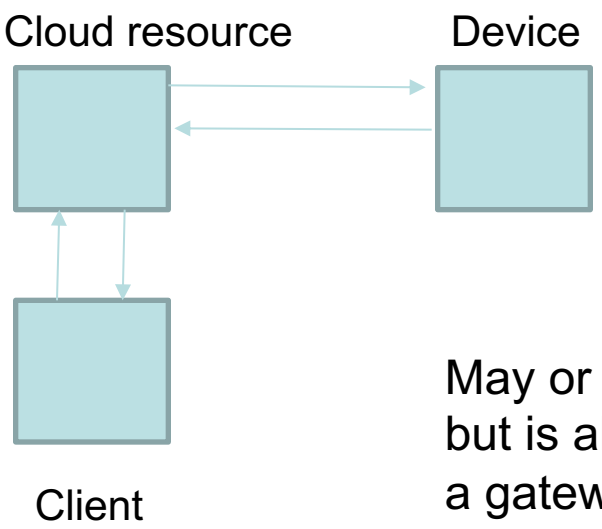


The interaction between the gateway and the device is hidden from the client. Legacy devices may be integrated onto the web with this approach.

Cloud Integration Pattern

- Some Things have access to the cloud and need powerful and scalable cloud support.
- This is the Cloud Integration Pattern.
- The cloud can provide REST API or publish/subscribe using web hooks or WAMP.

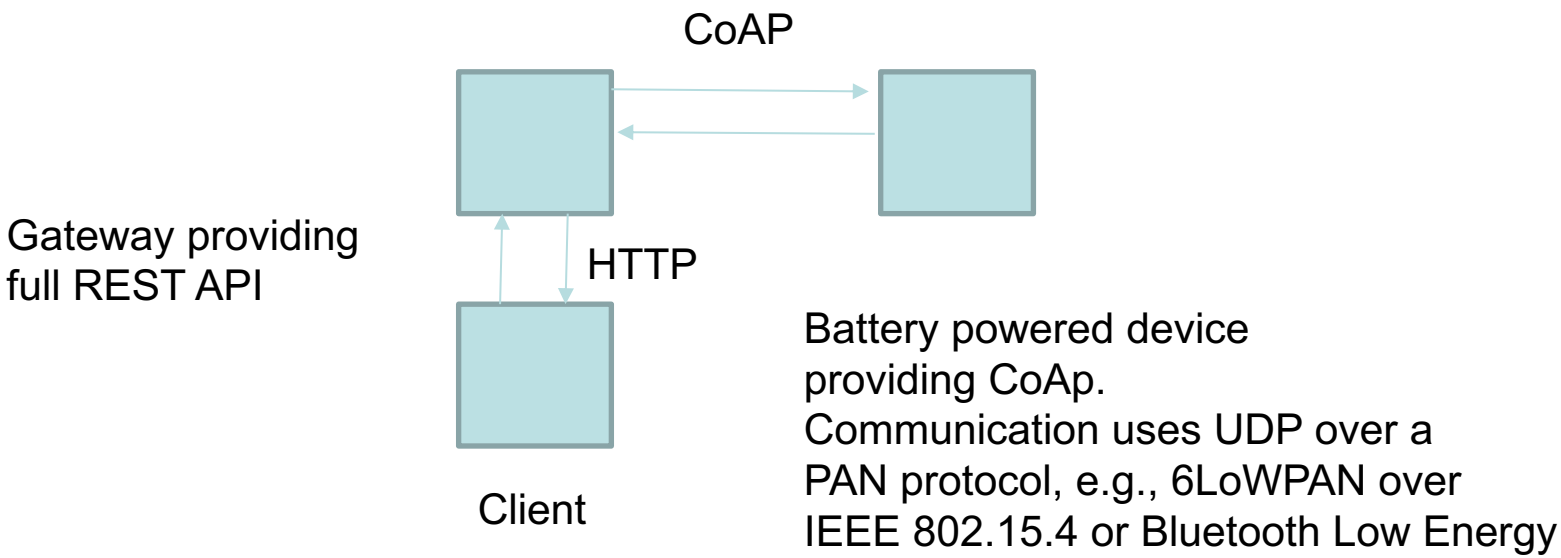
Cloud Gateway providing full REST API or pubsub using web hooks



May or may not use web protocols but is able to communicate with a gateway in the cloud.

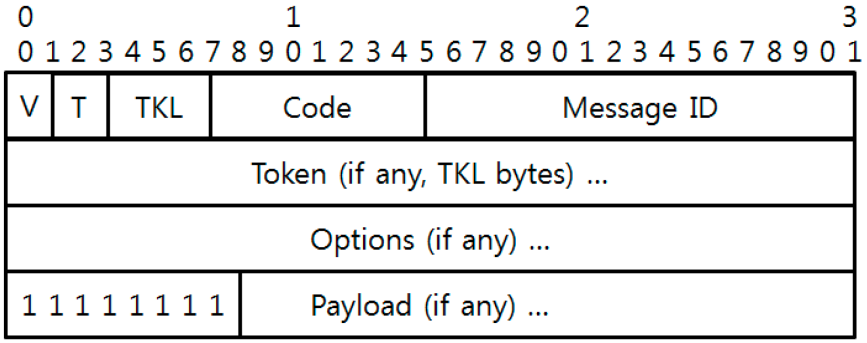
CoAP is based on REST

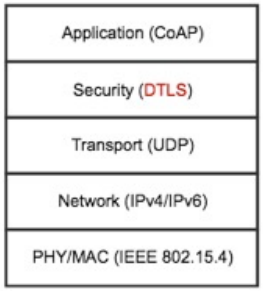
- CoAP provides a request/response RESTful interaction like HTTP.
- Smaller messages than HTTP and with very low overhead.



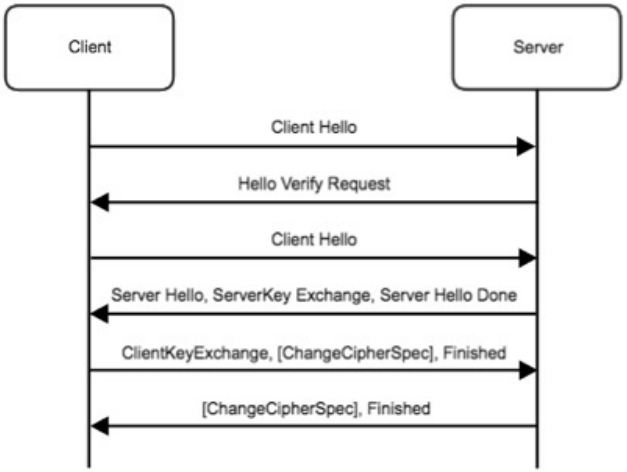
Constrained Application Protocol (CoAP)

- CoAP is a REST-based transfer protocol for power and CPU constrained nodes and low bandwidth, lossy networks
- CoAP uses the REST architecture but not HTTP
  - REST make information available as resources identified by URIs.
  - Sensors and actuators are simply CoAP REST resources
  - Applications communicate by exchanging representation of these resources using a transfer protocol similar to HTTP but lighter
  - CoAP uses UDP instead of TCP and has a simpler "message layer"
  - It also uses compression techniques and DTLS security (optional)
  - Can run over 6lowpan





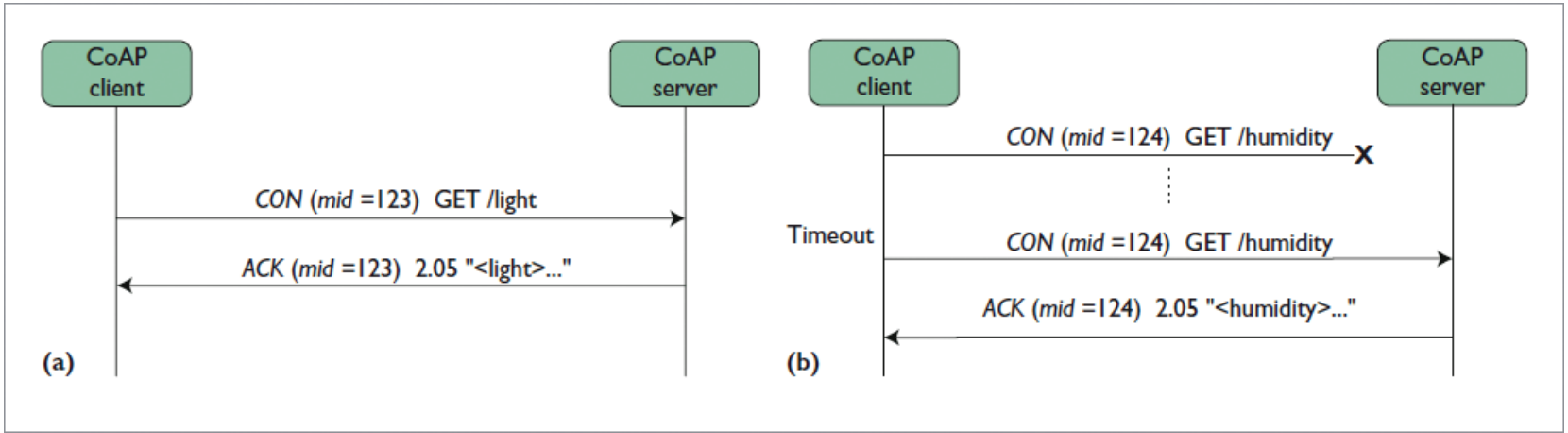
DTLS in Protocol Stack



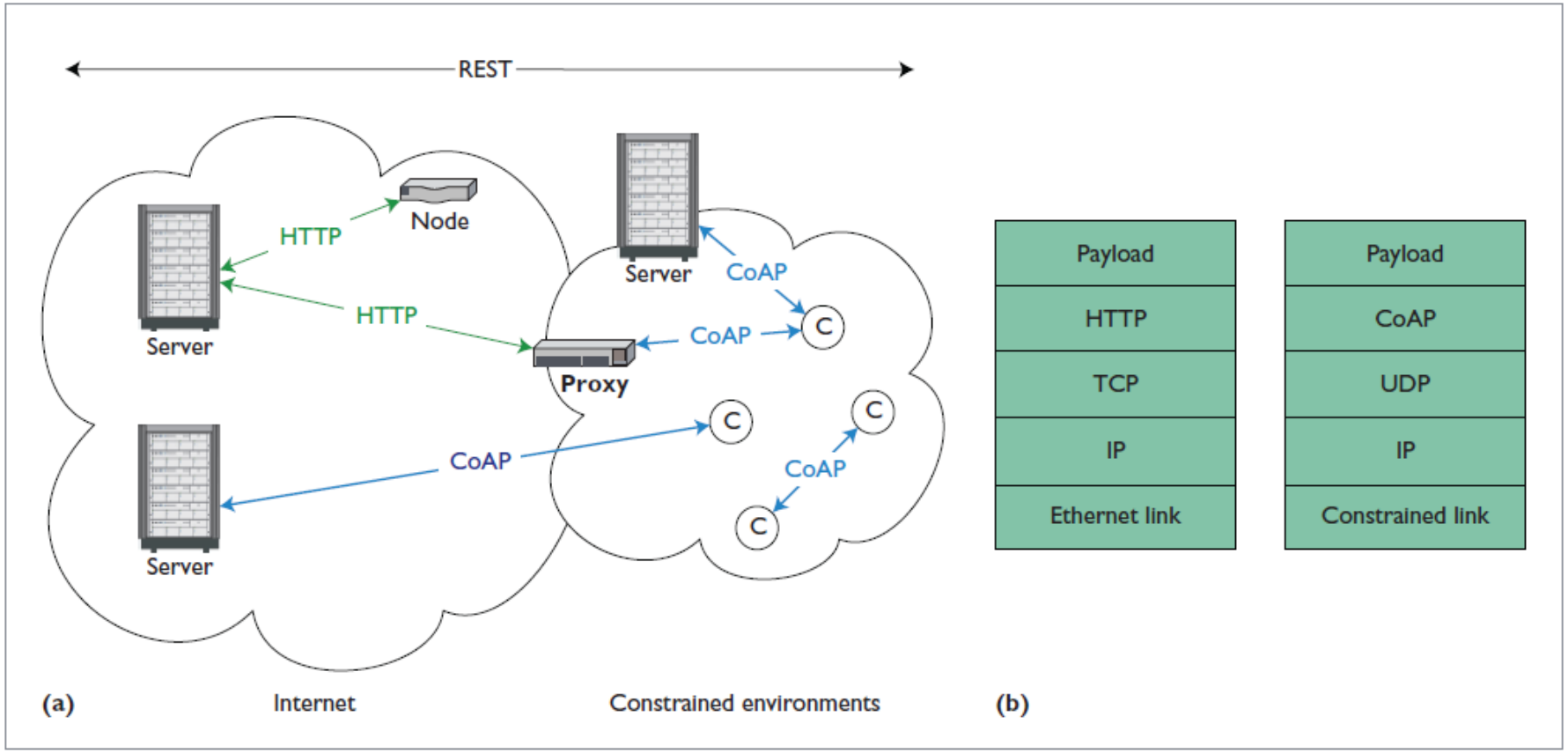
DTLS Process of Handshake

Constrained Application Protocol (CoAP)

- Clients access service-controlled resources using synchronous request/response mechanisms with 4 methods:
  - GET
  - PUT
  - POST
  - DELETE
- Responses are matched with requests via a client-generated Token (mid).
- Implements an ARQ mechanism based on timeouts for re-transmitting lost packets.
  - Confirmable messages requires an ACK with message ID. The message ID of the ACK matches the message ID of the confirmable message.
  - Non-confirmable messages do not require an ACK. Less reliable.



CoAP protocol stack and interactions



Implementing the Web architecture with HTTP and the Constrained Application Protocol (CoAP). (a) HTTP and CoAP work together across constrained and traditional Internet environments; (b) the CoAP protocol stack is similar to, but less complex than, the HTTP protocol stack.

C. Bormann, A. P. Castellani, Z. Shelby, "CoAP: An Application Protocol for Billions of Tiny Internet Nodes," IEEE Internet Computing, vol. 16, no. 2, pp. 62-67, Feb. 2012, doi:10.1109/MIC.2012.29

CoAP Timeouts over UDP

```
CoAP Client                                  CoAP Server
                              ---->
CON  {id} GET  /basement/light               lost request
                                             timeout
                             ---->
CON  {id} GET /basement/light                finally arrives
                             <----
ACK  {id} 200 Content {"status" : "on"}
```

The {mid} associates messages with acks and allows duplicates detection. What happens if the ACK is also lost?

CoAP Request/Acknowledge/Callback

CoAP Client CoAP Server

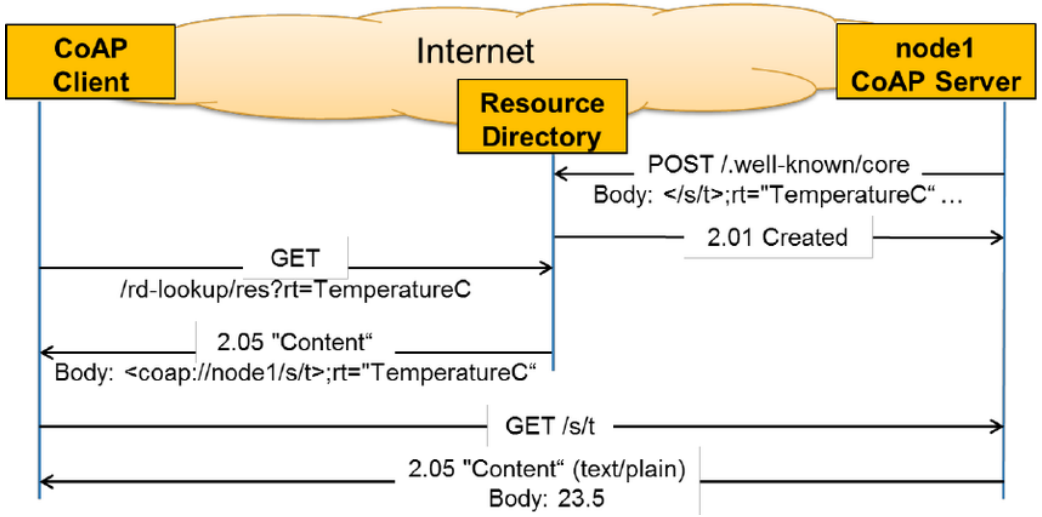
```
                        ---->
CON  {id} PUT  /basement/cleanFloor   Token:  0x22   Needs time
                        <----
ACK  {id}       I am on it!
                        <-----

CON {newID} 200 Content /basement/cleanFloor Token: 0x22 Done
                        ---->
ACK {newID}
```

In this example, the same token is used to identify this request and the service response. The id’s are used at the message level.

CoAP Resource Discovery

- With **resource discovery**, we know we are looking for web resources
  - We might register a printer, for example, with a discovery service and find it later on the fly.
- A well known resource is used to discover other resources
- Resource directories may be used to register resources.
  - Links are returned, including REL attributes – providing semantics
  - Perform a GET on the well known resource. Returned content is a list of links with REL attributes.
  - Registrations are simply POSTs with links.
  - PUTs are used for updates.
  - GETs for discovery.



CoAP Resource Discovery

CoAp Client CoAp Server

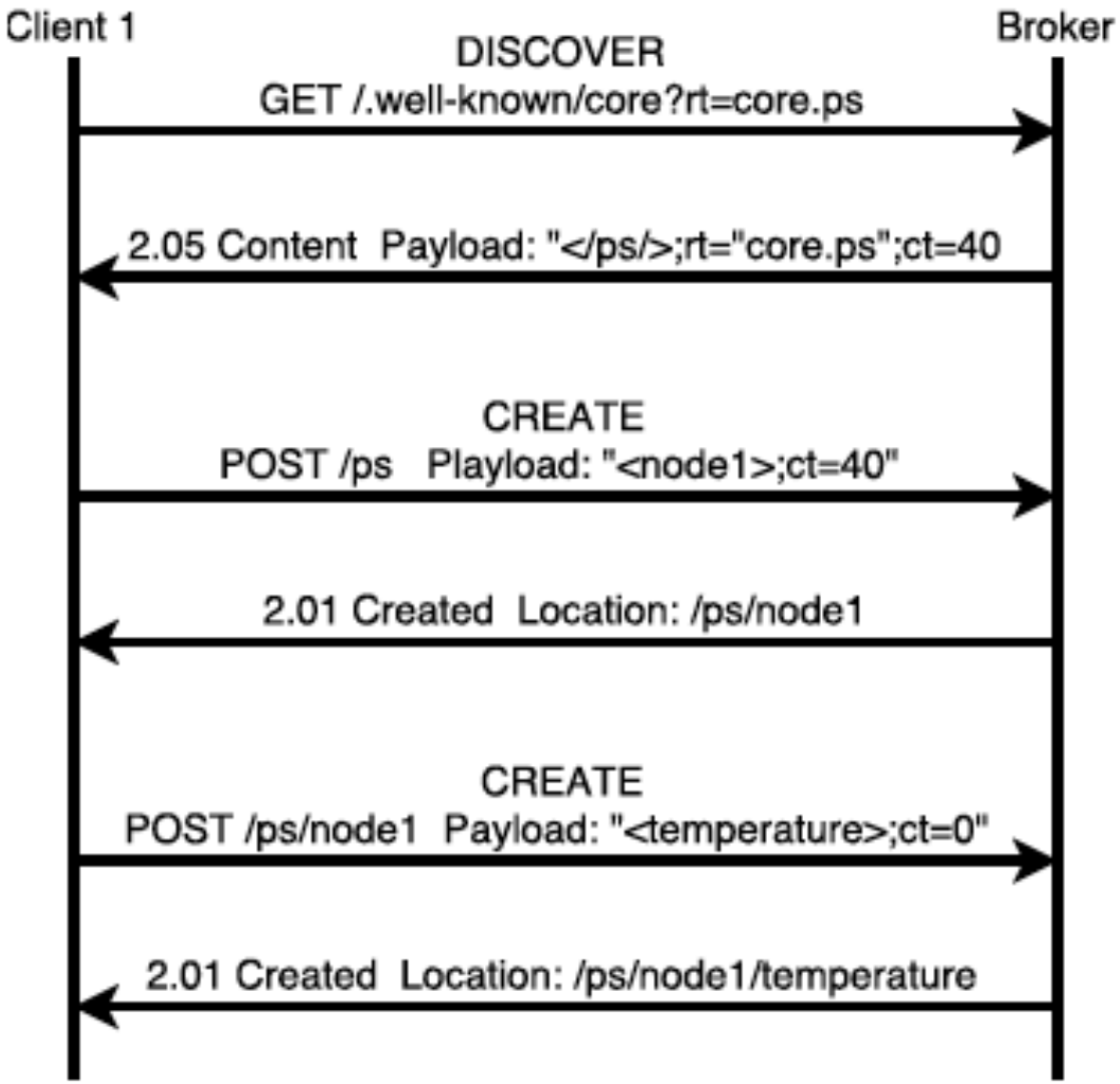
```
                        ---->
CON  {id} GET  /.well-known/core   Token:  0x22
                        <----
ACK  200 {id}  Content "/sensor/temp /sensor/light"
                        ---->
CON  {id} GET  /sensor/light
                        <----
ACK  205 {id}  Content "dim"
                        ---->
CON  {id} GET  /sensor/temp
                        <----
ACK  205 {id}  Content "72"
```

CoAP Publish/Subscribe

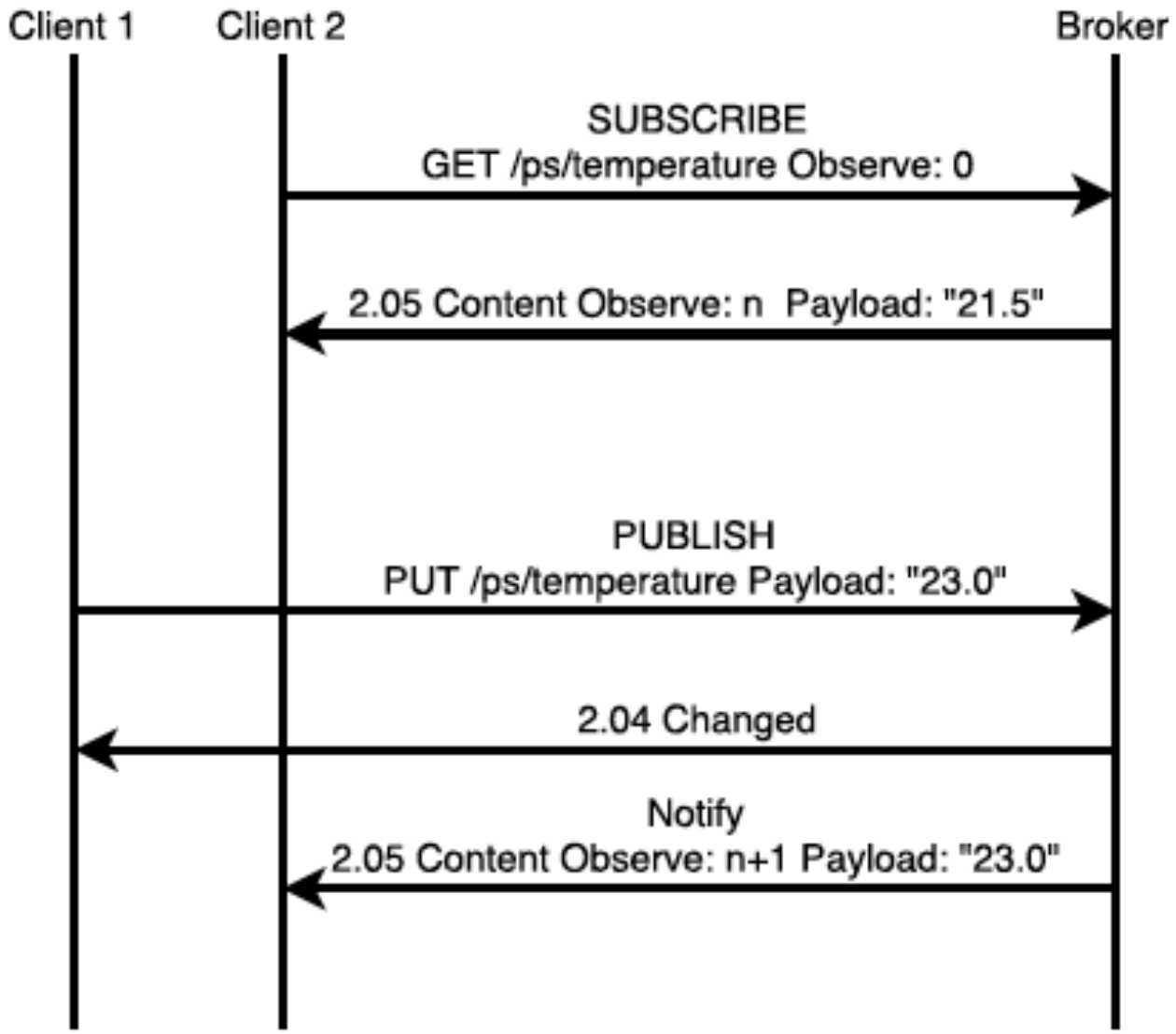
- The GET method includes an “Observe” message to establish a subscription request.
- The response includes an “Observe” to say this is a publication.
- The value included with Observe response is there for possible re-orderings.
- The client should take the most recent sent and not the most recent to arrive.

```
CoAP Client                                        CoAP Server
                              ---->
CON  {id} GET  /basement/light Observe: 0   Token:  0x22
                              <----
ACK  200 {id}  Observe: 27 Token 0x22
                              <----
CON 200 Observe: 28 Token: 0x22 {“light” : ”off”}
                              ----->
ACK Token: 0x22
                               <----
CON 200 Observe: 30 Token: 0x22 {“light” : ”on”}
```

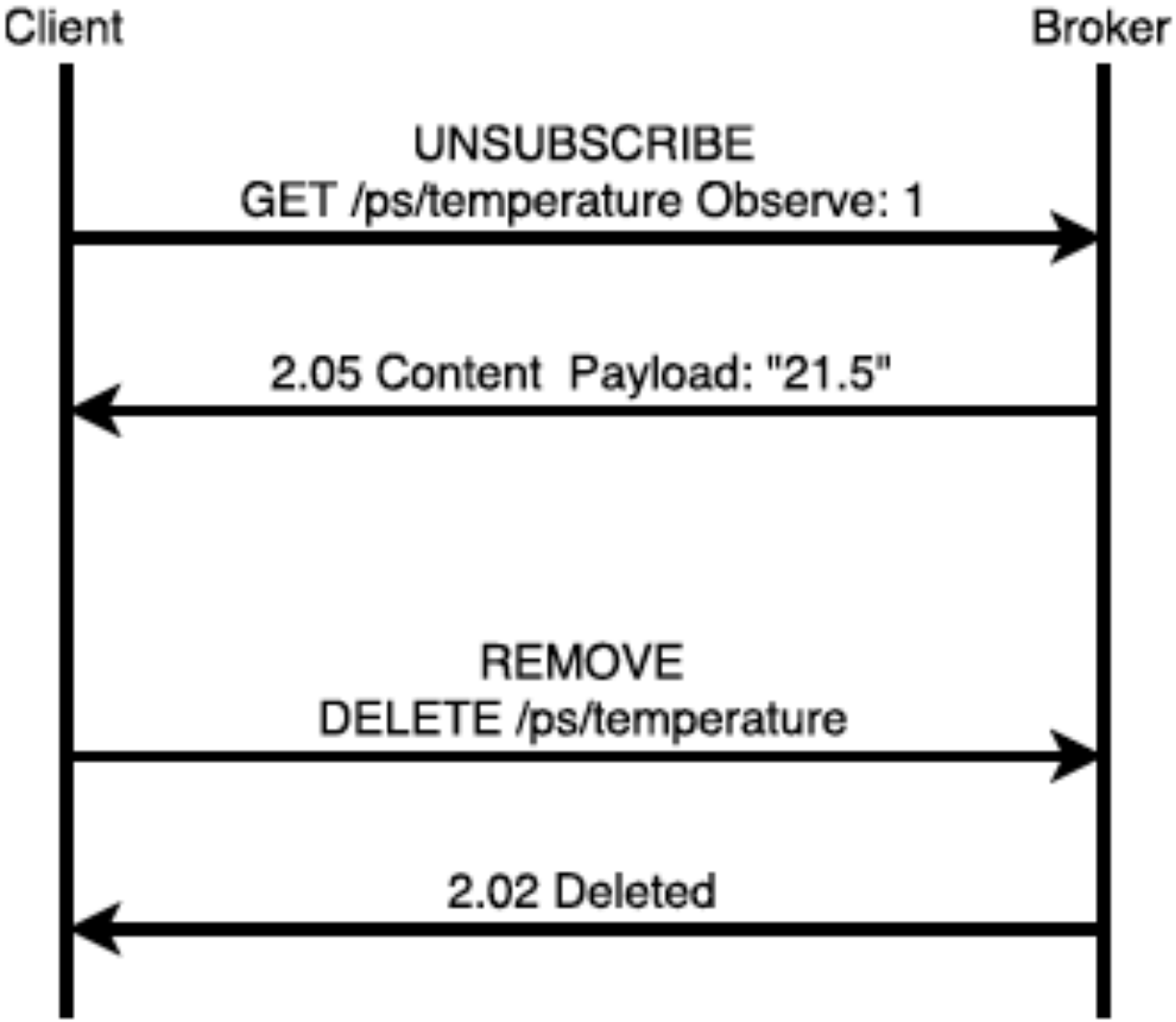
CoAP Publish/Subscribe: create a new topic



CoAP Publish/Subscribe: subscription and publication



CoAP Publish/Subscribe: unsubscription



Simple suggestions to building the WoT

1. A Smart Thing is a digitally enhanced object. A Connected Thing includes a network.
2. Identify resources with Uniform Resource Identifiers. Actuators, sensors, tabletops, rooms, Smart Things, etc. – all get a URI. These are the nouns.
3. Use a constrained interface and exploit polymorphism. Use REST/HTTP or CoAP. PUT, GET, etc. are the verbs.
4. Agree upon resource representation formats. Use JSON or JSON-LD (JSON with semantics). On the web, the wide adoption of HTML allows clients and servers to cooperate without individual agreements.
5. Use well known addresses and links for discovery – perhaps exploiting JSON-LD.
6. Provide stateless interactions. Each request should contain all that the server needs to satisfy the request.

From “A Resource Oriented Architecture for the Web of Things”, Guinard, Trifa, and Wilde

MQTT Pub/Sub Protocol

- MQ Telemetry Transport (MQTT) is a lightweight broker-based publish/subscribe messaging protocol.
- MQTT is designed to be open, simple, lightweight and easy to implement.
  - These characteristics make MQTT ideal for use in constrained environments, for example in IoT.
    - Where the network is expensive, has low bandwidth or is unreliable
    - When run on an embedded device with limited processor or memory resources;
- A small transport overhead (the fixed-length header is just 2 bytes), and protocol exchanges are minimized to reduce network traffic
- MQTT was developed by Andy Stanford-Clark of IBM, and Arlen Nipper of Cirrus Link Solutions.

Source: MQTT V3.1 Protocol Specification, IBM, http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html

MQTT Pub/Sub Protocol

- MQTT works over TCP or TLS
- MQTT–S uses UDP rather than TCP
- Built for unreliable networks
- Decouples readers and writers
- Messaging transport is agnostic to the content of the payload
  - MQTT is a binary protocol where the control structure and payload are binary element , not text strings as in HTTP-based protocols

MQTT vs MQTT-S

	MQTT	MQTT-S
Transport type	Reliable point to point streams	Unreliable datagrams
Communication	TCP/IP	Non-IP or UDP
Networking	Ethernet, WiFi, 3G	ZigBee, Bluetooth, RF
Min message size	2 bytes - PING	1 byte
Max message size	≤ 24MB	< 128 bytes (*)
Battery-operated		√
Sleeping clients		√
QoS: -1 “dumb client”		√
Gateway auto-discovery & fallbacks		√

ZADATA © 2013

MQTT Message Format

- Each MQTT command message contains a fixed header.
- Some messages also require a variable header and a payload.
- You can send anything as a message. Up to 256 MB.
- Message have a topic, QoS and retain status associated Hierarchical topic namespace and subscriptions with wildcards

-

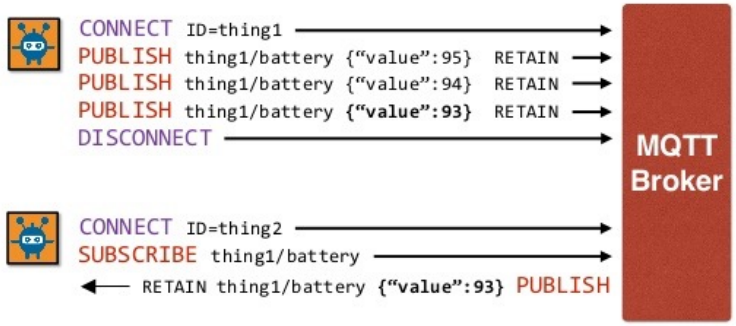
bit	7	6	5	4	3	2	1	0
byte 1	Message Type				DUP flag	QoS level		RETAIN
byte 2	Remaining Length							

- DUP: Duplicate delivery
- QoS: Quality of Service
- RETAIN: RETAIN flag
  - This flag is only used on PUBLISH messages. When a client sends a PUBLISH to a server, if the Retain flag is set (1), the server should hold on to the message after it has been delivered to the current subscribers.
  - This allows new subscribers to instantly receive data with the retained, or Last Known Good, value.

Source: MQTT V3.1 Protocol Specification, IBM, http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html

MQTT Messages

- As soon as you subscribe you may receive the most recently published message.
- One message per topic may be retained by the broker
- Uses a command and ACK format: a specific ACK message follow many actions (connect, publish, subscribe)
- A newly connected subscriber receives the most recent message.
- Publishing clients may connect with a **last will** and **testament** by using the LWT feature, to manage the actions in case of ungraceful disconnect
- In the case of graceful disconnect, the publisher sends a **disconnect** message to the broker – causing discard the of LWT actions.
- If publishing client has no data to send, it sends **ping requests** to the broker to inform the broker that it is still alive and waits for a **ping response**.




```
# Message types
CONNECT = 0x10
CONNACK = 0x20
PUBLISH = 0x30
PUBACK = 0x40
PUBREC = 0x50
PUBREL = 0x60
PUBCOMP = 0x70
SUBSCRIBE = 0x80
SUBACK = 0x90
UNSUBSCRIBE = 0xA0
UNSUBACK = 0xB0
PINGREQ = 0xC0
PINGRESP = 0xD0
DISCONNECT = 0xE0
```

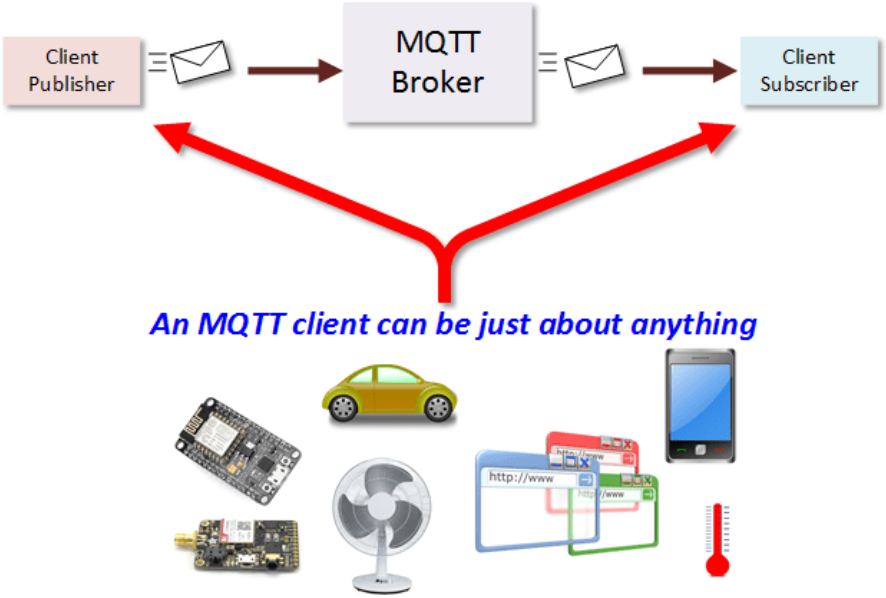
=16 decimal

=128 decimal

=224 decimal

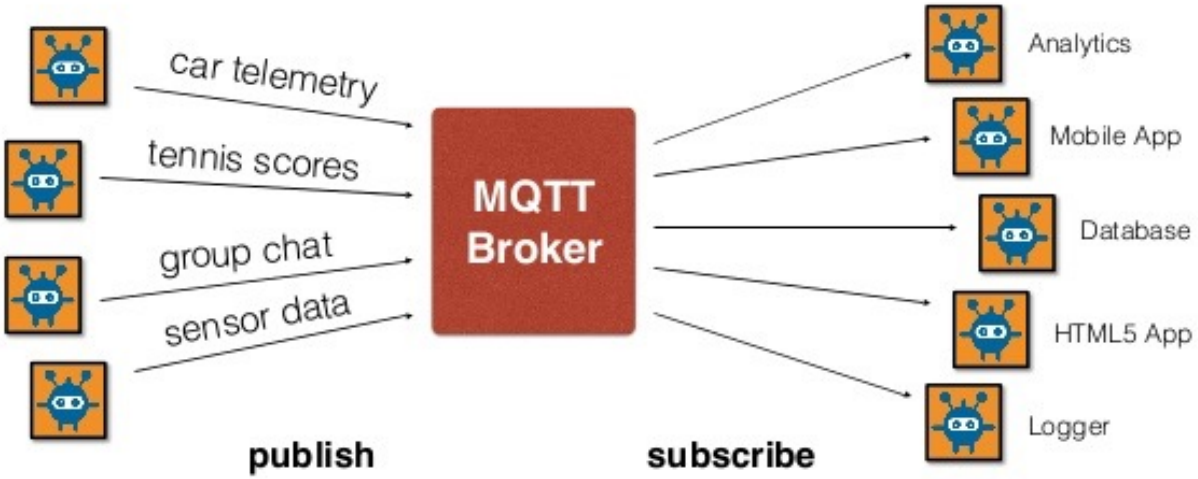
MQTT Client

- A client may publish or subscribe or do both.
- A client is any device from a microcontroller up to a full-fledged server, that has an MQTT library running and is connecting to an MQTT broker over any kind of network.
- A client is any device that has a TCP/IP stack and speaks MQTT.
- Client libraries are widely available (Android, Arduino, iOS, etc.)
- No client is connected directly to any other client but only to a broker



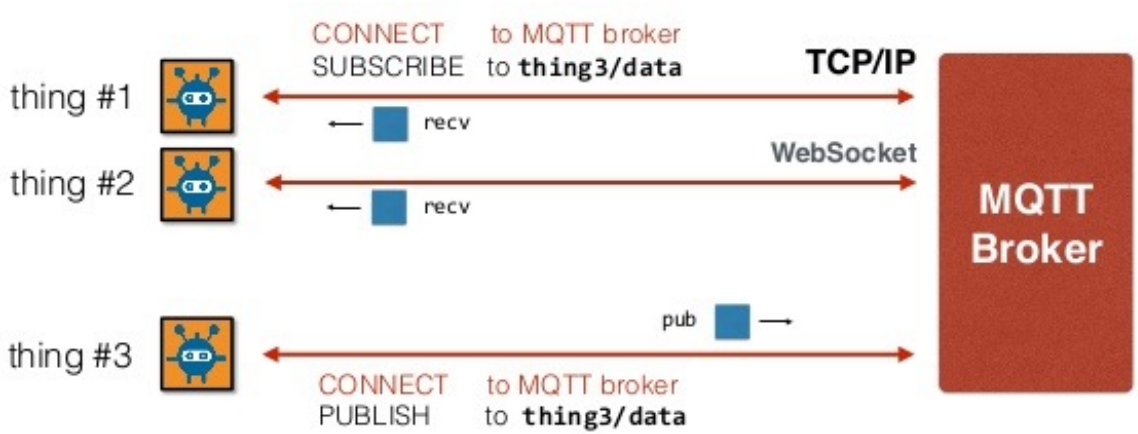
MQTT Broker

- The broker is primarily responsible for receiving all messages, filtering them, decide who is interested in it and then sending the message to all subscribed clients.
  - Authenticate and authorize clients.
  - Maintains sessions and missed messages
  - Maintains a hierarchical namespace for topics and allows subscribers (but not publishers) to use wildcards (+ and #).



MQTT: Sensor readings with pub/sub

- Decoupling is provided in space and time
  - The clients do not need each others' IP address and port (space)
  - They also do not need to be running at the same time (time).
- The broker's IP and port must be known by clients
- Topics are organized into a hierarchical namespace
  - Namespace hierarchy can be used for topic filtering
- It may happen that a published message is never consumed by any subscriber



Topics filtering example

- Suppose a client publishes to myhome/groundfloor/livingroom/temperature
- Another client might subscribe to: myhome/groundfloor/livingroom/temperature

- Or, using a single level wildcard (+) subscribe to all sensors at the grounfloor thet have a child called temperature: myhome/groundfloor/+/temperature

- Or, using a multi level wildcard (#) you can go deep by seeing every message in the myhome tree: myhome/#

MQTT Quality of Service (QoS)

- Defines how the two parties will work to ensure that messages arrive
- QoS levels between the client and a broker:
  - 2: exactly once - Client saves and retries and server discards duplicates
  - 1: at lest once - Client retries if no ACK from broker, duplicates tolerated
  - 0: at most once - No ACK from broker. No client storage or redelivery
- More QoS implies more resources: QoS 1 and QoS 2 messages are queued for offline subscribers, until they become available again
- From publishing client to broker, use the QoS in the message sent by the publishing client.
- From broker to subscribing client, use the QoS established by the client’s original subscription. A QoS may be downgraded if the subscribing client has a lower QoS than a publishing client.
- If a client has a subscription and the client disconnects, if the subscription is durable it will be available on reconnect

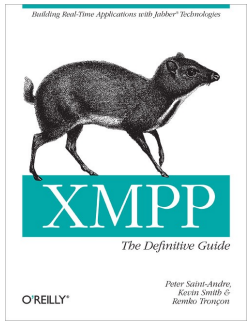
Example MQTT interfaces

- IBM Bluemix
  - The IBM Bluemix Internet of Things (IoT) service provides a simple but powerful capability to interconnect different kinds of devices and applications all over the world. What makes this possible? The secret behind the Bluemix IoT service is MQTT, the Message Queue Telemetry Transport
- Amazon AWS
  - The AWS IoT message broker is a publish/subscribe broker service that enables the sending and receiving of messages to and from AWS IoT. When communicating with AWS IoT, a client sends a message addressed to a topic like Sensor/temp/room1. The message broker, in turn, sends the message to all clients that have registered to receive messages for that topic. The act of sending the message is referred to as publishing.
- Microsoft Azure
  - Azure IoT Hub enables devices to communicate with the IoT Hub device endpoints using the MQTT v3.1.1 protocol on port 8883 or MQTT v3.1.1 over WebSocket protocol on port 443. IoT Hub requires all device communication to be secured using TLS/SSL (hence, IoT Hub doesn't support non-secure connections over port 1883).

Extensible Messaging and Presence Protocol

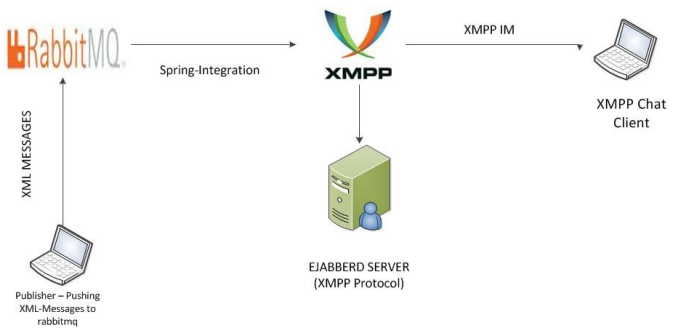
- Also known as XMPP - IETF Standard since 2004
- Open and free, supported by XMPP Standards Foundation (XSF)
- OS independent
- Originated as Jabber in 1998 for multi-user chat
- XMPP over WebSockets is defined in an IETF proposed standard
- Designed for client-server instant messaging but with no central authority
  - Architecture similar to email – anyone can run an XMPP server
  - May be isolated on a local intranet or may be public
  - XMPP messages are defined by an XML standards body
  - An XMPP message may be used to communicate between people (a chat application) or might be used by a sensor to report on its state.

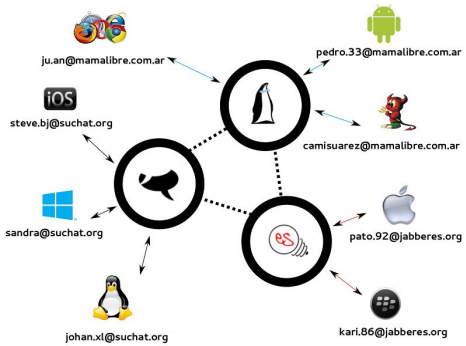
XMPP



XMPP Features

- Can be used to set up and tear down multimedia sessions – usually results in the use of the Real-time Transport (RTP) Protocol
- Supports generic publish-subscribe functionality – wide variety of payload types – RSS, Atom, etc.
- Supports Multi-user chat – similar to internet relay chat (IRC)
  - Reduces all the communication types and styles that are currently defined within the XMPP ecosystem to one model: publish-subscribe”.
  - Multi-user chat is a kind of publish-subscribe, since a single "publish" to the room results in multiple notifications to the room occupants, who are really subscribers to an information node.
  - One-to-one messaging is just another example of publish-subscribe (it is a special case of multi-user chat).



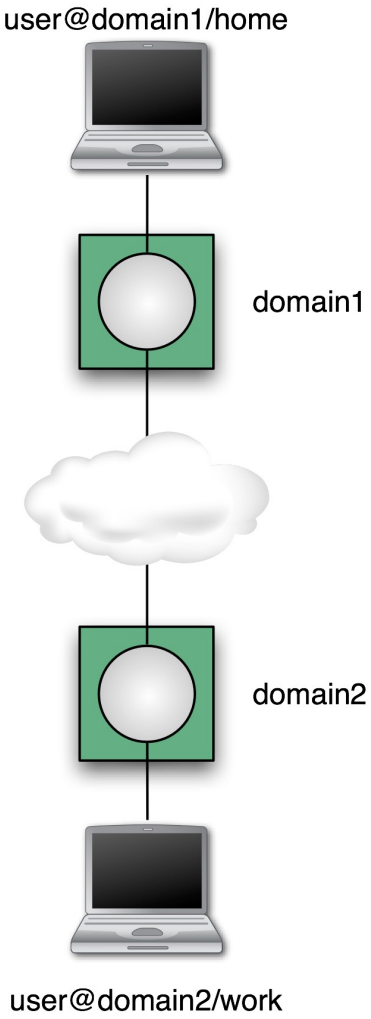


XMPP Presence and IM services

- The purpose of using such an application is to exchange relatively brief text messages with particular contacts in close to real time
- 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 XMPP needs to be able to:
  - Manage items in one's contact list (list is maintained on the server) in XMPP this list is called a "roster"
  - Exchange messages with one's contacts
  - Exchange presence information with one's contacts (send communication status to the server)
  - Manage presence subscriptions to and from one's contacts

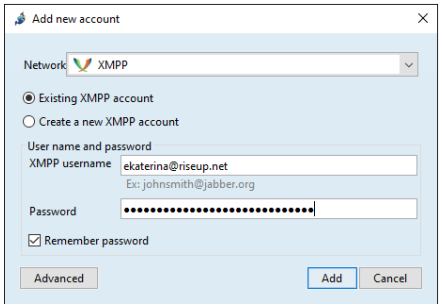
XMPP Architecture

- Addressing Scheme: `node@domain/resource`
  - JID = Jabber ID
    - Node: identity, e.g. user name
    - Domain: DNS domain name
    - Resource: device identifier
    - `node@domain` identifies a person
- Client talks to "local" server
  - Wherever the user account is hosted
  - Tied to directory if desired
  - Organizational policy enforced
- Servers talk to other servers
  - DNS lookup on domain portion of address
  - Dialback, MTLS for security
  - One connection for many conversations



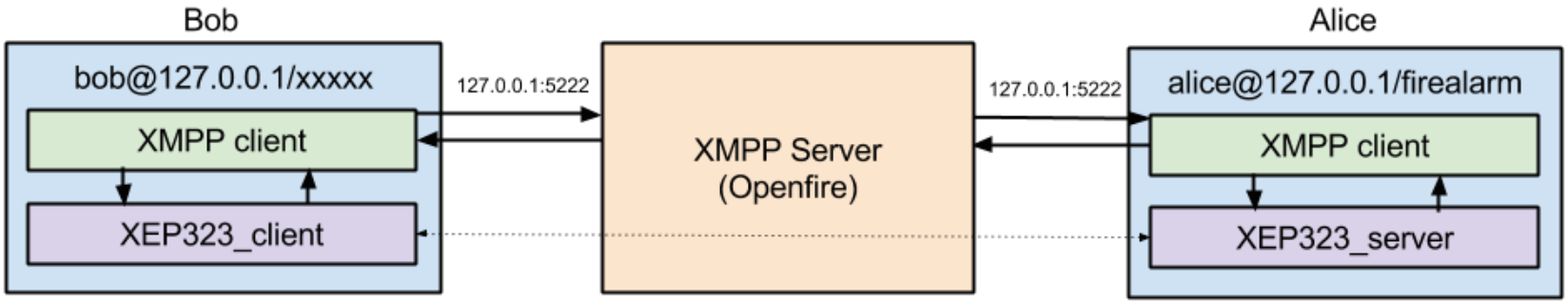
Naming Things

- Llike an email address , XMPP uses Jabber ID's (JIDs)
- All of these JIDs could be logged on at the same time.
  - mm6@andrew.cmu.edu Called a "bare JID"
  - mm6@andrew.cmu.edu/mobile Includes a resource. This is called a "full JID"
  - mm6@andrew.cmu.edu/tablet Resource used for message delivery
  - mm6@andrew.cmu.edu/auto Resource typically assigned by server
  - On Whatsapp [phone number]@s.whatsapp.net
- You may only subscribe to bare JIDS.
- A resource identifies a particular client belonging to the user
- A user can log in to an XMPP server multiple times, and in this case, the resource can denote a location.
  - For example, the sample user might have a JID for his main terminal
    - (DavidBowman@discovery.nasa.guv/terminal)
  - and another JID (session) from an EVA pod
    - (DavidBowman@discovery.nasa.guv/eva_pod1



Naming Things (2)

- The resource may be a random string
- Each connection is bound to a specific resource
- Many clients may use the same account at the same time. To send a message to one of them you must have the full JID.
- Only “friends” may communicate with each other.
  - Suppose Alice makes a request for a presence subscription to Bob’s bare JID.
  - If Bob agrees, Alice will receive Bob’s full JID along with presence status (online, offline, etc.)
  - Suppose Bob makes a request for a presence subscription to Alice’s bare JID.
  - If Alice agrees, Alice and Bob are now friends. Both each have the other’s full JID.



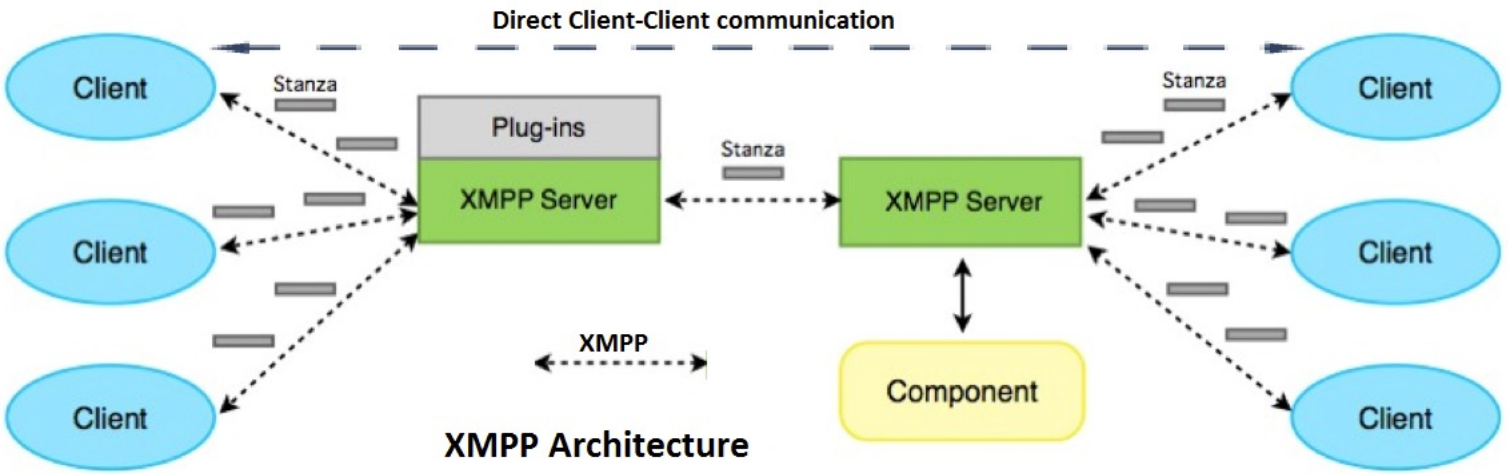
Naming Things (3)

- A contact list is called a **roster**
  - consists of any number of specific roster items, each roster item being identified by a unique JID
- A user's roster is stored by the user's server on the user's behalf so that the user may access roster information from any resource.

- Four operations are available
  - Roster Get
  - Roster Result
  - Roster Set
  - Roster Push

XMPP Basic connection

- Client initiates a TCP connection
- Client sends presence information to the server
- The client requests and receives its roster
- The client interacts with roster members (direct connection)
- The client disconnects



This is all done with standard XML messages using the XMPP vocabulary and grammar.

XMPP Streams

- Client connects TCP socket to server
- Client sends stream start tag:

```
<stream:stream xmlns='jabber:client'
xmlns:stream='http://etherx.jabber.org/streams'
               to='example.com'
               version='1.0'>
```

- Server sends stream start tag back:

```
<stream:stream xmlns='jabber:client'
xmlns:stream='http://etherx.jabber.org/streams'
               from='example.com’
               id='someid'
               version='1.0’>
```

- Each child element of stream is a “stanza”

Stream features

– After stream start, server sends feature list:

```
<stream:features>
  <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
  <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
    <mechanism>DIGEST-MD5</mechanism>
  </mechanisms>
  <compression xmlns='http://jabber.org/features/compress'>
    <method>zlib</method>
  </compression>
</stream:features>
```

– Client can negotiate any of these features

Within streams are XML stanzas

- Stanzas are well formed and complete XML messages
- All have `to='JID'` and `from='JID'` addresses
  - To gives destination
  - From added by local server
- Each stanza routed separately and all contents of a stanza passed along
- Three Stanza types enclosed in a stream tag:
  - <Presence> user status shared with all on the XMPP roster,
    - "I am online"
    - "I am interested in knowing about your presence", ...
  - <IQ> information query, request and change settings, service discovery find details about users or services available
    - four verbs available: set, get , result, and error
  - <Message> used for person to person chat (5 message types)
- Stanzas may include from and to parts as well as a stanza ID
- The stanza ID uniquely identifies the stanza

Presence Stanza

- Used to tell the server the user's or Things status
- May trigger backend events (announcements to others)
- Status is shared with others on the roster
- Five <show> tags (chat, away, extended away, do not disturb, and unavailable),
- default is "online" and may include text comments
- If you have a buddy who logs in, you will be notified of her status
- If she logs off, you will hear about that too.
- Server's may send the presence tag to the client

Example:

```
<presence>
  <show>dnd</show>
  <status>Meeting</status>
  <priority>1</priority>
</presence>
```

Subscribing to Presence

- Send a subscription request:

```
<presence to='juliet@example.com'
          type='subscribe’/>
```

- Approving a request:

```
<presence to='romeo@example.net'
          type='subscribed’/>
```

- Every time you change a subscription, you get a "roster push":

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item jid='romeo@example.net'
          subscription='from'/>
  </query>
</iq>
```

IQ Stanza

- Information Query is primarily about getting or setting information
- Four types (Get, Set, Result, and Error)
- Service discovery
  - Learn about servers, rooms, users, features, available services

IQ Request Example:

```
<iq type='get'
  id='roster_1'>
  <query xmlns='jabber:iq:roster'/>
</iq>
```

IQ Response Example:

```
<iq type='result'
    id='roster_1'>
  <query xmlns='jabber:iq:roster'>
    <item jid='romeo@example.net'
          name='Romeo'
          subscription='both'>
      <group>Friends</group>
    </item>
  </query>
</iq>
```

Message Stanza

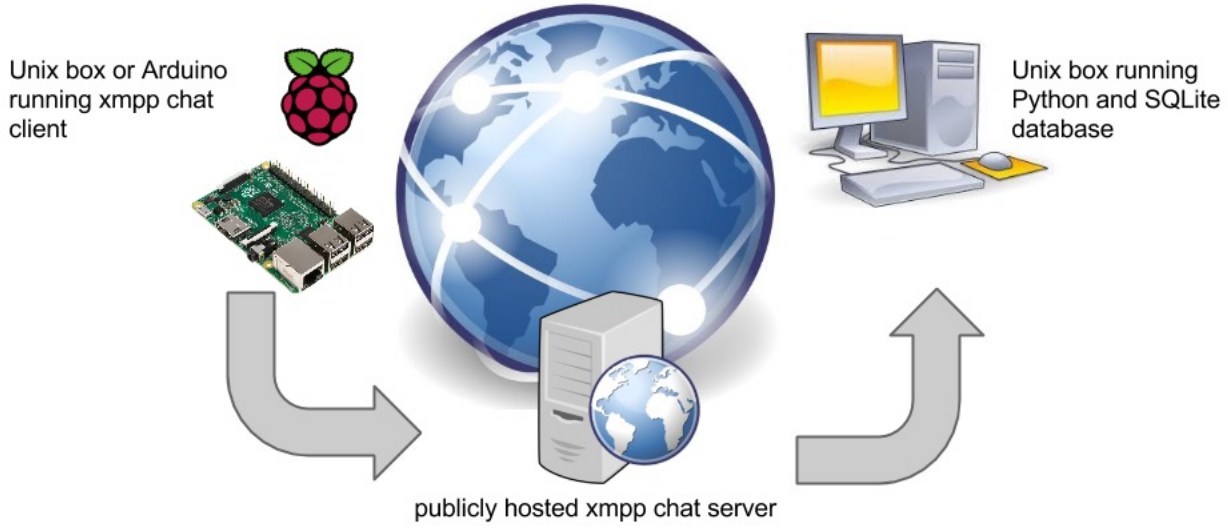
- Person to person or machine to machine extensible message
- Five types (Chat, Groupchat, Headline, Normal, Error)
- Typically includes a <body> element with XML content
- May include a <ping> element

Example:

```
<iq type='result'
    id='roster_1'>
  <query xmlns='jabber:iq:roster'>
    <item jid='romeo@example.net'
          name='Romeo'
          subscription='both'>
      <group>Friends</group>
    </item>
  </query>
</iq>
```

Devices (Not just people!)

- Programs or devices may listen on or write to an XMPP chat room
- The devices have a JID and so does the chat room.
- The devices need not be connected to the same server as the chat room. XMPP allows for client to server to server to client interactions.



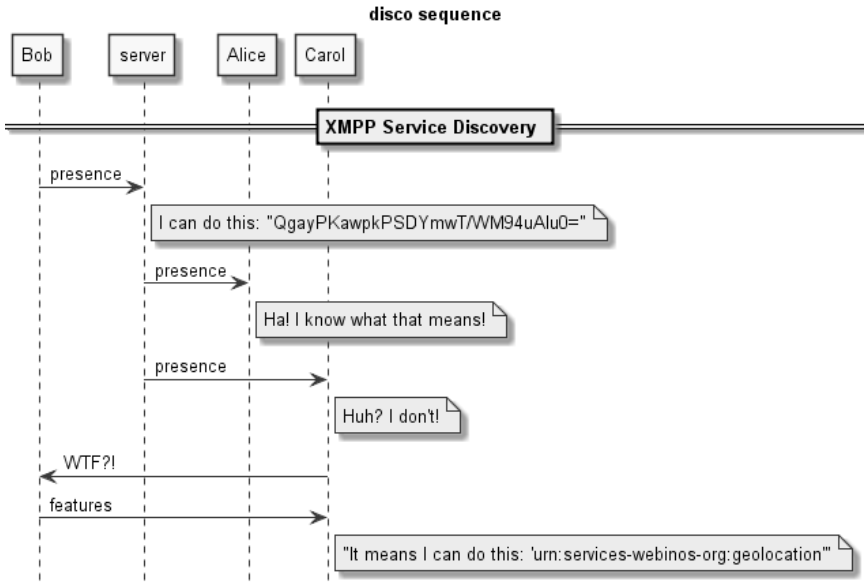
With Things we need Discovery[1]

- A naming service provides service references if you have the name in hand. Example: Phone book, the Domain Name Service (DNS)
- A directory service provides service references if you have attributes in hand. Example: Google search, Lightweight Directory Access Protocol (LDAP)
- A discovery service is a directory service that allows registration, de-registration, and lookUp of services in a spontaneous network – where clients and services may come and go. Example: You have printer attributes and discover a printer.
  - Discovery may be done with a directory server or be serverless.
  - In serverless discovery, participants collaborate to implement a distributed directory service. Two main models:
    - push model : services regularly advertise their services (multicast).
    - pull model: clients multicast their queries.

XMPP and Thing Discovery (1)

- During production of a Thing, decisions have to be made whether the following parameters should be pre-configured, manually entered after installation or automatically found and/or created by the device if possible (zero-configuration networking):
  - Address and domain of XMPP Server.
  - JID of the Thing.
  - JID of Thing Registry, if separate from the XMPP Server.
  - JID of Provisioning server, if different from the Registry and XMPP server

A “provisioning server” may be needed if the device needs special configuration parameters from that server. Perhaps with user involvement.



XMPP and Thing Registration

```
<iq type='set'
       from='thing@example.org/imc'
       to='discovery.example.org'
       id='1'>
      <register xmlns='urn:xmpp:iot:discovery'>
          <str name='SN' value='39487234873294872 3'/>
          <str name='MAN' value='www.ktc.se'/>
          <str name='MODEL' value='IMC'/>
          <num name='V' value='1.2'/>
          <str name='KEY' value='4857402340298342'/>
      </register>
</iq>
```

Reading sensor data

- Request: An AMR device is used for Automatic Meter Reading

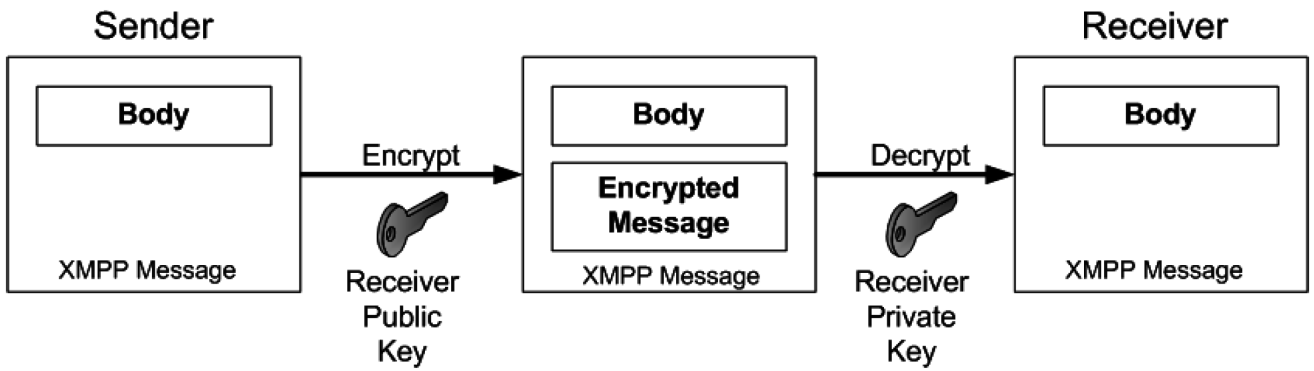
```
<iq type='get' from='client@clayster.com/amr'
               to='device@clayster.com' id='S0001'>
              <req xmlns='urn:xmpp:iot:sensordata' seqnr='1’
              momentary='true'/>
</iq>
```

- Response from AMR

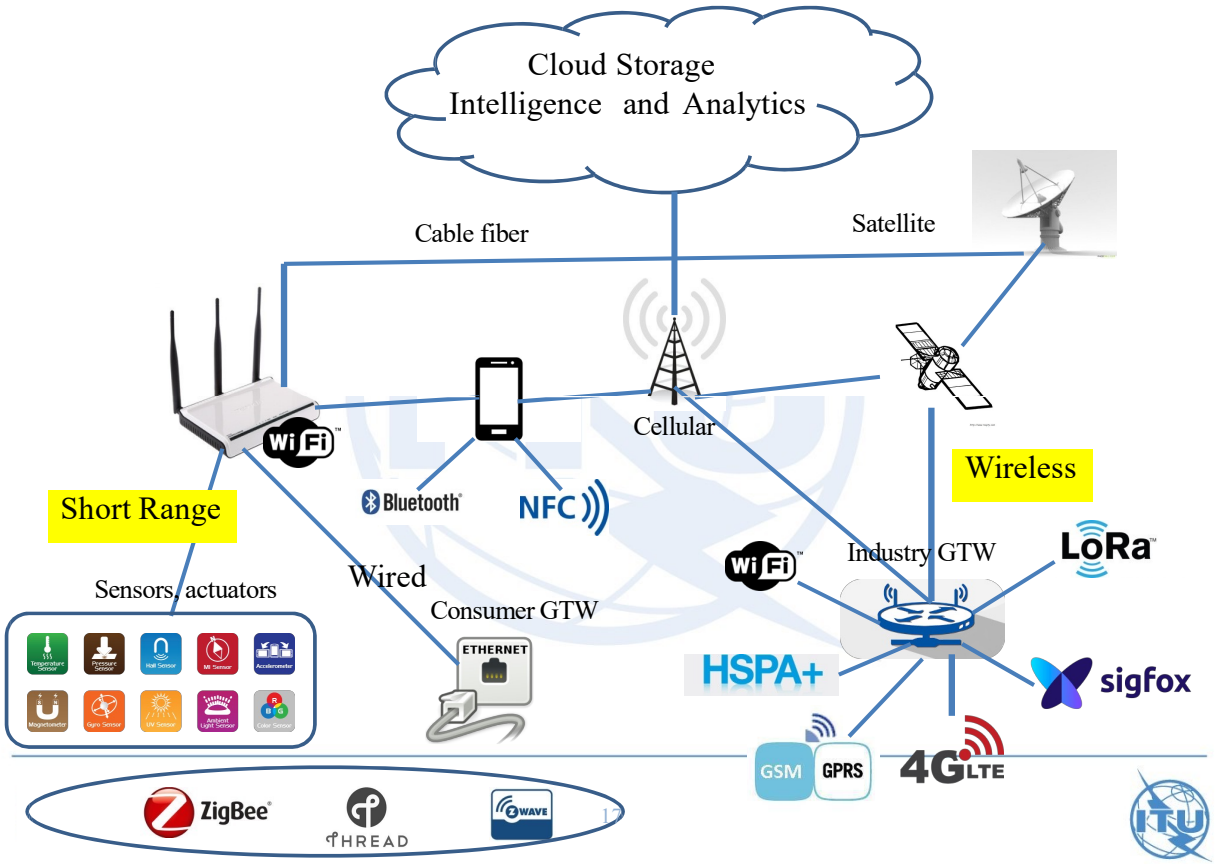
```
<iq type='result' from='device@clayster.com'
                  to='client@clayster.com/amr' id='S0001'>
       <accepted xmlns='urn:xmpp:iot:sensordata' seqnr='1'/>
</iq>
```

XMPP Security

- Start-TLS
  - Prove the identity of the server
  - Prove the identity of the user (optional)
  - Encryption
  - Data integrity
- SASL (RFC 4422)
  - Authentication
  - Optional encryption (rarely used)
  - Pluggable (e.g. passwords, Kerberos, X.509, SAML, etc.)



IoT: Is there a General Architecture?

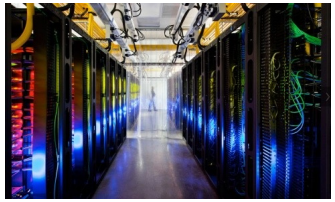


IoT: 4 layer Model

Integrated Applications

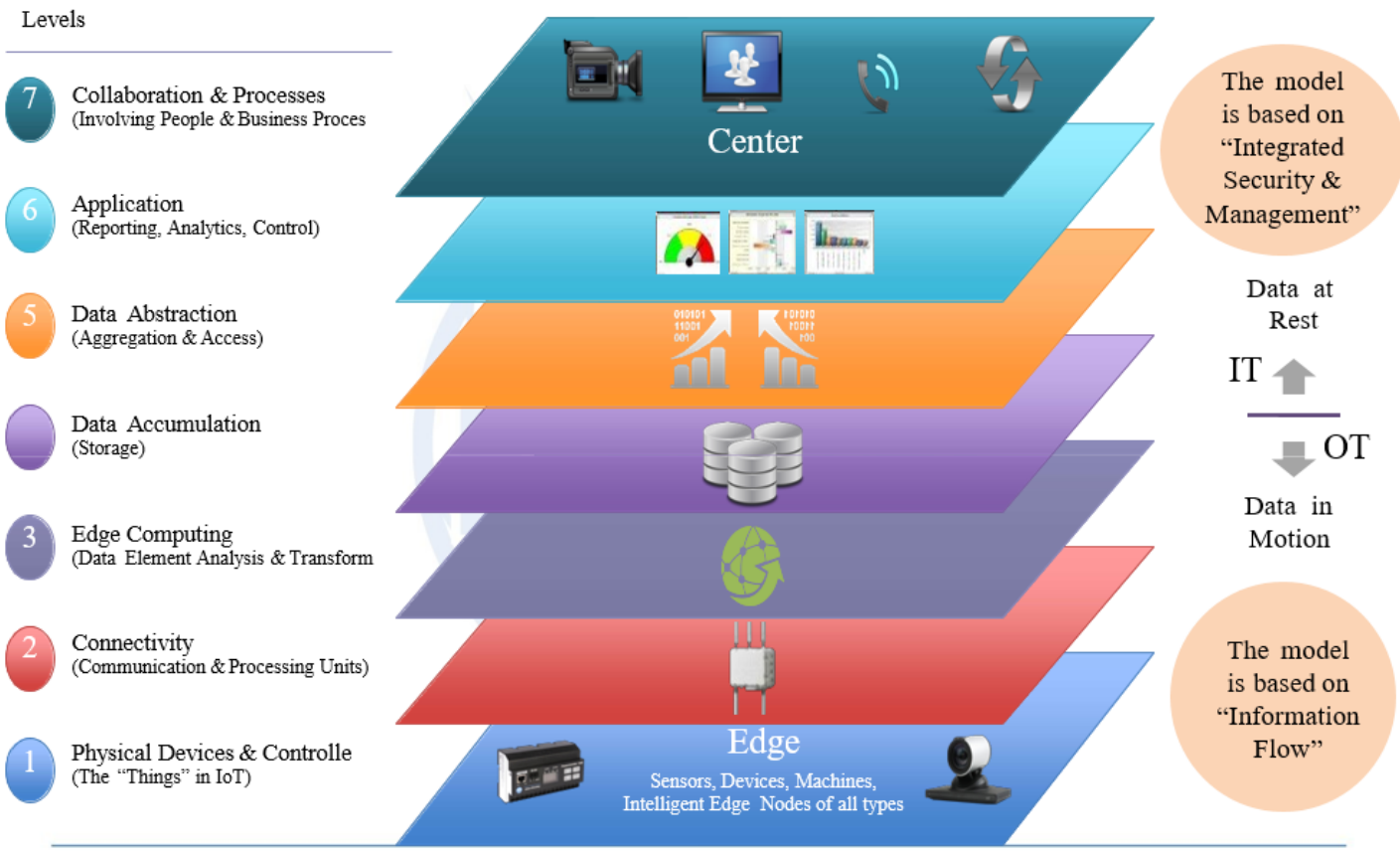
Information Processing

Network Infrastructure

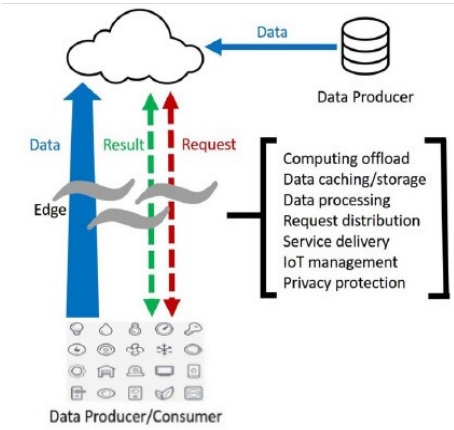


Sensing and Identification

IoT: Reference Model

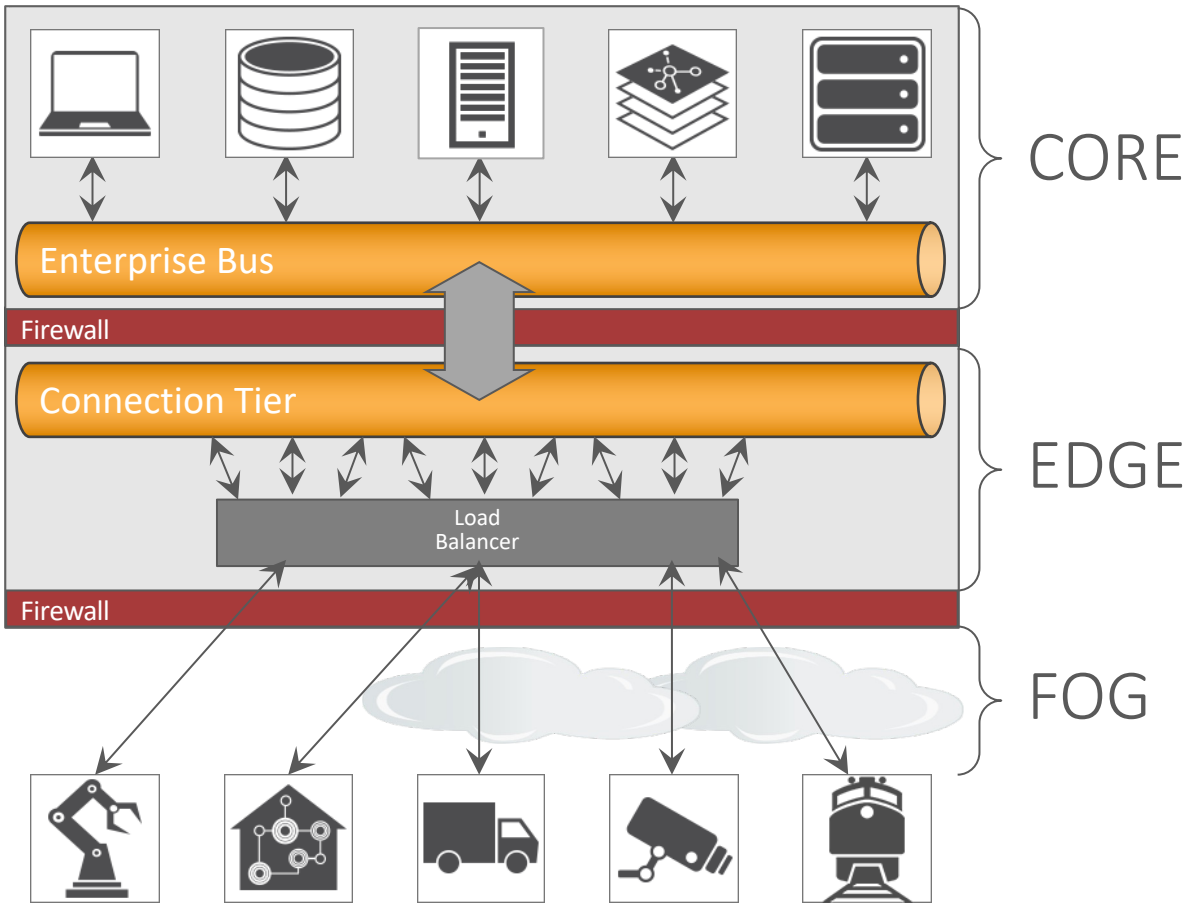


In IoT solutions supporting Fog Com putting (FC) part of the application processing is executed directly at IoT objects and only when needed. More complex and resource-consuming tasks are transferred to higher level units (FC units) or directly to the cloud.

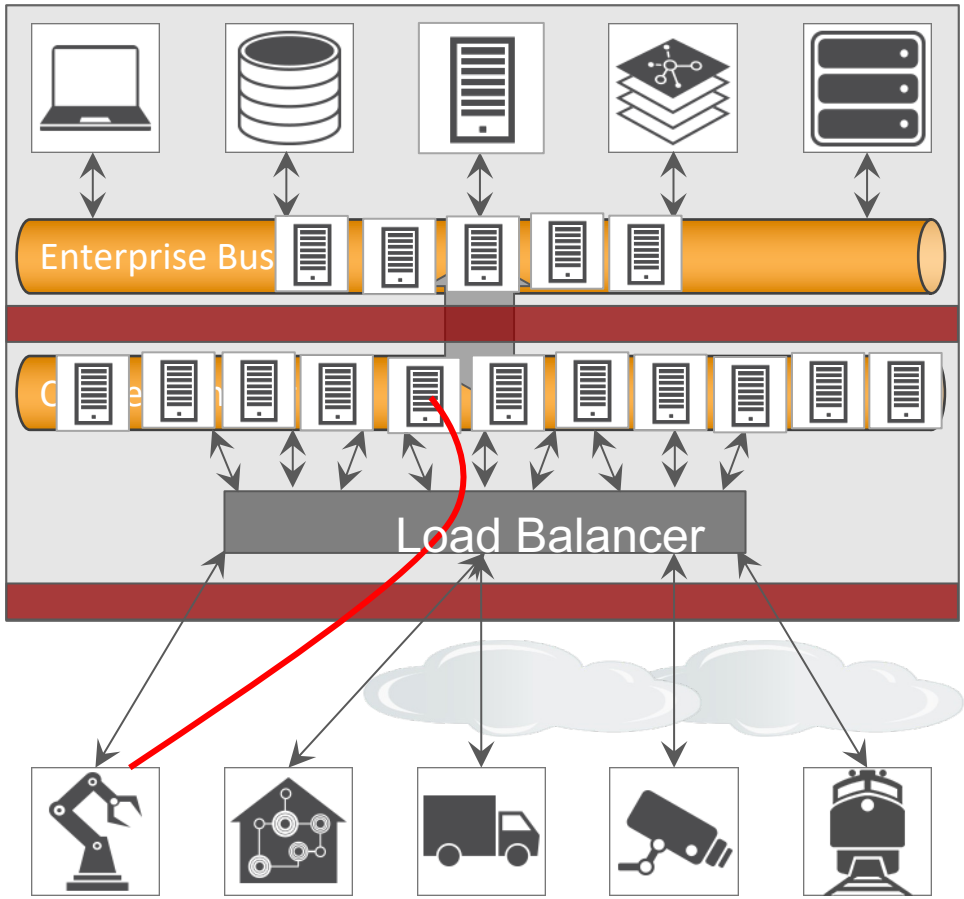


Typical IoT Cloud-based Architecture

- Connections, queues, subscriptions
- MEPs
  - Device-initiated pub/sub
  - Device-initiated request/reply
  - Cloud app-initiated pub/sub
  - Cloud app-initiated request/reply
- QoS – MQTT QoS0, QoS1
- Single point of contact for apps

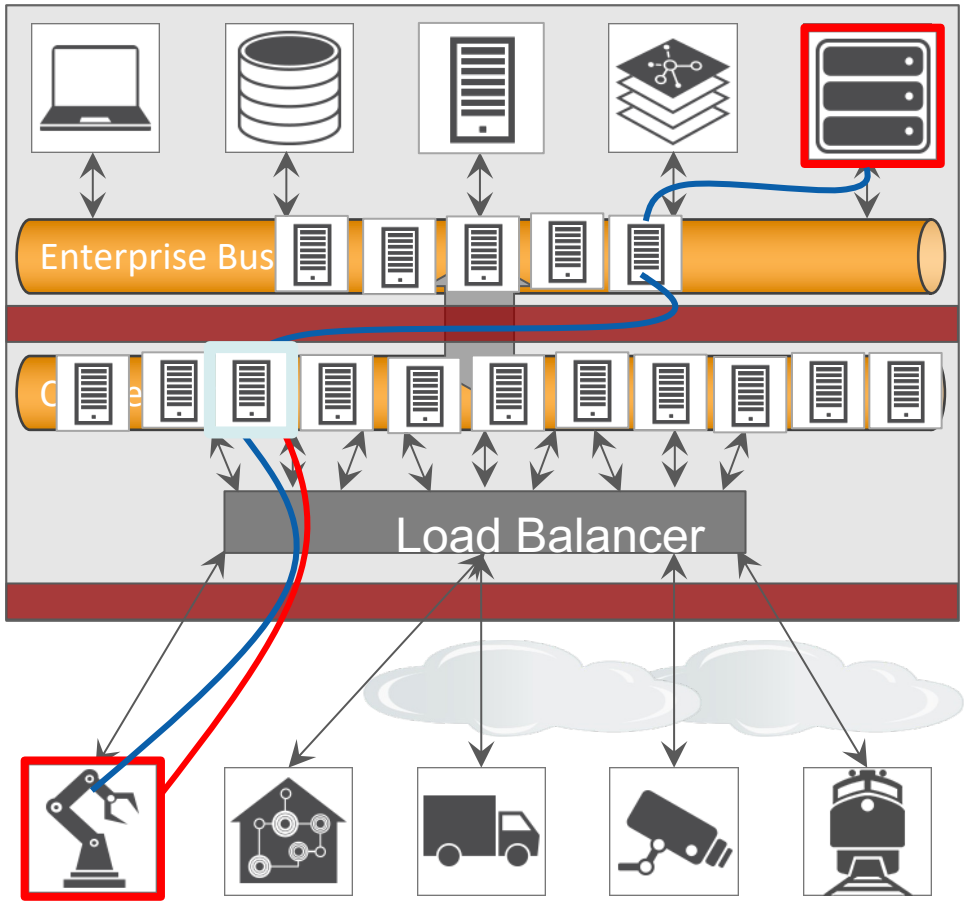


Implementation Considerations



- Connections, queues, subscriptions
- MEPs
  - Device-initiated pub/sub
  - Device-initiated request/reply
  - Cloud app-initiated pub/sub
  - Cloud app-initiated request/reply
- QoS – MQTT QoS0, QoS1
- Single point of contact for apps

Challenges with Stateless Connect Layer



- Device Queues
  - Where is the queue for each device for QoS1 subscriptions?
  - How does Connection layer find its queue/state? (among the millions....)
- App to device MEP
  - How does Core layer find Connection layer server for given MQTT device for QoS0?
  - Where is the Queue for a given device for QoS1?
  - Need massive subscription scalability updated in real time

IoT design requirements

IoT Network	Impact on IoT Systems Design
Scale	**Tens of thousand sensors in a given site; or millions distributed geographically** More pressure on application architectures, network load, traffic types, security, non-standard usage pattern
Heterogeneous end-points	**Vast array of sensors, actuators, smart devices – IP or non-IP** Diverse data rate exchange, form factor, computing and communication capabilities, legacy protocols
Low Capex and Opex requirement	**May be deployed before activation, maybe or cannot-be accessed once deployed** • Low numbers of gateways Link budget: e.g: UL: 155 dB (or better), DL: Link budget: 153 dB (or better) • Devices deliver services with little or no human control, difficult to correct mistakes, device management is key
Criticality of services	**Human life critical (Healthcare), Critical infrastructure (Smart Grid)** Stringent latency (10ms for SG) and reliability requirements, may challenge/exceed network capabilities of today
Intrusiveness	**Things with explicit intent to better manage end-users (eHealth, Smart Grid)** Issues of Privacy become major obstacles
Geography	**Movement across borders** Issues of numbering for unique identification

Source: **ITU CoE training on BB networks planning, Bangkok, Sep 2017**

IoT network connectivity requirements

IoT Network	Impact on IoT Systems Design
Resource-constrained endpoints	**Severely resource constrained (memory, compute)** Cost motivation: compute/memory several orders of magnitude lower, limited remote SW update capability, light protocols, security
Low Power	**Some end-point types may be mostly 'sleeping' and awakened when required** • Sensors cannot be easily connected to a power source • Reduced interaction time between devices and applications (some regulations state duty cycle of no more than 1%) • Idle mode most of the time (energy consumption of around 100 μW). Connected mode just for transmission (mA) • < 100 MHz clock frequency • Embedded memory of few Mb
Embedded	**Smart civil infrastructure, building, devices inside human beings** Sensors deployed in secure or hostile operating conditions, difficult to change without impacting system, Security
Longevity	**Deployed for life typically, have to build-in device redundancy** Very different lifetime expectancy, rate of equipment change in IoT business domains much lower than ICT Industry
High Sensitivity on reception	**Gateways and end-devices with a high sensitivity around** -150 dBm/-125 dBm with Bluetooth lower than -95 dBm in in cellular

Source: ITU CoE training on BB networks planning, Bangkok, Sep 2017

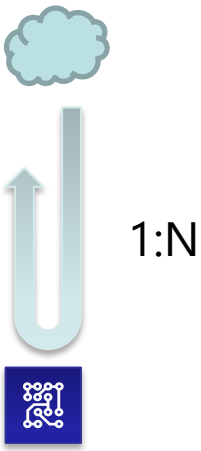
Patterns of communication

Telemetry

Information flowing from a device to other systems for conveying status of device and environment

Inquiries

Requests from devices looking to gather required information or asking to initiate activities



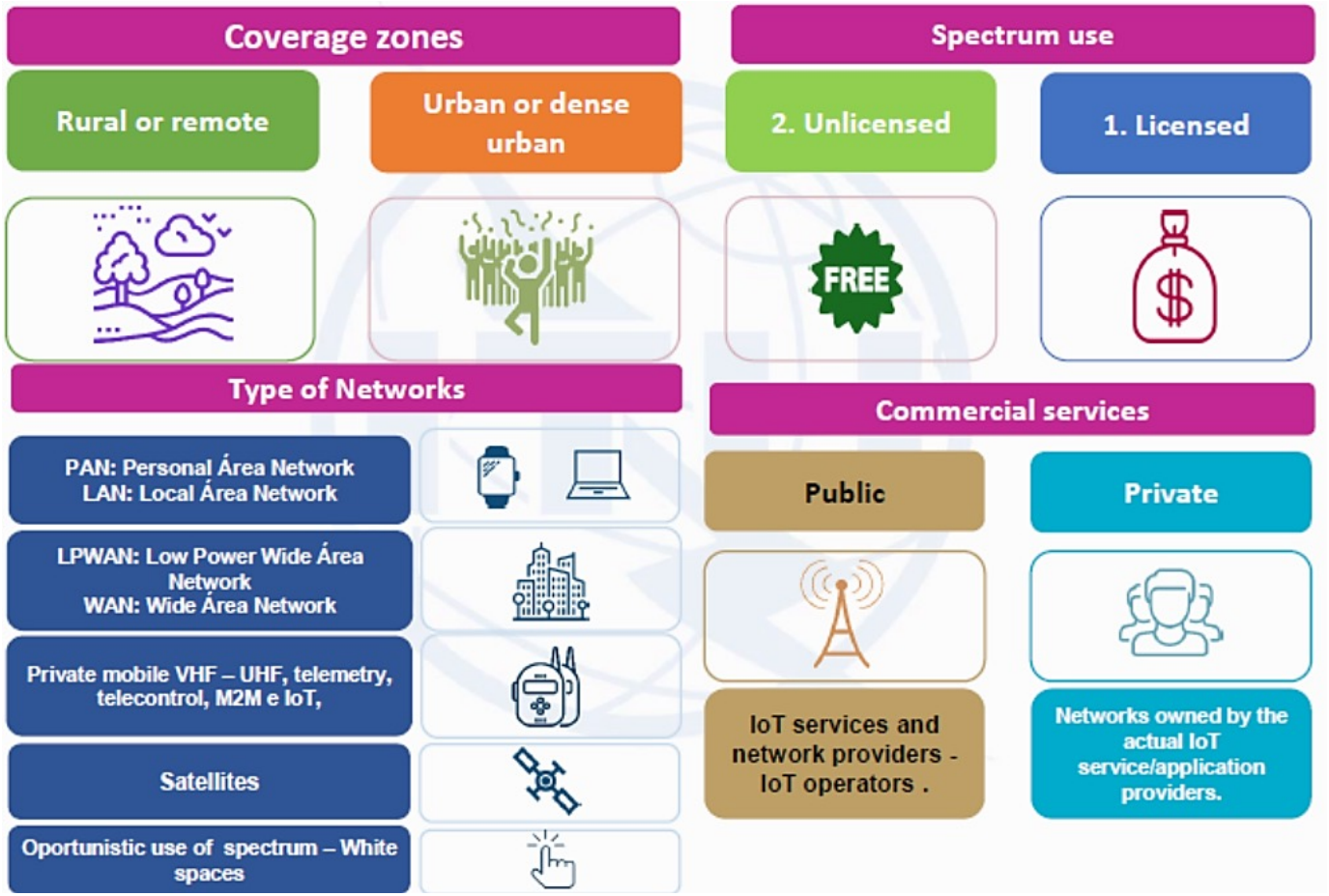
Commands

Commands from other systems to a device or a group of devices to perform specific activities

Notifications

Information flowing from other systems to a device (-group) for conveying status changes in the rest of the world

IoT Usage Cases



Source: ITU
Workshop on Spectrum Management for Internet of Things Deployment, 22 November 2016, Geneva

IoT Usage Categories

Category	Sub-category	
Consumer IoT	Consumer electronics	Smart TVs, home entertainment (games consoles, speakers), personal entertainment (MP3 players, portable gaming devices), set-top boxes
	Smart home	Home appliances (fridges, washing machines), home infrastructure (routers), home security (alarms), energy monitoring (thermostats)
	Wearables	Fitness trackers (including personal health trackers), smart watches
	Smart vehicles	Connected cars, connected bikes, insurance telematics
	Consumer – others	Trackers for children, the elderly and pets, as well as drones and robots
Industrial IoT	Smart city	Public transport, surveillance, electric vehicle charging, street lighting, parking, waste management
	Smart utilities	Energy, water and gas smart metering, smart grid
	Smart retail	PoS, digital signage, vending machines, ATMs
	Smart manufacturing	Inventory tracking, monitoring and diagnostics, warehouse management
	Smart buildings	Heating and air con, security, lighting, hot desks, office equipment
	Health	Remote monitoring of medical devices, emergency vehicle infrastructure
	Enterprise – others	Fleet management, applications in agriculture, oil, mining, construction

Source: GSMA Intelligence - IoT: the next wave of connectivity and services

IoT and regulatory issues

- Licensed Vs Non Licensed spectrum
- Area of license
- Numbering
- Standardization
- Infrastructure sharing
- Access to data and open IOT platforms
- Data analytics
- Mobile data roaming
- Consumer protection
- Quality of Service
- USO
- Taxation

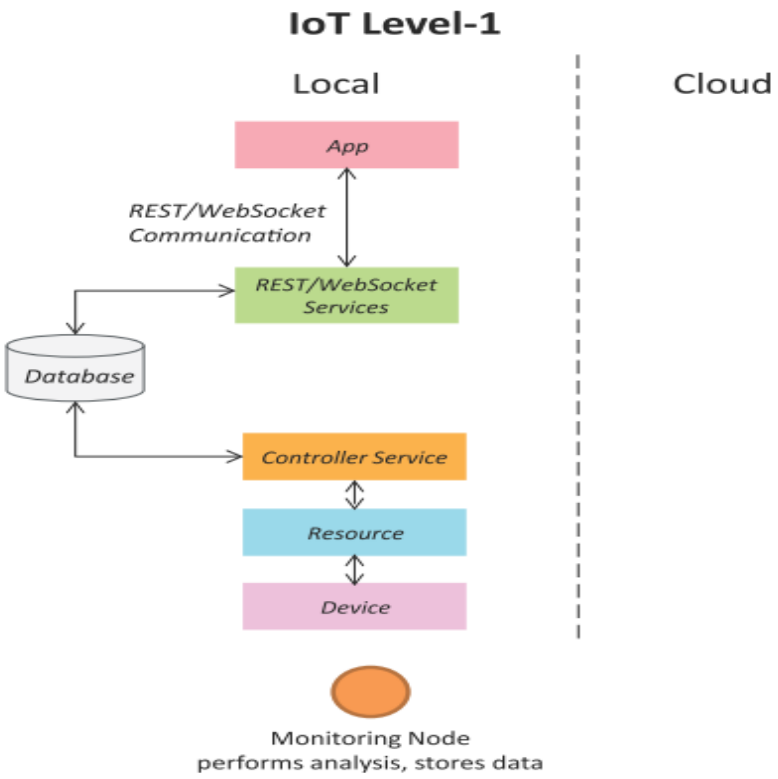
One world, one global SIM: How ITU-allocated 'global IMSI ranges' support IoT and M2M connectivity

https://news.itu.int/one-world-one-global-sim/

! Global International Mobile Subscriber Identity (IMSI) ranges are signified by the shared Mobile Country Code '**901**', a code without ties to any particular country.

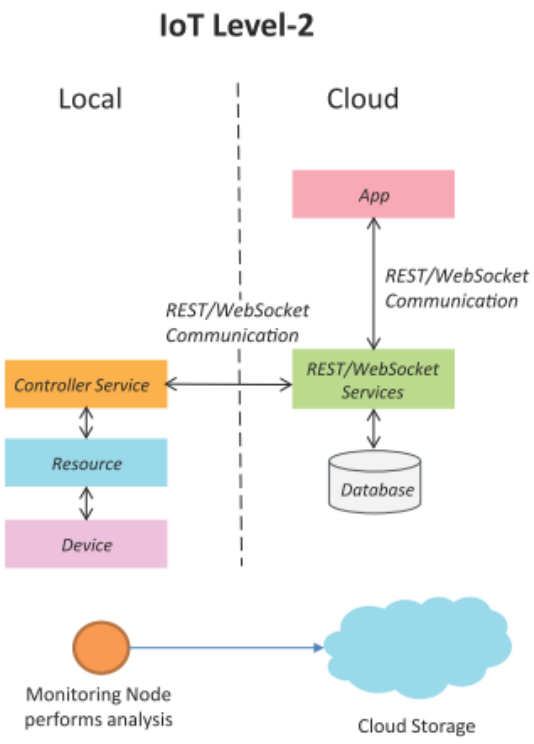
IoT Level-1 architectures

- A level-1 IoT system **has a single node/device** that performs sensing and/or actuation, stores data, performs analysis and hosts the application.
- Level-1 IoT systems are suitable for **modelling low-cost and low-complexity solutions** where the data involved is not big and the **analysis requirements are not computationally intensive.**



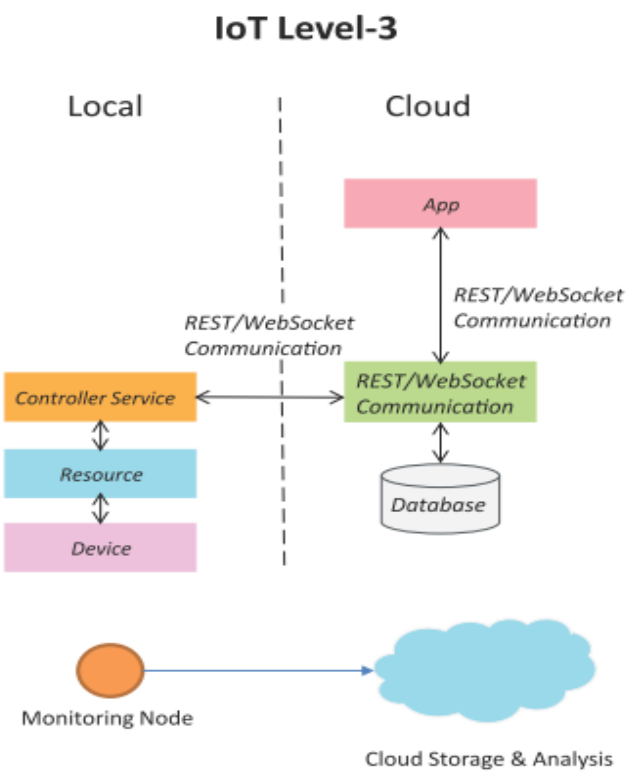
IoT Level-2 architectures

- A level-2 IoT system has a **single node that performs sensing and/or actuation and local analysis.**
- **Data is stored in the cloud** and the application is usually cloud-based.
- Level-2 IoT systems are **suitable for solutions where the data involved is big;** however, the primary **analysis requirement is not computationally intensive** and can be done locally.



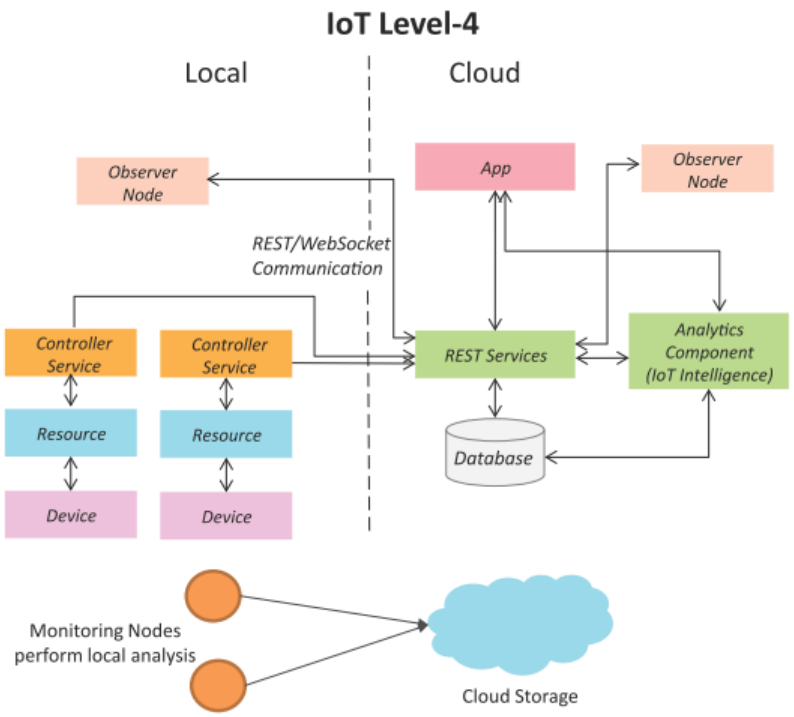
IoT Level-3 architectures

- A level-3 IoT system has a **single node. Data is stored and analyzed in the cloud** and the application is cloud-based.
- Level-3 IoT systems are suitable for solutions **where the data involved is big and the analysis requirements are computationally intensive.**



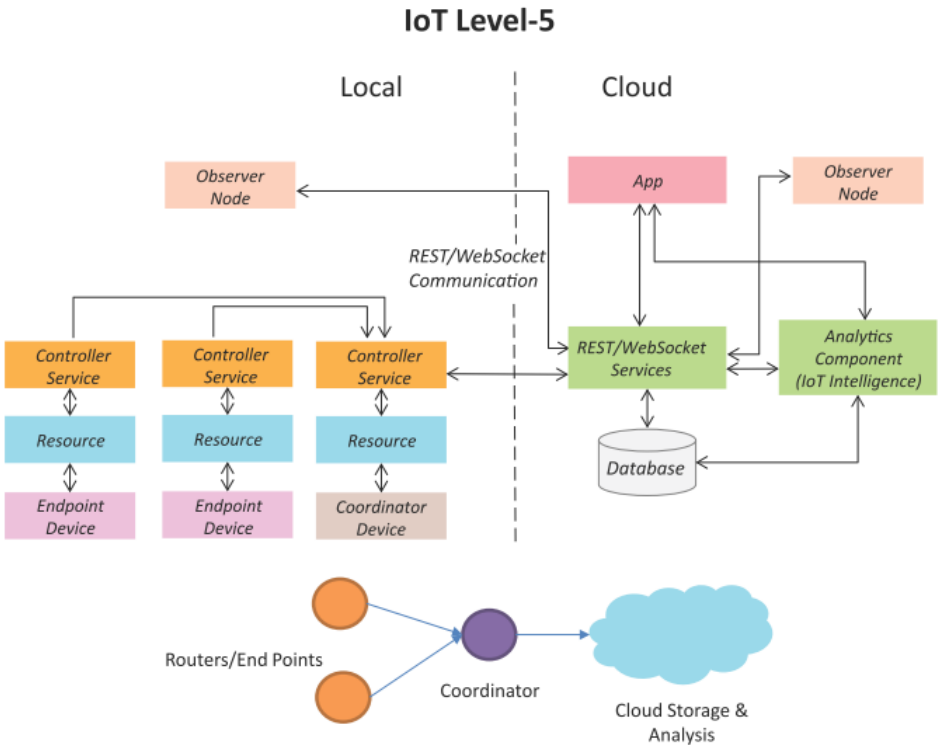
IoT Level-4 architectures

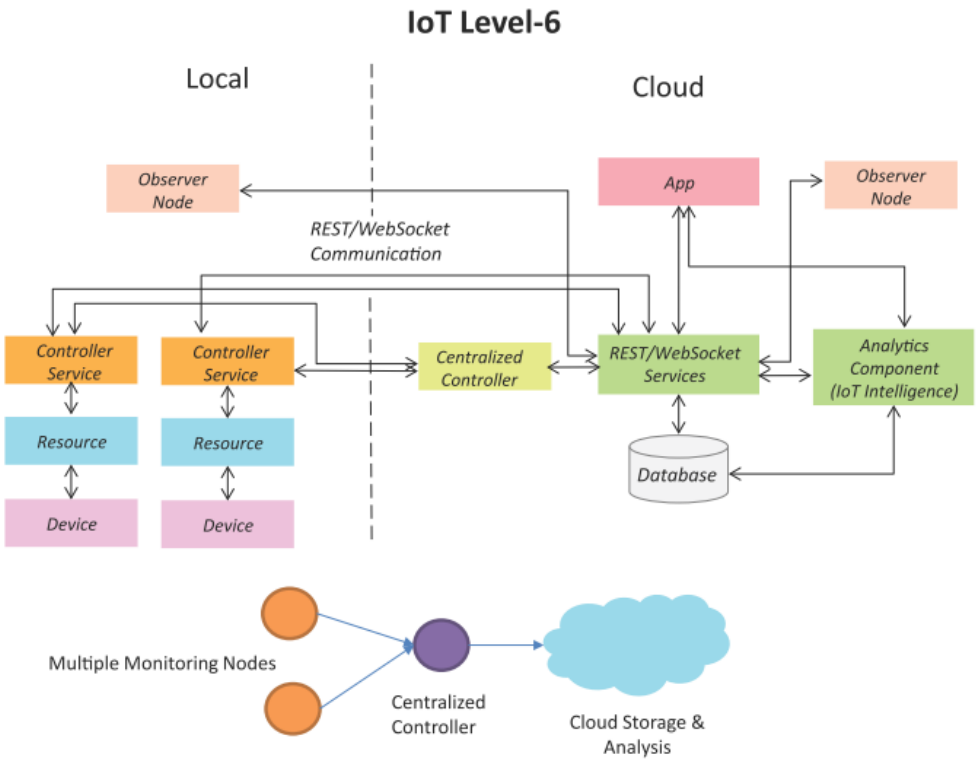
- A level-4 IoT system has multiple nodes that perform local analysis. Data is stored in the cloud and the application is cloud-based.
- Level-4 contains local and cloud-based observer nodes which can subscribe to and receive information collected in the cloud from IoT devices.
- Level-4 IoT systems are suitable for solutions where multiple nodes are required, the data involved is big and the analysis requirements are computationally intensive.



IoT Level-5 architectures

- A level-5 IoT system has multiple end nodes and one coordinator node.
- The end nodes perform sensing and/or actuation.
- The coordinator collects data from the end nodes and sends it to the cloud.
- Data is stored and analyzed in the cloud and the application is cloud-based.
- Level-5 IoT systems are suitable for solutions based on WSNs networks, in which the data involved is big and analysis requirements are computationally intensive.



IoT Level-6 architectures

- A level-6 IoT system has multiple independent end nodes that perform sensing and/or actuation and send data to the cloud.
- Data is stored in the cloud and the application is cloud-based.
- The analytics component analyzes the data and stores the results in the cloud database.
- The results are visualized with the cloud-based application.
- The centralized controller is aware of the status of all the end nodes and sends control commands to the nodes.