

Internet of Things

(Appunti delle lezioni teoriche del prof. Vallati, tranne IoT platforms)

Autore: Gabriele Frassi

Università di Pisa

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Artificial Intelligence and Data Engineering

prof. Giuseppe Anastasi, prof. Carlo Vallati, ing. Francesca Righetti

Se questi appunti sono stati utili e vuoi ringraziarmi in qualche modo:

<https://www.paypal.com/paypalme/GabrieleFrassi>

SOMMARIO DISPENSA

1	IOT AND CLOUD INTEGRATION.....	5
1.1	INTRODUZIONE	5
1.1.1	<i>Differenza di argomenti.....</i>	<i>5</i>
1.1.2	<i>Idea of Cloud Integration.....</i>	<i>5</i>
1.1.3	<i>Is still a vision?.....</i>	<i>6</i>
1.2	TIPOLOGIE DI SISTEMI.....	6
1.2.1	<i>Architettura tipica: sistemi verticali.....</i>	<i>6</i>
1.2.2	<i>Architettura ideale (Goal/Vision of IoT): sistemi orizzontali</i>	<i>7</i>
1.2.2.1	<i>Idea</i>	<i>7</i>
1.2.2.2	<i>Come otteniamo sistemi orizzontali e converged infrastructure?.....</i>	<i>8</i>
1.3	MQTT (MESSAGE QUEUING TELEMETRY TRANSPORT) PROTOCOL	9
1.3.1	<i>Introduzione.....</i>	<i>9</i>
1.3.2	<i>MQTT-based Cloud Platforms.....</i>	<i>11</i>
1.3.3	<i>Rapid prototype-to-product time: centralità della WiFi</i>	<i>11</i>
1.4	VANTAGGI E SVANTAGGI DI UNA CLOUD PLATFORM	12
1.5	ESEMPI DI INDUSTRIAL USE CASES	13
1.5.1	<i>Smart Manufacturing</i>	<i>13</i>
1.5.2	<i>Autonomous Robot.....</i>	<i>13</i>
1.5.3	<i>Augmented Reality</i>	<i>14</i>
1.6	DEMO IOT PRODUCT.....	14
1.7	FOG COMPUTING	16
1.7.1	<i>Questioni precedenti.....</i>	<i>16</i>
1.7.2	<i>Introduzione.....</i>	<i>17</i>
1.7.3	<i>Tecnologie alla base del fog computing</i>	<i>18</i>
1.7.3.1	<i>Virtualization</i>	<i>18</i>
1.7.3.2	<i>Wireless Technologies</i>	<i>18</i>
1.7.3.3	<i>Da MQTT a Web of Things</i>	<i>19</i>
2	WEB OF THINGS AND REST SYSTEMS.....	20
2.1	INTRODUZIONE	20
2.2	SERVICE-ORIENTED ARCHITECTURE.....	20
2.3	ISPIRAZIONI PER IL WEB OF THINGS: WORLD WIDE WEB	21
2.3.1	<i>Introduzione.....</i>	<i>21</i>
2.3.2	<i>Identification functionality: Uniform Resource Identifier (URI)</i>	<i>22</i>
2.3.3	<i>Data exchange protocol: Hypertext Transfer Protocol (HTTP)</i>	<i>22</i>
2.3.4	<i>Data encoding system: HyperText Markup Language (HTML).....</i>	<i>23</i>
2.3.5	<i>Evolution to Web Services: URI e HTTP, XML al posto di HTML.....</i>	<i>23</i>
2.4	PROPOSTA: IMPLEMENTAZIONE DEI WEB SERVICES CON PROTOCOLLI CLOUD (SOAP E WSDL).....	24
2.5	PROPOSTA: RESTFUL WEB SERVICES	26
2.5.1	<i>Introduzione.....</i>	<i>26</i>
2.5.2	<i>Esempio: API di Google Maps.....</i>	<i>26</i>
2.5.3	<i>Esempi nel Web of Things.....</i>	<i>27</i>
2.5.4	<i>Siamo soddisfatti? Non ancora.....</i>	<i>28</i>
3	CONSTRAINED APPLICATION PROTOCOL (COAP).....	29
3.1	INTRODUZIONE	29
3.1.1	<i>CoAP: come ci siamo arrivati e cosa consiste</i>	<i>29</i>
3.1.2	<i>Novità principali</i>	<i>30</i>
3.1.3	<i>Modello request-response e optional reliability implementata in asynchronous messages</i>	<i>30</i>
3.1.4	<i>Request methods.....</i>	<i>31</i>
3.1.5	<i>Classificazione dei response codes</i>	<i>33</i>
3.2	REQUEST/RESPONSE TRANSPORT	34
3.2.1	<i>Introduzione di lightweight reliability mechanisms.....</i>	<i>34</i>
3.2.2	<i>Reliable transmission.....</i>	<i>34</i>
3.2.3	<i>Non-reliable transmission.....</i>	<i>35</i>

3.2.4	<i>Modalità di trasmissione della risposta</i>	35
3.2.5	<i>Esempi di Request/Response</i>	36
3.2.6	<i>Empty message: definizione e CoAP ping</i>	38
3.2.7	<i>Message format</i>	38
3.3	<i>PROXYING</i>	40
3.3.1	<i>Introduzione</i>	40
3.3.2	<i>Tipologie di proxy: forward proxy e reverse proxy</i>	41
3.3.3	<i>Proxy translation</i>	42
3.4	<i>[NOVITÀ] RESOURCE OBSERVING</i>	42
3.4.1	<i>Introduzione</i>	42
3.4.2	<i>Esempi di observable resources</i>	44
3.4.3	<i>Consistency</i>	45
3.4.4	<i>Observing and intermediaries</i>	45
3.5	<i>[NOVITÀ] DISCOVERY</i>	46
3.5.1	<i>Introduzione</i>	46
3.5.2	<i>Service discovery</i>	46
3.5.3	<i>Resource discovery e CoRE Link Format</i>	46
4	INTEROPERABLE DATAENCODING	48
4.1	<i>INTRODUZIONE</i>	48
4.2	<i>[DAL CLOUD] XML</i>	48
4.2.1	<i>Introduzione</i>	48
4.2.2	<i>Well-formed document</i>	50
4.2.3	<i>Document schema</i>	50
4.3	<i>[DAL CLOUD] JSON</i>	52
4.4	<i>SCELTA TRA JSON E XML: COME?</i>	53
4.5	<i>[NOVITÀ] ACCENNI SU EXI E CBOR</i>	53
5	SENSOR DATA REPRESENTATION AND MODELS	54
5.1	<i>INTRODUZIONE E NUOVO PROBLEMA</i>	54
5.2	<i>DATA SEMANTIC</i>	54
5.2.1	<i>Problema</i>	54
5.2.2	<i>Proposta per lo use-case data reporting: Sensor Markup Language (SenML)</i>	55
5.2.2.1	<i>Spiegazione</i>	55
5.2.2.2	<i>Esempio: misurazione di voltaggio e corrente</i>	58
5.3	<i>AUTOMATED DISCOVERY: IPSO SMART OBJECTS</i>	58

1 IOT AND CLOUD INTEGRATION

1.1 Introduzione

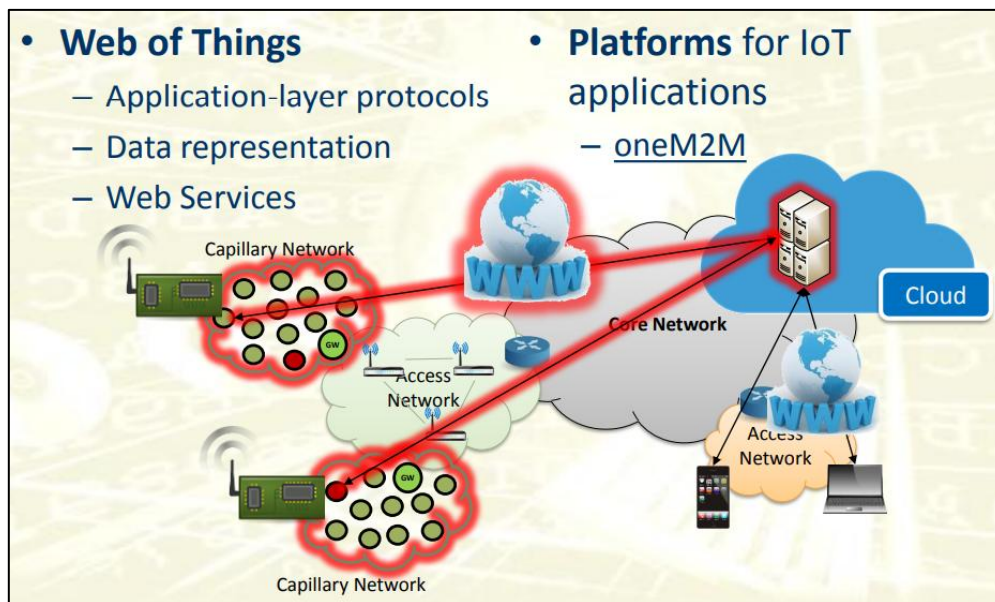
1.1.1 Differenza di argomenti

Cosa introduciamo di più rispetto alla parte del prof. Anastasi?

- **Anastasi: lower part of communication stack.**
Concetti teorici basilari sull'Internet of Things. Cosa sono gli Smart Objects, come è possibile integrare i dispositivi utilizzando IPv6.
- **Vallati: higher part of communication stack.**
Vogliamo riempire il vuoto (*fill the gap*) tra quanto spiegato dal prof. Anastasi e le applicazioni effettivamente utilizzate da noi utenti. Come? Col Cloud! Definiamo quanto segue per creare *large scale and interoperable systems*
 - o *Application protocols*
 - o *Semantics*
 - o *Languages*
 - o *Architectural schemas*

1.1.2 Idea of Cloud Integration

Il primo concetto che andiamo a introdurre è quello della *Cloud Integration*



- Vogliamo creare un sistema di grandi dimensioni (*large scale system*) dove i dispositivi IoT sono integrati insieme.
- Cosa abbiamo?
 - o Da una parte la rete IoT (tipicamente una LLN) che permette comunicazione tra i dispositivi (*where devices are interconnected*).
 - La rete non opera in isolamento: è solitamente caratterizzata da un gateway/border router che permette la connessione al resto di Internet (o comunque ad alcune reti esterne, grazie a IPv6).
 - La cosa appena detta è importante perché connessione alla rete esterna permette alla WSN di ricevere e trasmettere messaggi a soggetti non appartenenti alla rete IoT. E' il passaggio antecedente alla creazione dell'infrastruttura Cloud.
 - o Dall'altra parte abbiamo l'infrastruttura Cloud, che supporta l'esecuzione di applicazioni.
 - Si fornisce un'interfaccia agli utenti
 - Si forniscono funzionalità/servizi.

1.1.3 Is still a vision?

Domanda: *is still a vision?* Domanda retorica!



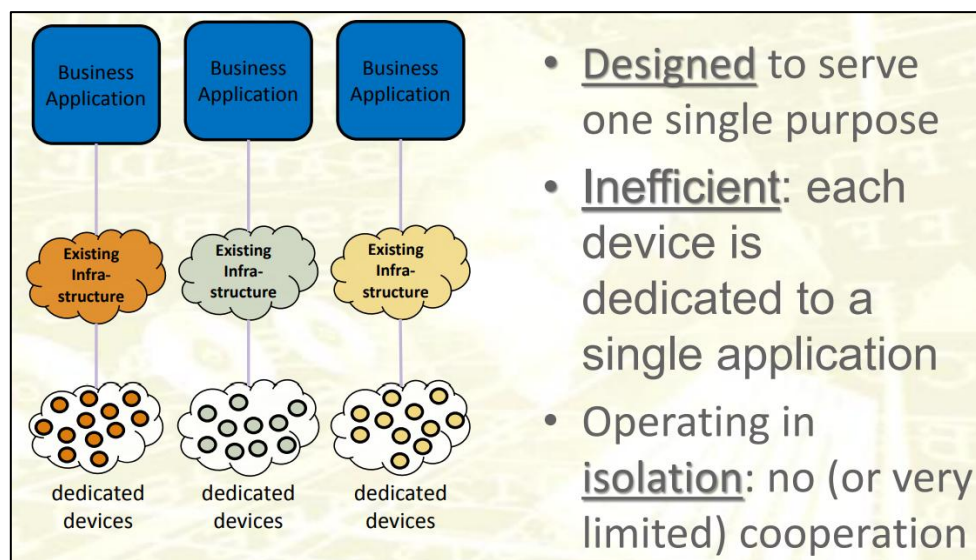
- Il market è ricco di smart devices (*smart thermostat, smart washing machine, smart alarm system, etc...*). Questi costituiscono un'intera categoria di prodotti.
- **Sfida:** connettere dispositivi con forti limitazioni (*strong constraints*). Si pensi a RAM, batteria, consumo energetico, transmission range, ...

Ci interessa capire come certe cose vengono fatte oggi e in quale direzione andremo in futuro. Esiste un mercato, ma c'è una questione (architetture verticali VS architetture orizzontali).

1.2 Tipologie di sistemi

1.2.1 Architettura tipica: sistemi verticali

La maggioranza dei sistemi (il 90%) sono caratterizzati dalla stessa architettura (che definiamo "verticale")



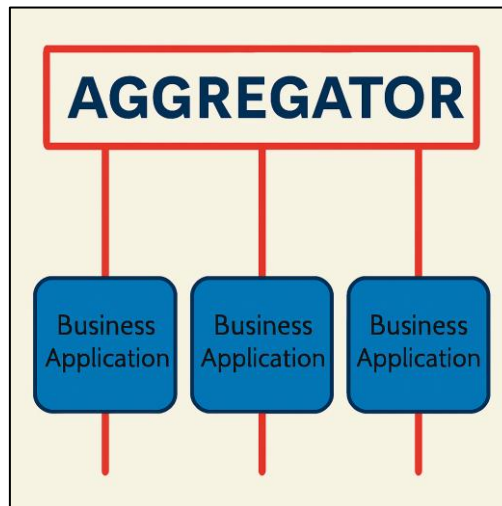
- Abbiamo acquistato e installato nella nostra casa uno *smart thermostat*. Tale dispositivo viene connesso a Internet.
- Dall'altra parte di Internet abbiamo lo *smart thermostat producer*, che sviluppa e istanzia l'applicazione per mezzo di un'infrastruttura cloud. Implementa una *business logic*.
- Questa cosa avviene per ogni smart device disponibile sul mercato.

- **Perché vertical systems?**

Tutte le componenti sono sviluppate con un unico scopo (*single purpose*).

- Prendiamo ad esempio lo smart thermostat: l'infrastruttura del producer/manufacture è in grado di comunicare unicamente con gli smart thermostat dello stesso producer e non con altri dispositivi (Esempio: *smart thermostat* di altri produttori). Un sistema non aperto.
- Detta in altre parole: *they're in pair* (smart device e infrastruttura).
- In alcuni casi si introduce (nel sistema) una componente detta **aggregator**, che permette una forma limitata di integrazione tra dispositivi di altri manufacturers.
 - Raro
 - Forma limitata e non completa di integrazione.

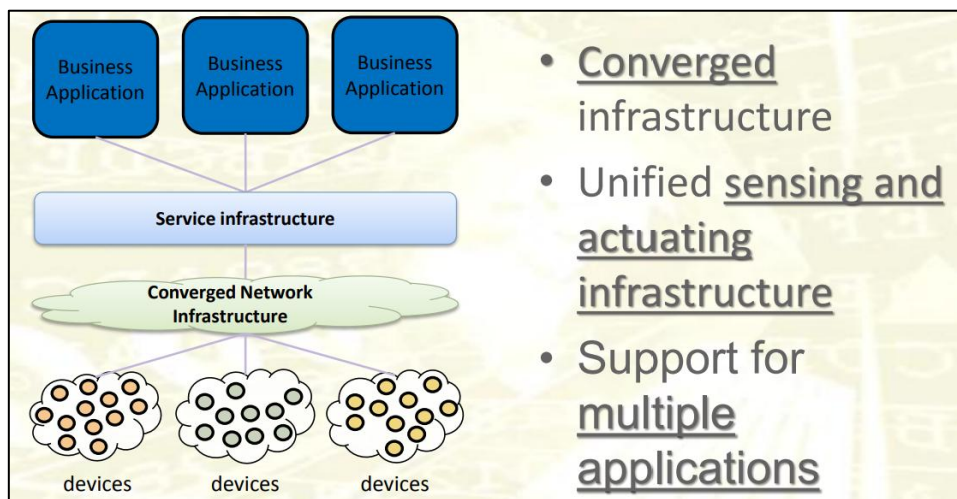
- **[Da domande]** Alexa è un esempio di *aggregator*: è un framework che integra IoT systems di manufacturer differenti. Questa cosa è implementata al di sopra della *business logic*, al di sopra dell'applicazione!
 - L'applicazione non ha ovviamente pieno controllo dell'aggregator e soprattutto non può dialogare/ottenere informazioni da altre applicazioni!
 - Detta in altre parole: il vantaggio dell'integrazione è fruibile soltanto dagli utenti, che utilizzano il dispositivo per interagire con molteplici sistemi. Le applicazioni continuano ad operare in modo indipendente.



1.2.2 Architettura ideale (Goal/Vision of IoT): sistemi orizzontali

1.2.2.1 Idea

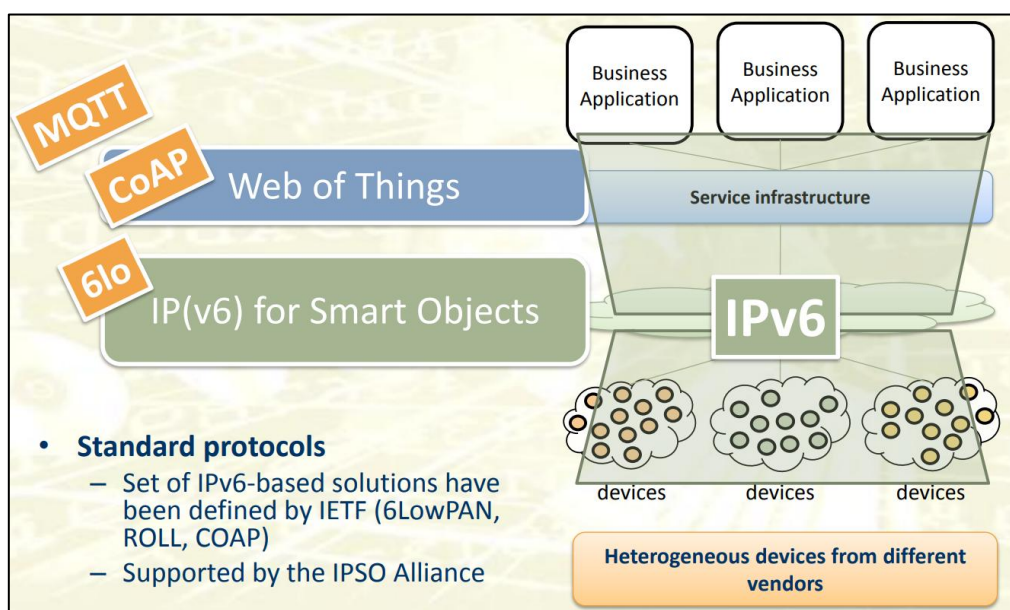
L'alternativa a un insieme di sistemi verticali isolati è una *converged infrastructure* dove adottiamo un approccio orizzontale.



- Immaginiamo un ambiente: la nostra casa. Molteplici *Smart Device* sono installati all'interno della nostra abitazione.
- L'approccio orizzontale prevede una *converged infrastructure* di cui fanno parte i dispositivi.
 - o I dispositivi utilizzano la stessa *converged infrastructure*.
 - o I dispositivi non sono associati a una particolare applicazione.
 - o Detta in altre parole: abbiamo un sistema aperto.
- Alcuni esempi:
 - o Installazione di applicazioni da un marketplace (si pensi al nostro telefono)
 - o Sensori sul dispositivo mobile. Questi sensori possono essere utilizzati da alcune applicazioni e non da altre, il comportamento dei sensori cambia in base all'applicazione eseguita.
- Altro aspetto interessante: flessibilità. Pensiamo a una *smart alarm application*:
 - o abbiamo un elenco di sensori obbligatori (nel senso che senza questi sensori l'applicazione non può funzionare);
 - o possono esistere dispositivi opzionali che supportano ed espandono le funzionalità delle applicazioni;
 - o l'installazione di questi dispositivi e di queste applicazioni può avvenire anche dopo il *deployment*.

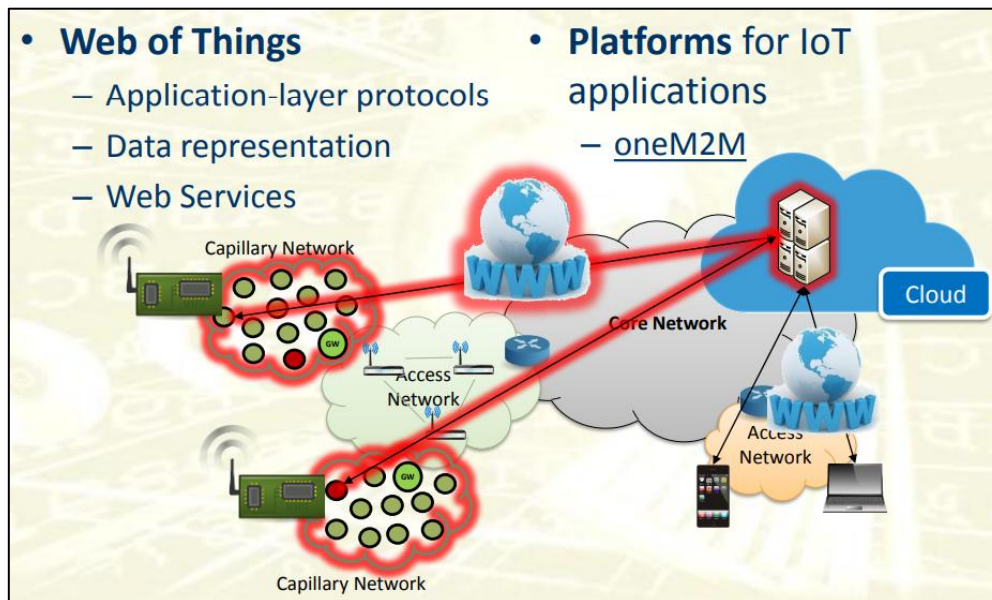
1.2.2.2 Come otteniamo sistemi orizzontali e converged infrastructure?

Come otteniamo le cose che abbiamo appena descritto?



- Necessario standardizzare protocolli e procedure per dispositivi IoT. Cosa tipica che faremo molte volte: riprendere protocolli esistenti e applicarli al nuovo contesto IoT con le dovute modifiche.
- La parola chiave è interoperabilità.
 - o Pensiamo al Web: vogliamo fare la stessa cosa con *smart devices* prodotti da manufacturer differenti.
 - o Esposizione di interfacce/funzionalità, che sono note tra i dispositivi in modo da garantire l'interoperabilità. Ciò renderà più semplice la comunicazione tra dispositivi eterogenei.
 - o IPv6 riveste un ruolo cruciale (*universal communication protocol*) come in Internet. Si tenga a mente che IPv6 è nato in un contesto completamente diverso, ergo si introduce una versione per Smart Objects,
 - o Sopra IPv6 abbiamo il cosiddetto "Web of Things": un set di protocolli che garantiscono che i dispositivi espongano un'interfaccia comune e sia possibile quindi l'interoperabilità tra dispositivi. Occhio in particolare a CoAP e MQTT.

Poniamo il focus sull'architettura che abbiamo già visto qualche pagina fa.



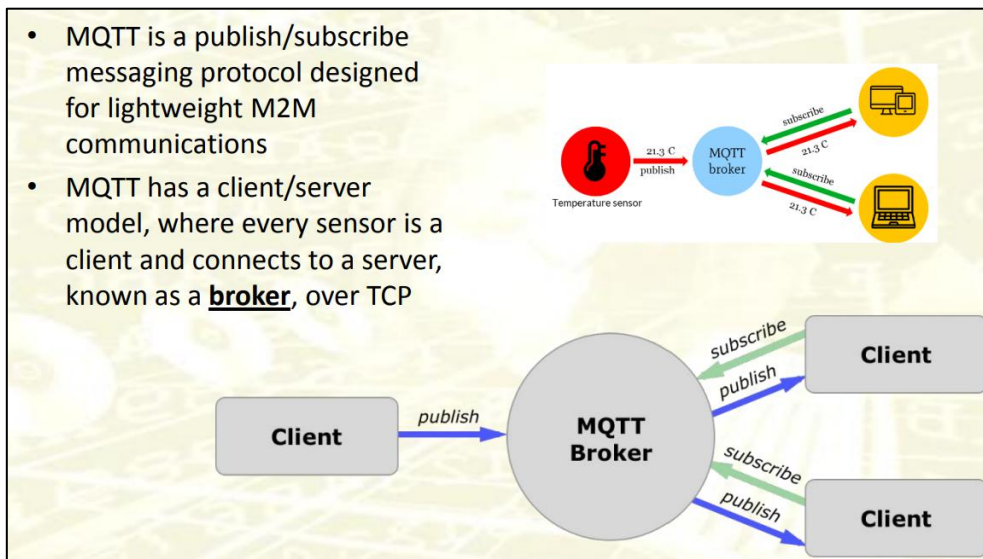
- Abbiamo dispositivi IoT caratterizzati da Networking capabilities che permettono ad essi di essere collegati a Internet.
- Il deployment delle applicazioni avviene *on top of cloud platforms*. Questo avviene col 90% delle applicazioni!
 - **Low costs.**
I costi sono decisamente più bassi rispetto a quelli presenti in forme tradizionali di computing.
 - **Scalability.**
Se la nostra attività cresce saremo in grado di gestire il crescente numero di richieste in modo rapido. *More resources will be allocated to the application, so the application can scale out proportionally.*
- **Fun fact.**
Praticamente tutte le soluzioni disponibili sul mercato sono solitamente basate sulle stesse tecnologie. Questo ci permette di dire che l'horizontal IoT è nei fatti già realtà!
 - Tutti i **Cloud Provider** offrono **native IoT support** agli **IoT vendor**. I Cloud Provider forniscono un *package* che permette la creazione di un nuovo IoT system "from scratch" (si veda poco più avanti).
 - Amazon WS, Microsoft Azure, Google Cloud, ...
 - Il *package* è fornito nella forma di servizio (ricordare cosa si intende con servizio, dalle dispense di Cloud Computing), si forniscono "programmi scheletro" che possono essere eseguiti sui dispositivi IoT. Tutto questo permette la creazione di applicazioni IoT in modo rapido.
 - Tra le tecnologie comunemente adottate si ha un protocollo (*a common and interoperable protocol*): MQTT, acronimo di *Message Queuing Telemetry Transport*.

1.3 MQTT (Message Queuing Telemetry Transport) Protocol

1.3.1 Introduzione

Introduciamo il protocollo MQTT, che è un protocollo publish/subscribe creato prima dell'emergere dell'IoT.

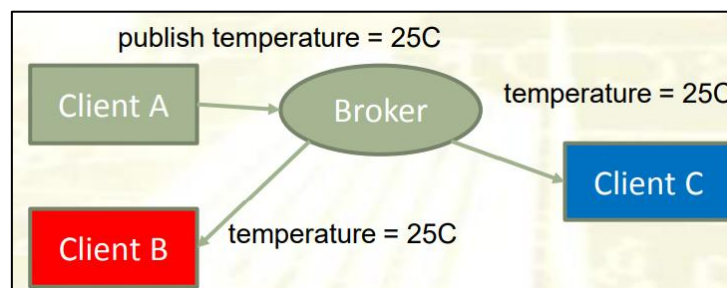
- **Scopo iniziale di MQTT.**
Protocollo *application layer* pensato per raccogliere dati da sensori distribuiti su oleodotti (di grande dimensione): perdite, quantità di olio presente, ...
- **Complessità dietro lo scopo iniziale.**
Necessario tenere conto dei *constraints*: tecnologie che permettono una trasmissione di pochi KB/s, dispositivi di piccola dimensione, dispositivi collocati in luoghi difficili da raggiungere...



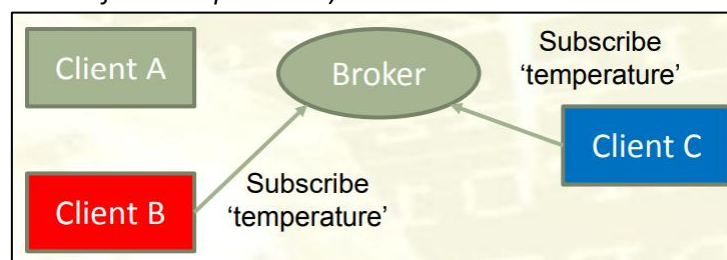
- Da una parte abbiamo i client, che "producono" o "consumano" messaggi (con dati al loro interno).
- Dall'altra abbiamo il cosiddetto MQTT Broker, che è il server responsabile per ricevere e inoltrare messaggi (*dispatcher*). Il Broker è centrale nel protocollo MQTT.
- Basato sul protocollo TCP (garantita *reliability*).

Il protocollo prevede due operazioni, lanciabili da qualunque *client* presente nella rete:

- **Publish.** Ogni *client* che vuole trasmettere dati "pubblica" i dati, trasmettendoli al MQTT Broker. Pensiamo a un sensore di temperatura: il client "pubblica" la temperatura appena captata dal relativo sensore.



- **Subscribe.** Ogni *client* interessato a ricevere certi dati contatta il MQTT Broker e segnala l'interesse a ricevere tali dati (i dati sono identificati da un topic). Ogni topic è identificato univocamente da una stringa (esempio: "*notifiche temperature*").



Cosa intendiamo con client?

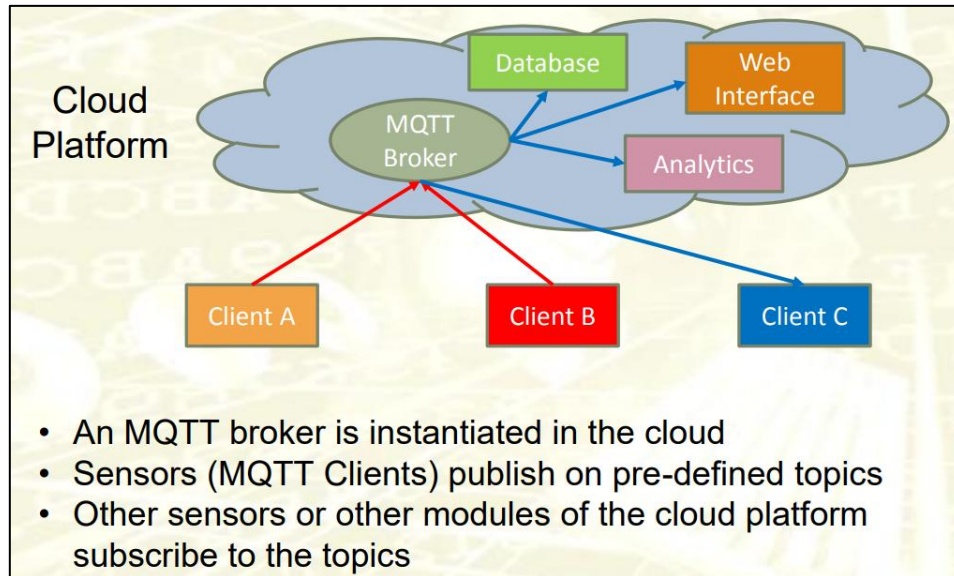
- Un sensore che produce dati
- Un sensore che ha bisogno di ricevere dati da altri sensori.
- Un'applicazione (che per poter funzionare deve ricevere periodicamente i dati).

Ricapitoliamo:

- Un client capta una temperatura, la trasmette al MQTT Broker (*publish operation*) sotto il topic "*notifiche temperature*". Il topic è uno dei metadati del messaggio trasmesso.
- Il MQTT Broker recupera la lista dei client che hanno sottoscritto il thread "*notifiche temperature*" (ciascuno lo ha fatto con la *subscribe*). La temperatura viene trasmessa ai nodi presenti nell'elenco.

1.3.2 MQTT-based Cloud Platforms

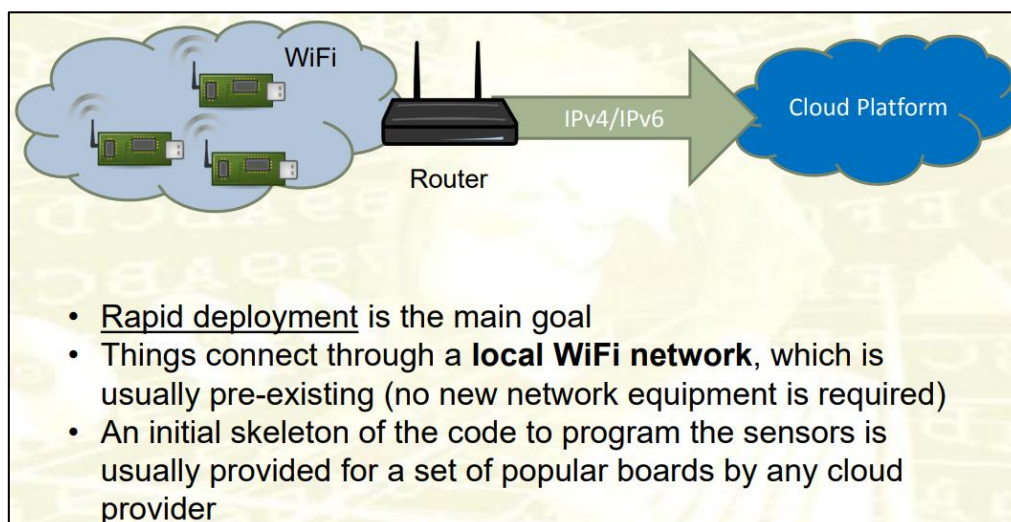
Come è utilizzato MQTT nel Cloud Package fornito da un generico Cloud Provider?



- Il Cloud Provider solitamente istanzia per ogni customer (per ogni IoT vendor) un MQTT Broker sulla Cloud Platform.
- Il MQTT Broker è utilizzato dai sensori (Client A, Client B, Client C), che interagiscono con esso per mezzo delle operazioni *publish* e *subscribe*.
- Presenti molteplici applicazioni sulla Cloud Platform:
 - database per memorizzare dati;
 - *web interface* per restituire i risultati (dai dati) agli utenti;
 - Analytics per svolgere "fancy predictions" (Nel nostro esempio: predire temperature).
- Abbiamo detto che il Cloud Provider istanzia il MQTT Broker. Per quanto riguarda Database, Web Interface e Analytics devono essere sviluppati dall'IoT vendor (application logic).
 - La cosa preoccupa? Neanche troppo, il Cloud Provider fornisce scheletri di applicazione (in questo modo l'IoT vendor non deve sviluppare qualcosa *from scratch*).

1.3.3 Rapid prototype-to-product time: centralità della WiFi

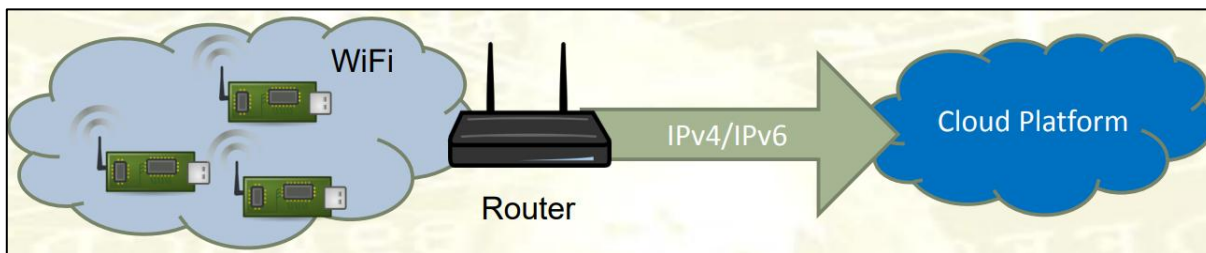
Stiamo ponendo un focus molto forte sul *rapid deployment*: vogliamo minimizzare il tempo necessario per introdurre l'IoT device sul mercato.



Contribuisce in modo significativo l'uso della connessione WiFi, che è onnipresente e disponibile praticamente in ogni abitazione.

1.4 Vantaggi e svantaggi di una Cloud Platform

Riepiloghiamo i vantaggi nell'uso del Cloud nella seguente tabella:



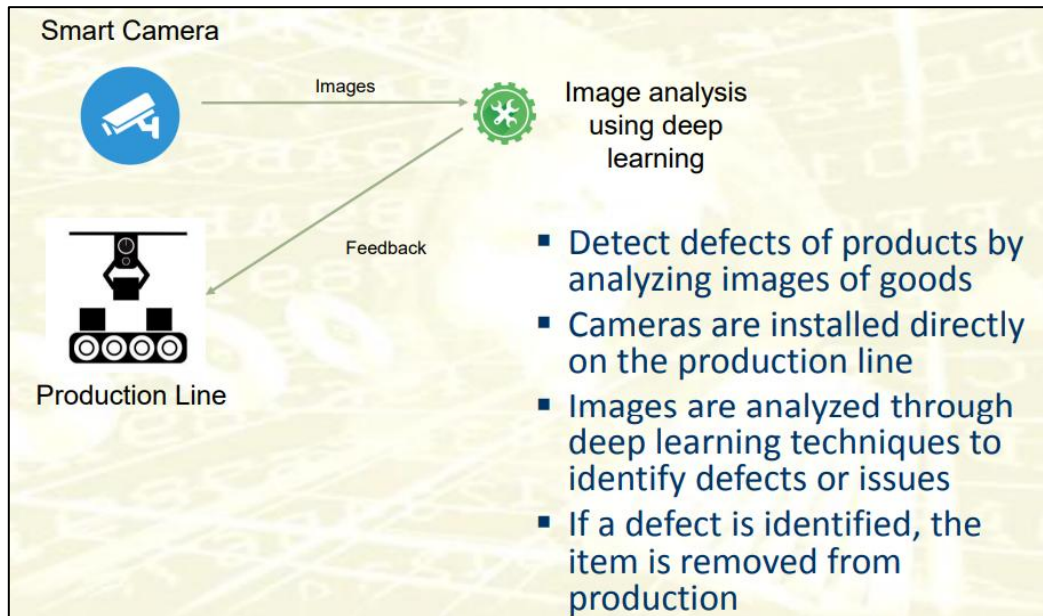
Vantaggi	Svantaggi
- Il deployment è semplice e rapido. - o I Cloud Provider forniscono packages con <i>ready-to-use solutions</i>. - o Semplicità anche per l'utente, dato che si utilizza la rete WiFi. - Non è necessaria una nuova infrastruttura (in particolare per la connettività). Non servono nuovi gateway, router, o altri dispositivi. - Tutti i vantaggi dell'infrastruttura Cloud sono presenti, in particolare la scalabilità. Possibile gestire aumento del numero di richieste con costi non elevati.	La Cloud Platform è coinvolta in ogni comunicazione, non è possibile neanche una comunicazione diretta tra dispositivi presenti nella stessa rete locale. - Segue non supporto per applicazioni che richiedono bassa latenza (il fatto che il pacchetto debba attraversare Internet per raggiungere la Cloud Platform e successivamente tornare indietro aumenta i tempi in modo importanti, ms). The diagram in the 'Svantaggi' section shows a similar setup to the first one, but with a red 'X' over the local network cloud. A red arrow labeled 'IPv4/IPv6' points from the Router to the 'Cloud Platform' cloud. A blue double-headed arrow labeled 'Internet' connects the Router to the Cloud Platform, indicating that all communication must pass through the Internet, even for local devices.

1.5 Esempi di industrial use cases

1.5.1 Smart Manufacturing

Supponiamo di avere delle fotocamere smart che fotografano i prodotti presenti sulla linea di produzione.

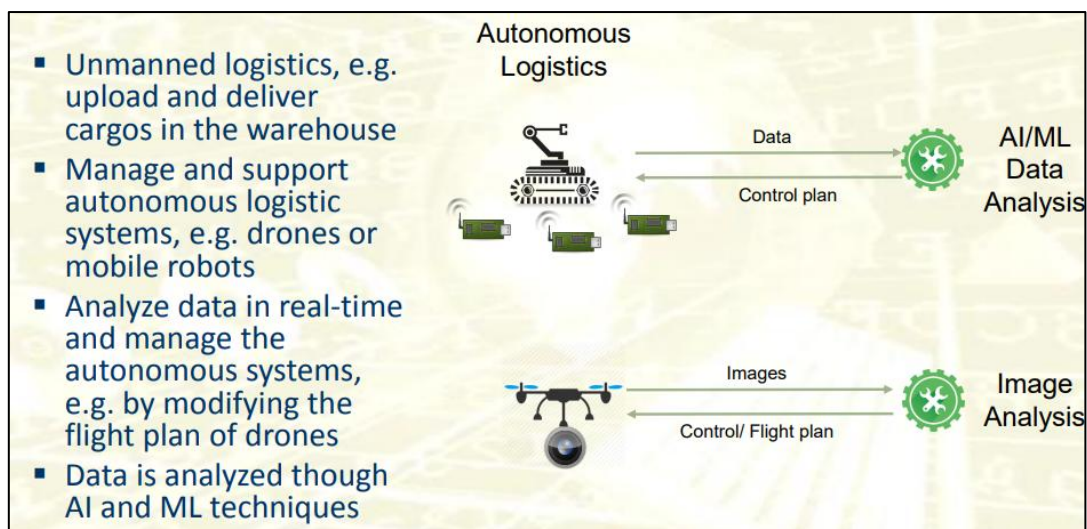
- Vogliamo utilizzare queste foto per individuare difetti di produzione sui prodotti presenti sulla linea: lo facciamo utilizzando tecniche di deep learning.
- In caso di individuazione di difetti (non rispetto degli standard qualitativi) viene trasmesso un feedback a un robot.
- Il robot rimuove il prodotto dalla linea di produzione dopo aver ricevuto il feedback.
- Si cerca di intervenire e rimuovere il prodotto prima che sia troppo tardi.



1.5.2 Autonomous Robot

Supponiamo di avere un impianto di produzione dove robot spostano oggetti da un punto a un altro della linea di produzione. In alternativa possiamo pensare addirittura a droni.

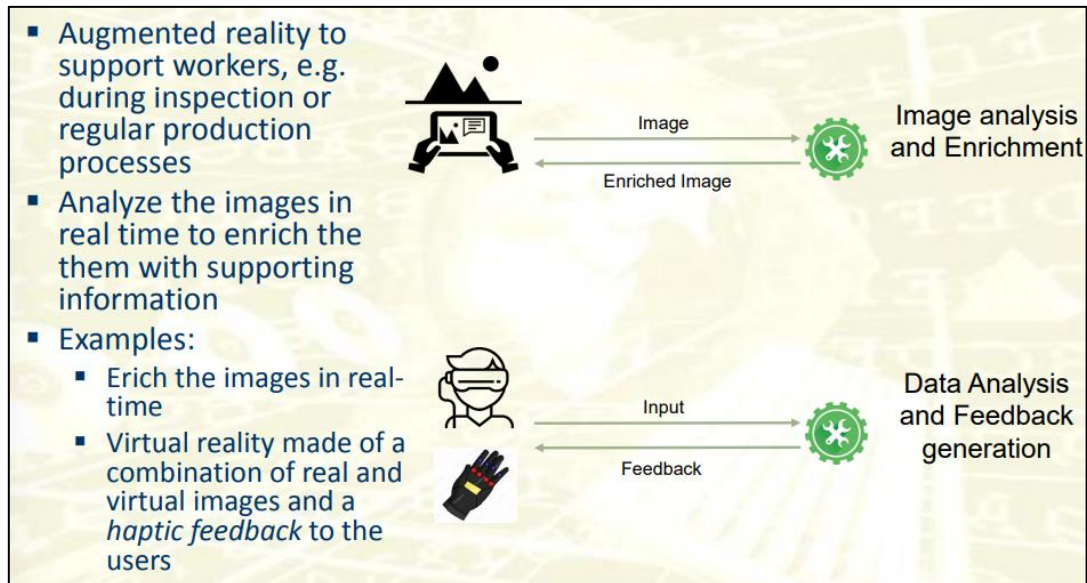
- Sono dotati di sensori che captano dati. Tali dati sono inviati a un'applicazione che svolge analisi con tecniche di Machine Learning. Dopo lo svolgimento dell'analisi un comando viene trasmesso al robot, che agisce di conseguenza.
 - o Attenzione alla latenza, il comando potrebbe essere trasmesso troppo tardi.
- I dati possono essere immagini!



1.5.3 Augmented Reality

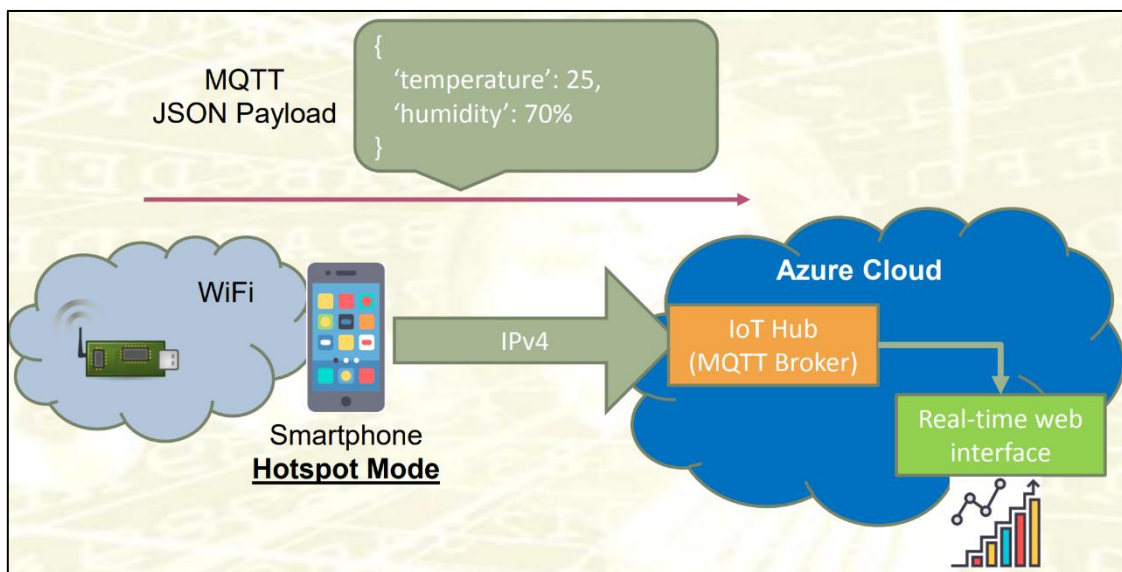
Un altro esempio è la realtà aumentata! Immaginiamo di avere un operatore (all'interno dell'impianto) che sta utilizzando un tablet/device per gestire le sue operazioni (ad esempio facendo fotografie).

- Le immagini sono trasmesse per "*Image analysis and enrichment services*". Con enrichment si intende associare alle immagini ulteriori informazioni utili per l'attività dell'operatore. Le informazioni sono inviate al dispositivo dell'operatore.



1.6 Demo IoT Product

Il professore ha mostrato una demo di un *IoT Product* molto semplice.



Sono coinvolte le seguenti componenti:

- **Microsoft Azure**
Si utilizza l'IoT package fornito da Microsoft Azure. Sottoscrizione all'IoT package da parte del docente, conseguente istanziamento dell'IoT Hub (MQTT Broker) da parte del Cloud Provider (Microsoft). E' stato utilizzato lo scheletro per una interfaccia web che mostra in tempo reale i dati trasmessi dal dispositivo.
- **ESP32**
ESP32 è una delle più popolari *IoT platform* disponibili oggi sul mercato. Il 90% circa dei prodotti IoT provenienti dalla Cina sono basati su questo microcontrollore.
 - o Sviluppato da *espressif* (chinese company)

- Vantaggio principale è il costo: 2/5 euro (per il produttore addirittura meno dato che si acquista in grandi quantità)
- Dual Core microprocessor (cosa non particolarmente comune, i Dongle dati dalla Righetti sono monoprocesso)
- Supporto a WiFi e Bluetooth (adottato BLE).
 - Il consumo energetico dipende dalla tecnologia adottata per comunicare.
 - Se si adotta WiFi il consumo energetico aumenta in modo significativo, ragione per cui ESP32 non è in generale utilizzato su dispositivi alimentati da batteria.
- Supportato dai principali Cloud Provider: Azure, Amazon WS, ...

- **Rapid prototyping IoT platform**
- **Ultra low cost: board 5 Euros, Chip 2 Euros**
- **Rich specs:**
 - RAM 520 Kbyte
 - Dual Core microprocessor
 - Ultra low power design
 - Cryptographic hardware
 - Wi-Fi: 802.11 b/g/n
 - Bluetooth: v4.2 BR/EDR and BLE
- **It supports the Arduino framework (simple programming) or FreeRTOS (a complete operating system for embedded systems)**

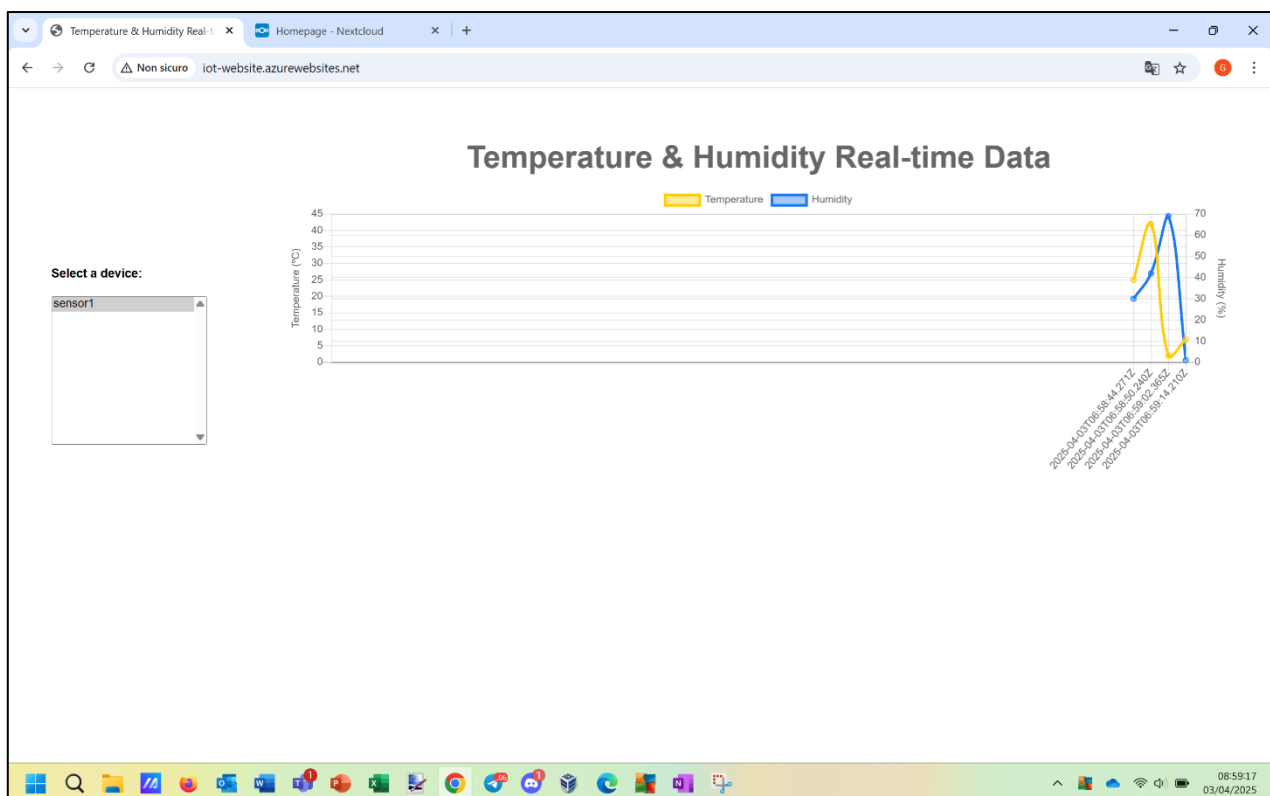




ESPRESSIF
<https://www.espressif.com/>

Supported by many cloud platforms, i.e. Azure, Amazon WS, etc.
Many commercial products already use this platform

Informazioni su ESP32



Anteprima della web interface

1.7 Fog computing

1.7.1 Questioni precedenti

Abbiamo già posto nelle pagine precedenti gli aspetti critici riguardanti l'utilizzo di una Cloud Platform.

- **Architettura estremamente centralizzata**

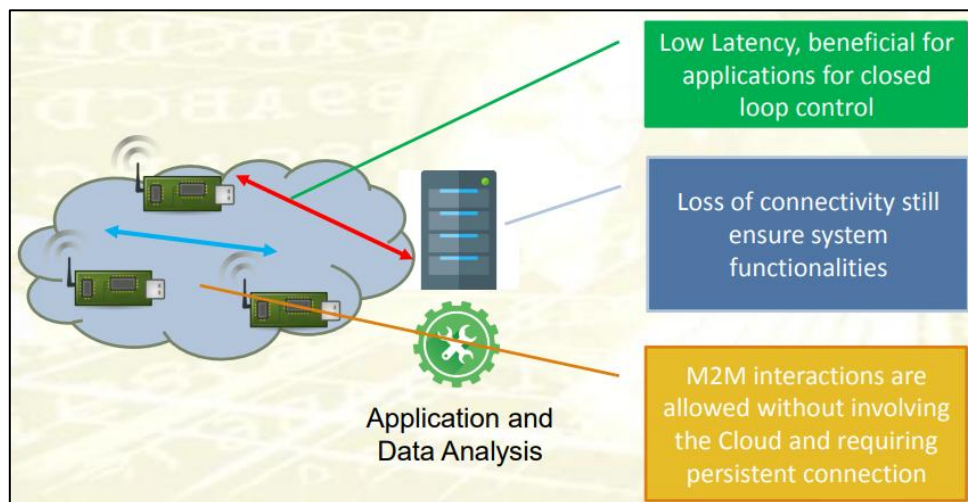
L'architettura descritta è estremamente centralizzata: va bene nel 90% dei casi, non nel rimanente 10%. Il problema di fondo è che il MQTT Broker è sempre coinvolto, con conseguente aumento della latenza. Non è possibile interazione diretta tra nodi (anche nodi presenti nella stessa rete locale) e deve essere garantita la connettività.

- Applicazioni che richiedono *reliability* e *low latency* non sono compatibili con l'approccio precedentemente introdotto.
- Esempio con lo use case di Augmented Reality. In caso di network outage la linea di produzione continua a funzionare, ma l'applicazione no e pertanto l'operatore non può continuare a supervisionare l'impianto (con conseguenze economiche).

- **Proposta: local Application?**

Con requisiti particolarmente stringenti potrebbe essere utile un ritorno a un approccio tradizionale e in locale: il deployment è in locale e non sull'infrastruttura del Cloud Provider. Risolviamo le questioni sollevate precedentemente?

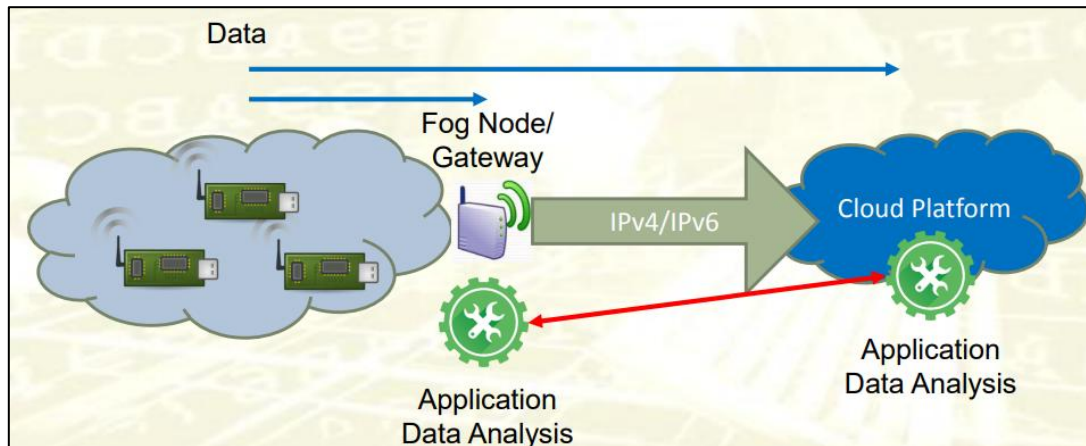
- **Requisito di bassa latenza.**
Riduciamo sicuramente la latenza.
- **Requisito di high reliability.**
Garantiamo il funzionamento del sistema in caso di perdita di connessione.
- **Comunicazioni machine-to-machine.**
Diventa possibile stabilire comunicazioni *machine-to-machine* (questo solo se cambiamo l'architettura e non ci limitiamo semplicemente a spostare il Broker dalla Cloud Platform alla rete locale).



Ovviamente fare questo implica non godere dei benefici del Cloud Computing: è chiaro che la soluzione migliore sta nel mezzo. Introduciamo il *Fog Computing*!

1.7.2 Introduzione

Riportiamo in locale (vicino agli utenti e ai dispositivi IoT) parte della logica computazionale presente nel datacenter, ma non interamente (non vogliamo rinunciare ai benefici del Cloud, in particolare i costi minori).



Idea:

- La maggioranza delle risorse è mantenuta nel datacenter.
- Si introduce un ulteriore *computing layer* detto *fog layer* costituito da dispositivi (detti *fog nodes*) che supportano *high reliability* e *low latency* nella comunicazione. Questo *fog layer* è vicino all'utente e ai dispositivi IoT, in contrasto alla *cloud platform* che è lontana.
 - **Definizione di fog node da ChatGPT:** *Un fog node è un dispositivo fisico o virtuale (come un router intelligente, un gateway, un server locale, o anche un computer) che elabora i dati localmente, invece di inviarli subito a un data center nel cloud.*
 - Sono possibili molteplici *fog layers*.
- **Cosa cambia?**
 - Il *fog node* è ospitato nella stessa rete locale dei dispositivi IoT, o comunque in una rete "vicina".
 - Si offre *low latency* e *resiliency* per quel 10% di applicazioni che lo richiedono.
 - Si garantisce anche *privacy* (se il fog node è nella rete locale allora le informazioni non girano su reti esterne).
 - In base ai requisiti possiamo stabilire l'esecuzione della logica dell'applicazione nel fog node e/o nel datacenter.
 - Il Cloud è sempre lì. Se abbiamo un'applicazione che lavora con big data e che non richiede *low latency* e *reliability* possiamo sempre fare riferimento al Cloud.
 - Industrial example: *predictive maintenance*.

Non è una semplice idea: questi dispositivi sono già sul mercato! Attenzione all'aspetto hardware:

- sono tipicamente dispositivi low-power;
- sono supportati da *AI/ML accelerators* per l'esecuzione di "*small models*" per l'analisi di dati (Esempio: immagini).

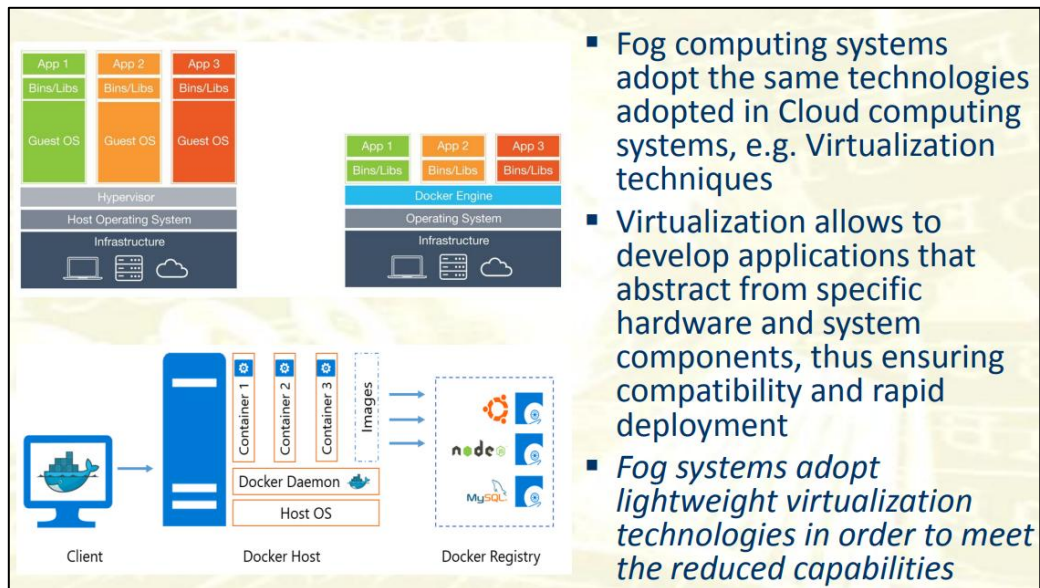
1.7.3 Tecnologie alla base del fog computing

Quali tecnologie sono alla base del Fog Computing? Le tecnologie che hanno reso il Cloud come lo conosciamo sono utilizzate per i *fog nodes*.

1.7.3.1 Virtualization

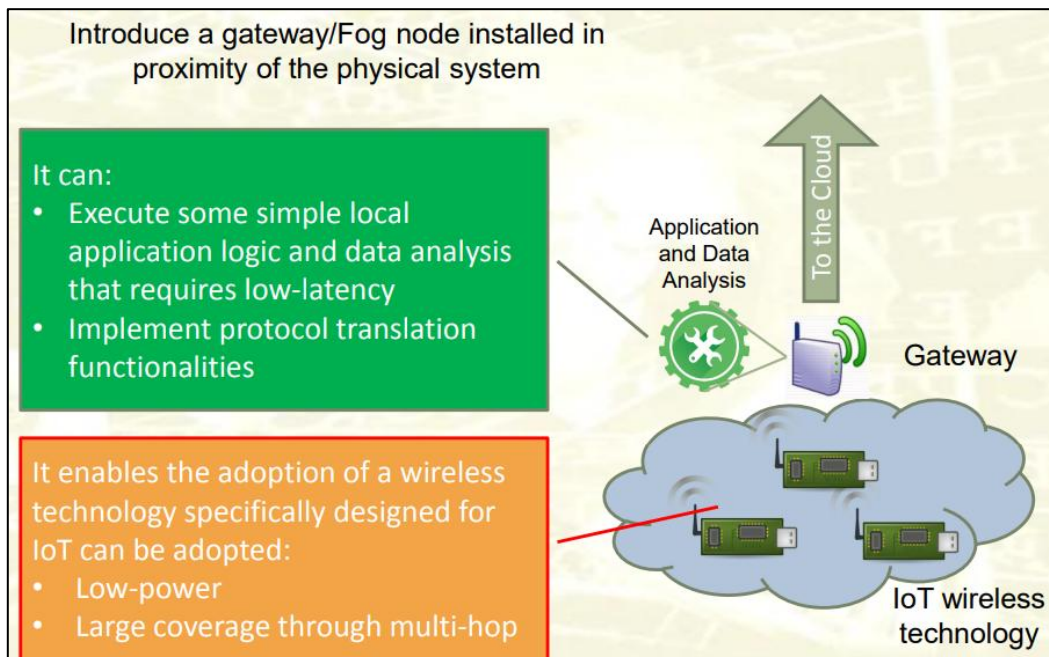
Una delle tecnologie adottate è la virtualizzazione, che permette la creazione di un ambiente astratto rispetto all'hardware.

- Fog/Edge devices sono collocati in prossimità dei dispositivi IoT
- Su questi dispositivi le applicazioni sono eseguite utilizzando tecniche di virtualizzazione.
- Attenzione: ribadiamo che i fog devices sono caratterizzati da risorse fisiche inferiori (CPU, RAM, Storage). Per tale ragione si adottano forme di *lightweight virtualization* (in particolare i container).



1.7.3.2 Wireless Technologies

Un semplice spostamento dell'*application logic* non è sufficiente: pensiamo al ruolo centrale del WiFi, già ribadito più volte. I *fog device* sono un'opportunità per introdurre tecnologie wireless IoT friendly ed efficienti dal punto di vista energetico (quelle introdotte dal prof. Anastasi - cit.).



1.7.3.3 Da MQTT a Web of Things

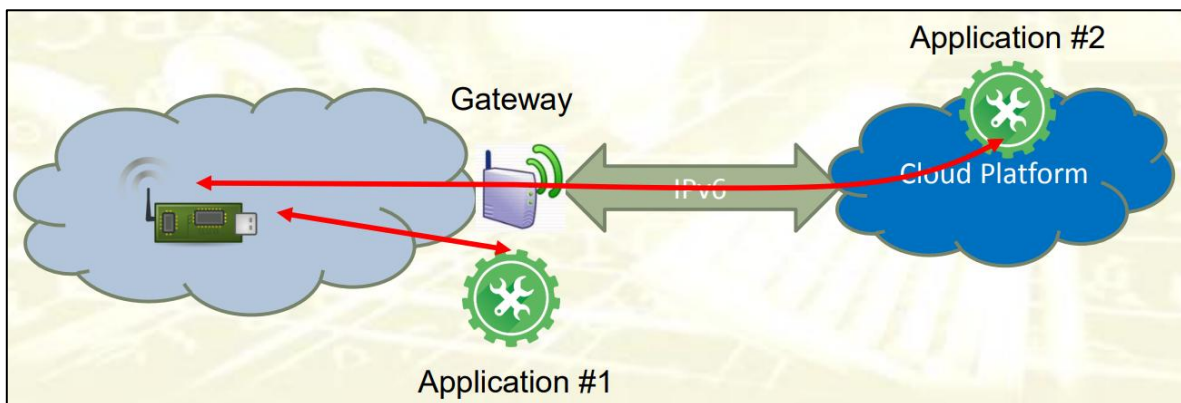
Riprendiamo il protocollo MQTT. Poniamoci le seguenti domande:

1. Possiamo preservare MQTT nel contesto di un'architettura multi-layer caratterizzata da *fog devices*?
2. In caso affermativo, dove collochiamo il MQTT Broker?
 - Nella *Cloud Platform* o nel *fog node*?
 - Porlo nella Cloud Platform significa rendere incompatibile il tutto con applicazioni che richiedono *low latency*.
 - Porlo nel fog node/gateway significa che i dati sono gestiti da dispositivi con una quantità limitata di risorse fisiche. Questo può risultare in problemi dal punto di vista della scalabilità, e quindi in bottleneck.
 - In entrambe? Un Broker nella Cloud Platform e uno nel fog node
 - Quando si fa publish su broker e quando su un altro? Difficile da gestire...

Abbiamo difficoltà nel rispondere alla seconda domanda, ergo possiamo dedurre che il protocollo MQTT non sia particolarmente adatto per la tipologia di architettura che stiamo analizzando.

MQTT → Web of Things

Risolviamo introducendo un nuovo paradigma per la comunicazione tra applicazioni e dispositivi IoT: il **Web of Things**, spostiamo il centro del nostro sistema dai broker ai dispositivi IoT (cit.)!



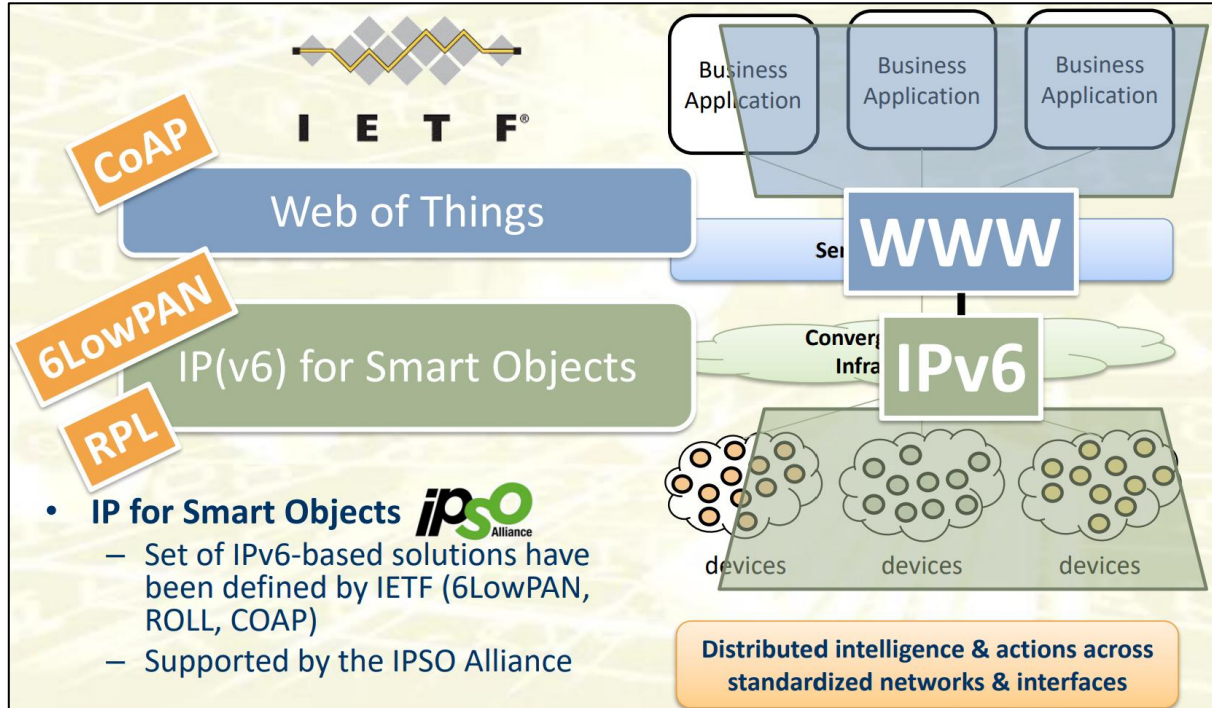
- L'idea è di esporre i dispositivi IoT alle applicazioni esattamente come in un web server: se l'applicazione necessita di dati contatta "personalmente" i dispositivi IoT e richiede tali dati (esattamente come un browser che richiede un sito web a un web server).
 - I dispositivi IoT sono esposti sulla rete Internet.
 - L'applicazione #1 (sul gateway) contatta il dispositivo IoT utilizzando il relativo indirizzo IPv6.
 - L'applicazione #2 (sulla Cloud Platform) contatta il dispositivo IoT nella stessa maniera!
 - Non ci interessa dove è collocata l'applicazione, possiamo muoverci dal gateway alla cloud platform e viceversa senza cambiare l'implementazione.
 - Perché abbiamo detto che "spostiamo il centro del nostro sistema"?
 - Nel modello precedente si passa sempre dal Broker
 - Nel nuovo modello si passa sempre dai dispositivi IoT (richieste fatte dall'applicazione)
- Se non ci sono applicazioni che richiedono dati non avvengono trasferimenti di dati (in contrasto col modello precedente dove il dispositivo produce dati). Sostanzialmente il dispositivo lavora solo quando effettivamente necessario (cosa che è profondamente *energy efficient*).
- Approccio che permette un'interazione *machine-to-machine*: i dispositivi IoT espongono tutte le stesse interfacce, che possono essere invocate da altri sensori e non per forza dall'applicazione!

2 WEB OF THINGS AND REST SYSTEMS

2.1 Introduzione

Abbiamo già introdotto il concetto di architettura orizzontale (*converged infrastructure*), che abbiamo detto essere ciò a cui ambiamo. Come possiamo realizzare questa infrastruttura?

Riprendiamo un'immagine già vista in spiegazioni precedenti.



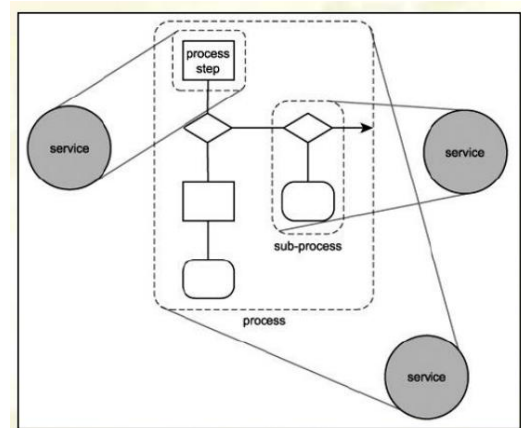
Ci muoviamo adottando quanto segue:

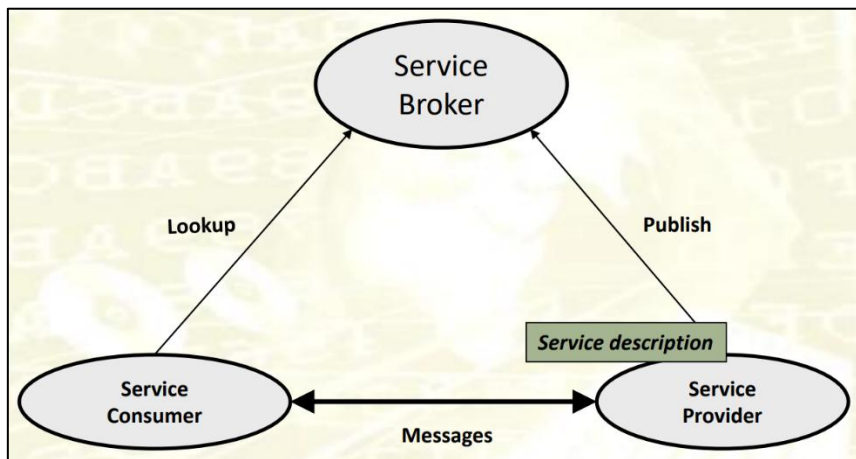
- **Network protocol**
Adottiamo *IPv6 for Smart Objects* (come vedremo con l'ing. Righetti).
- **Application protocol**
Adozione di tecnologie inizialmente pensate per il Web (e non per l'IoT).
 - o Passaggi simili rispetto a quelli già visti nelle spiegazioni precedenti: si prendono a riferimento tecnologie già esistenti pensate per altri scopi (come fatto con MQTT, che poi abbiamo scartato prendendo atto dei suoi limiti dopo aver introdotto il fog computing).

2.2 Service-oriented architecture

Le tecnologie Cloud sono state introdotte recentemente: ciò ha richiesto nuove metodologie per organizzare le applicazioni basate sul cloud. Il risultato di queste riflessioni è stata la *service-oriented architecture*.

- L'applicazione, che inizialmente era pensata come un'unica struttura monolitica, vede le sue funzionalità frammentate in servizi.
- Il servizio è un modulo che implementare parte della logica dell'applicazione e che funziona in modo indipendente dagli altri servizi.
- L'intera applicazione consiste nella composizione di molteplici servizi.
- Si garantiscono *maintenability* ed *expandability*. La composizione in servizi agevola il re-design dell'applicazione.





- I servizi dialogano tra di loro scambiandosi messaggi.
 - Il servizio è esposto per mezzo di un'interfaccia.
 - Il servizio è accompagnato da una *service description* che riporta il nome del servizio e il formato dei dati che il servizio si aspetta in ingresso.
- Ruoli:
 - **Service Provider**
Servizio che fornisce dati e/o funzionalità
 - **Service Consumer**
Servizio che usufruire dei dati e/o funzionalità fornite dal service provider
 - **Service Broker (non confondere con MQTT Broker)**
Servizio che mantiene una directory dei servizi disponibili nel sistema.

2.3 Ispirazioni per il Web of Things: World Wide Web

2.3.1 Introduzione

- **Origins (1989, CERN, Tim Berners-Lee)**
 - A system to easily share research documents
 - Three components
 - A mechanism to identify documents
 - A protocol to exchange documents
 - A language to encode data (allowing for non sequential access)
- **The Web today**
 - A world-wide platform for the exchange and management of generic data
 - A huge collection of multimedia data
 - An interface to access net applications
 - **A platform enabling generic service invocation in distributed computing systems (Web Services)**

Sul World Wide Web

- Inizialmente introdotto per permettere la condivisione di documenti tra ricercatori.
- Evolve nel tempo: non più solo per condividere documenti, ma anche per fornire servizi!
- Le Cloud Applications si basano sul paradigma *service-oriented* detto prima. Ci si pone una questione: possiamo semplificare le Cloud Applications adottando le tecnologie che supportano il World Wide Web? Si adottano gli stessi protocolli

Introduciamo i pilastri del World Wide Web per capire meglio come questi saranno utilizzati nella creazione di Cloud Applications.

2.3.2 Identification functionality: Uniform Resource Identifier (URI)

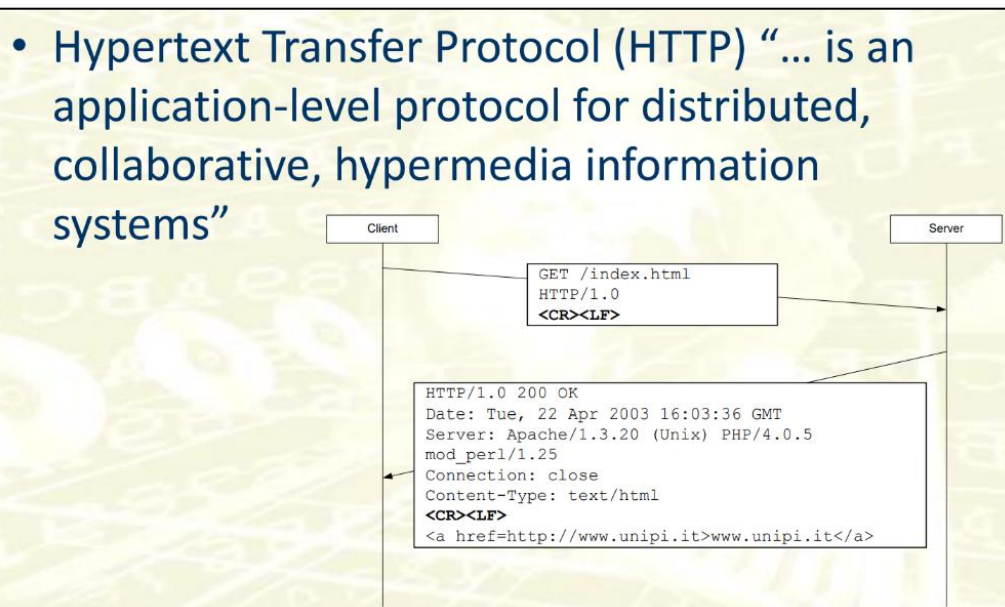
La prima componente è la funzionalità che permette l'identificazione delle risorse: serve un meccanismo che permette di stabilire la posizione della risorsa permettendone così il recupero (attraverso il browser).

- A Uniform Resource Identifier (URI) “is a compact string of characters for identifying an abstract or physical resource” [RFC 2396]
- Identifier
 - A string with a restricted syntax
- Resource
 - Any entity that has an identity
 - an electronic document, an image, a service, a collection of other resources
- Uniform
 - Different types of resources identified in the same context
 - Common syntactic conventions
 - Flexibility of extension

- Si pone una stringa di caratteri.
- Si possono identificare molteplici tipologie di risorse: immagini, video, etc...
- Standard alla base dell'URL (*Uniform Resource Locator*): sintassi (come deve essere costruita la stringa), semantica (come deve essere interpretata).
 - URL è una sottocategoria di URI che permette esplicitamente la localizzazione della risorsa (e non semplicemente l'identificazione).

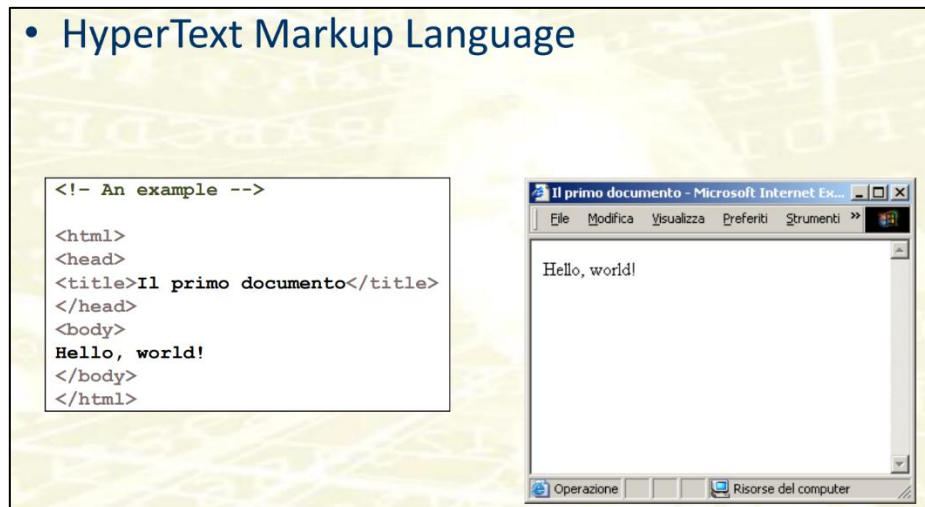
2.3.3 Data exchange protocol: Hypertext Transfer Protocol (HTTP)

La seconda componente è un protocollo a livello applicazione che regola lo scambio di dati tra un client e un server: HTTP. Protocollo pensato per il web (dialogo tra client e web-server per caricare sul browser una pagina web, si riporta nel payload il contenuto della pagina web), adottato nel Cloud Computing.



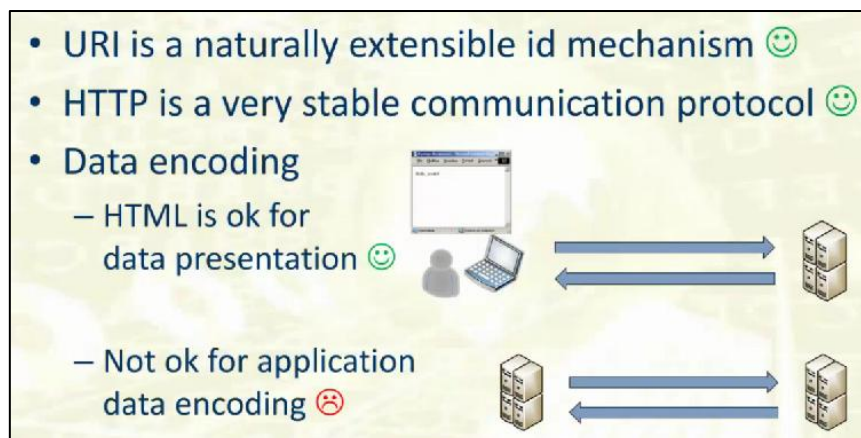
2.3.4 Data encoding system: HyperText Markup Language (HTML)

Le informazioni fornite dai server possono essere di varie tipologie: per favorire l'interoperabilità e la comprensione da parte di tutti i client è necessario definire un *data encoding language*. Tale linguaggio (standardizzato, quindi interpretato allo stesso modo da tutti i client) permette l'inclusione di dati/oggetti all'interno di un documento.



2.3.5 Evolution to Web Services: URI e HTTP, XML al posto di HTML

Ci vogliamo muovere da uno scenario dove abbiamo un server che offre le pagine web e un browser che le recupera a uno scenario dove abbiamo un server che offre funzionalità/dati, ma non per una pagina web (*anything* - cit.). Le informazioni/funzionalità offerte sono utilizzate da un servizio eseguito su un'altra macchina.



Riepiloghiamo i pilastri del World Wide Web:

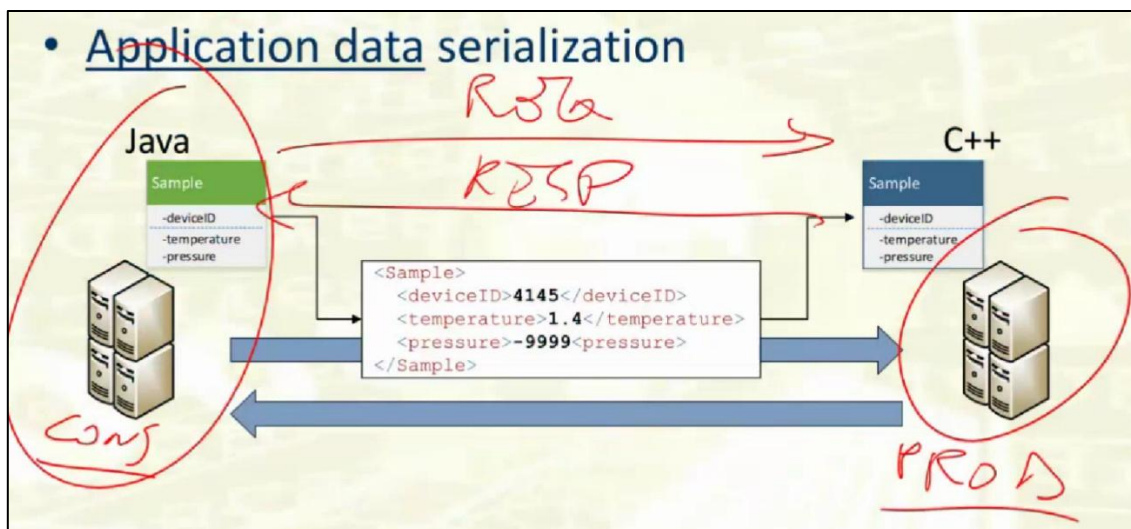
- **URI [Identification functionality]**
Pensato per essere espandibile e modificabile in modo semplice. URI può puntare a una pagina web come a un'altra tipologia di risorsa che fornisce dati e informazioni.
 - Ok, possiamo usare senza modifiche!
- **HTTP [Data exchange protocol]**
Stabile, popolare e flessibile. Può essere utilizzato per pagine web, così come per trasportare generic data.
 - Ok, possiamo usare senza modifiche!
- **HTML [Data encoding system]**
Va bene per pagine web, ma non per applicazioni generiche
 - No, dobbiamo trovare un *data encoding language* alternativo.
 - Quale? **XML** (*eXpandable Markup Language*)! Viene introdotto come una generalizzazione di HTML: non solo encoding di pagine web, ma encoding di qualunque tipologia di dati.

- Markup metalanguage (W3C – 1st ed. Dic. 2000)
- Text-based encoding
- Rules for specifying markup languages (*XML applications*)
 - Common syntax
 - Grammar specification is also possible
- Why extensible?
 - The tag set is not prespecified

L'applicazione può essere scomposta in servizi, il servizio è implementato come un set di risorse.

Quindi. La soluzione è utilizzare un servizio che fornisce dati/funzionalità:

- il servizio è esposto per mezzo delle URI;
- Il servizio può essere invocato dal consumer utilizzando una HTTP REQ;
- il servizio risponde per mezzo di una HTTP RESP, dialogando col consumer per mezzo del payload;
- **tale payload è posto in XML.**



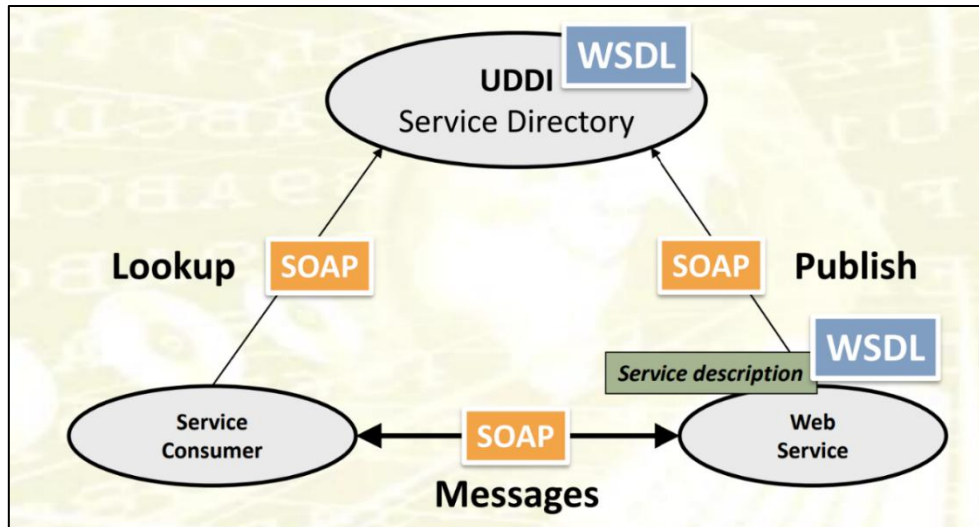
2.4 Proposta: implementazione dei Web Services con protocolli Cloud (SOAP e WSDL)

I protocolli utilizzati originariamente per IoT sono gli stessi visti per una *Service Oriented Architecture* nel *Cloud Computing*:

- **SOAP** (was Simple Object Access Protocol)
 - "lightweight protocol intended for exchanging structured information in a decentralized, distributed environment"
 - XML-based messaging framework
 - SOAP messages can be exchanged over any protocol (e.g., HTTP, SMTP)
 - SOAP messages originate at an initial SOAP *sender* and are sent to an ultimate SOAP *receiver* via zero or more SOAP *intermediaries*
 - Multiple *message exchange patterns* supported: one-way, request/response, peer-to-peer
- **WSDL** (Web Services Description Language)
 - "XML grammar for describing network services as collections of communication endpoints capable of exchanging messages"

- **Simple Object Access Protocol (SOAP)**
Protocollo per lo scambio di messaggi tra *sender* e *receiver*, per gestire le interazioni tra consumers e producers.
- **Web Services Description Language (WSDL)**
Regole per gestire le descrizioni dei servizi, si regola le interazioni tra consumers e brokers.

Lo schema è il solito già visto prima, dove la *Service Directory* equivale al *Broker* e il *Web Service* è il *Service Provider*.



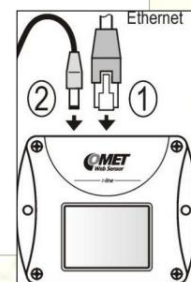
La proposta è un miglioramento rispetto a MQTT: è possibile avere interazioni *machine-to-machine*!

- **Publish.**
È stato appena creato un Web Service, viene contattato il Broker (Service Directory) per segnalare la sua esistenza.
- **Lookup and Messages.**
Il Service Consumer contatta il Broker (Service Directory) per avere informazioni su Web Service a cui desidera accedere. Il Broker risponde fornendo le informazioni necessarie, a quel punto il Service Consumer dialoga direttamente col Web Service.

I vari passaggi descritti sono possibili grazie all'implementazione di SOAP e WSDL in tutti i dispositivi coinvolti, sopra il protocollo HTTP.

• SOAP message example

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <InsertTx5xxSample xmlns="http://cometsystem.cz/schemas/soapTx5xx_v2.xsd">
      <passKey>13960932</passKey>
      <device>4145</device>
      <temp>1.4</temp>
      <relHum>91.9</relHum>
      <compQuant>0.3</compQuant>
      <pressure>-9999</pressure>
      <alarms>hi,no,no,no</alarms>
      <compType>Dew point</compType>
      <tempUC</tempUC>
      <pressureUC</pressureUC>
      <timer>60</timer>
    </InsertTx5xxSample>
  </soap:Body>
</soap:Envelope>
```



Tutto molto bello, ma c'è una questione: se il producer e il consumer sono dispositivi IoT allora non abbiamo a disposizione storage e risorse computazionali in abbondanza. Quanto descritto funziona, ma è *too cumbersome* per il *Web of Things* (e certe volte lo è pure per il Cloud!).

2.5 Proposta: RESTful Web Services

2.5.1 Introduzione

Adottiamo un approccio diverso, tenuto conto delle osservazioni precedenti: introduciamo REST (acronimo di *REpresentation State Transfer*).

- Vengono rimosse tutte le funzionalità/protocolli extra non rilevanti ai fini dell'IoT, rimane unicamente l'indispensabile. Approccio sempre basato sulle tecnologie del Web (continuiamo a utilizzare URI, HTTP e XML) e sulle architetture *service-oriented*.
- Approccio *web-centric*!
 - o Il server è il *provider* e il client il *consumer*. Il client conosce l'URI che identifica particolari dati/servizi: lancia una *HTTP Request* al server, che risponderà con una *HTTP Response*.
 - o Il server espone un'interfaccia caratterizzata da risorse.
 - o Ogni risorsa vede associata dati e/o funzionalità. Per ogni risorsa andiamo a definire quattro *methods*: GET, POST, PUT e DELETE (azioni CRUD, in prestito dai database).

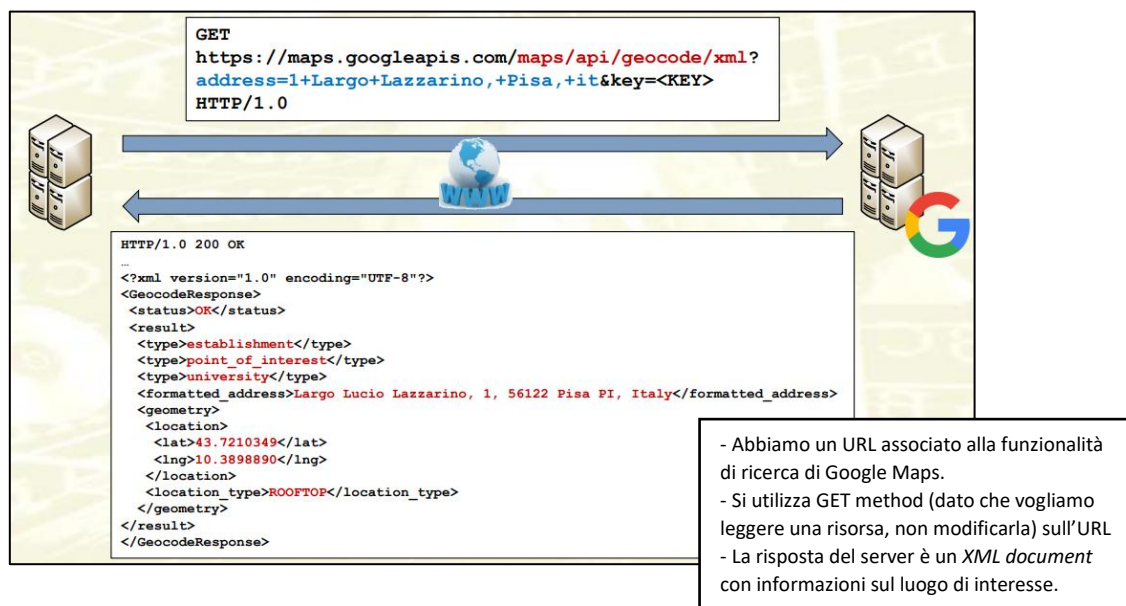
- Interaction is realized by **four standard actions (CRUD)**
 - **Create** a new resource from the request data
 - **Read** a resource state
 - **Update** a resource state from the request data
 - **Delete** a resource
- A representation of the resource state is retrieved as a result of the interaction

- o Ogni risorsa è associata a un URI: non si punta a una pagina web, ma a una risorsa!

- HTTP methods associated to actions
 - **POST**: Create a new resource from the request data
 - **GET**: Read a resource
 - **PUT**: Update a resource from the request data
 - **DELETE**: Delete a resource
- URIs provide a method to identify (cyberphysical) **resources**
 - `http://example.com/sensors`
 - `http://example.com/sensors/temp`

2.5.2 Esempio: API di Google Maps

L'approccio introdotto è largamente utilizzato per la creazione di API. Prendiamo ad esempio Google Maps



2.5.3 Esempi nel Web of Things

Come applichiamo le cose dette nel *Web of Things*? Consideriamo i seguenti esempi!

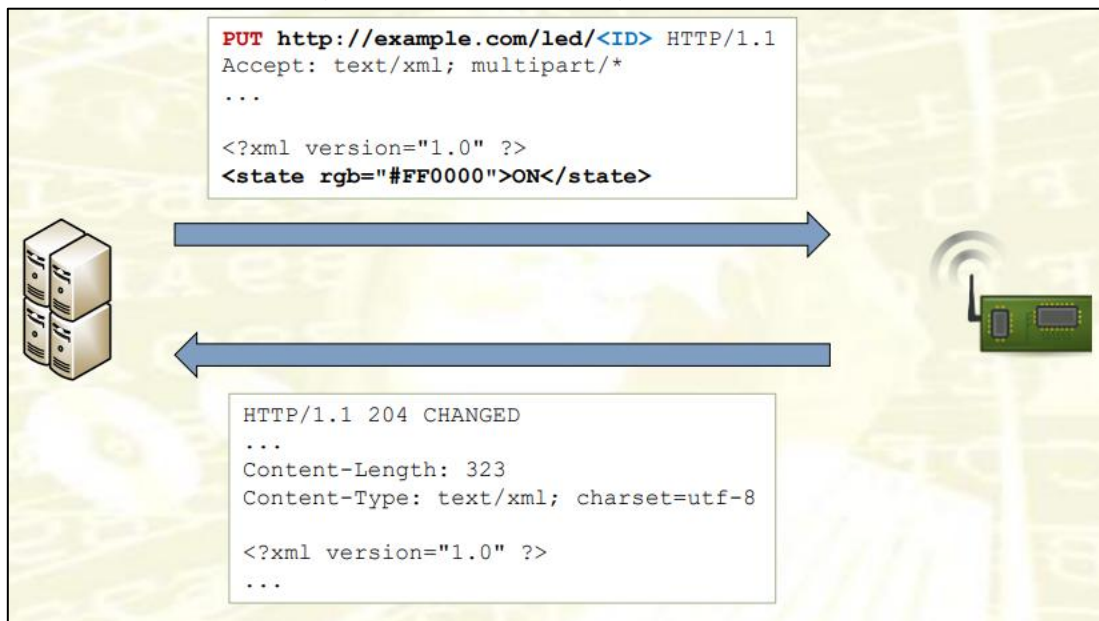
- **Reading sensor data.**

Supponiamo di avere un generico *Smart Device* identificato da un <ID>, vogliamo visualizzare i dati. Implementa il ruolo dell'HTTP Server: è dotato di risorsa rappresentata da URL `http://example.com/sensors/<ID>`. La risposta è un *XML Document* con tali dati (Esempio: temperature se lo Smart Device è dotato di un sensore per la temperatura).



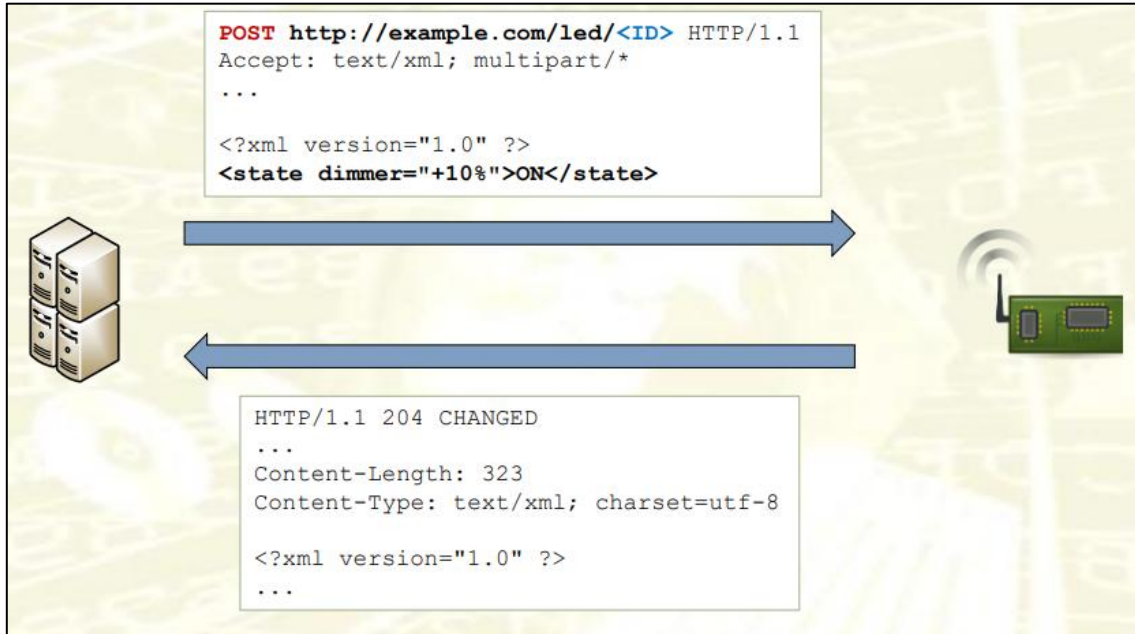
- **Accendere un led sul sensore.**

Supponiamo che lo Smart Device sia dotato di un led. Lo Smart Device, che anche qua si comporta da Web Server, implementa una risorsa `led`. Poniamo PUT perché aggiorniamo una risorsa esistente. Nel payload poniamo ulteriori informazioni in formato XML (la volontà di accendere il led e il colore che dovrà avere il led). La risposta del Web Server è semplice: si limita a segnalare che lo stato della risorsa è cambiato (HTTP/1.1 204 CHANGED).



- **Aumentare la luminosità del led.**

Un altro esempio potrebbe essere aumentare la luminosità dell'output. Inviando una POST Request, il payload indica di quanto si desidera aumentare (in percentuale) la luminosità del led.



2.5.4 Siamo soddisfatti? Non ancora

La soluzione proposta è sicuramente un miglioramento rispetto a quelle precedenti: sono possibili interazioni machine-to-machine e il focus è posto interamente sull'IoT device. Overhead minimizzato, rimozione di funzionalità/protocolli superflui.

- **RESTful web services** have emerged as a predominant approach for the Web of Things
- **Advantages**
 - **less** overhead, **less** parsing complexity, **tighter** integration with HTTP, **stateless**
- However, IoT resources are hosted by **constrained devices** connected to **constrained networks**
 - URI is an extensible id system 😊
 - HTTP (over TCP) is stable, but **large** overhead and complexity 😞 → **new protocol CoAP**
 - XML is flexible, but large overhead and complexity 😞 → **alternative encoding and data representation**

Possiamo considerarci soddisfatti? Non ancora.

- Su URI non abbiamo nulla da dire, è perfetto così.
- HTTP è valido per computer e certi IoT Device, ma su altri IoT device è ancora *too cumbersome*. Il problema principale è che HTTP si basa su TCP, che è veramente pesante: ricordiamo che dobbiamo eseguire cose su dispositivi con risorse estremamente limitate.
- XML è flessibile, ma estremamente ridondante. Siamo in una LLN, dove i pacchetti trasmessi sono di pochi byte: vorremo cercare di ridurre la dimensione dei nostri messaggi!

Prossimi passaggi: HTML → CoAP | XML → JSON

3 CONSTRAINED APPLICATION PROTOCOL (CoAP)

3.1 Introduzione

3.1.1 CoAP: come ci siamo arrivati e cosa consiste

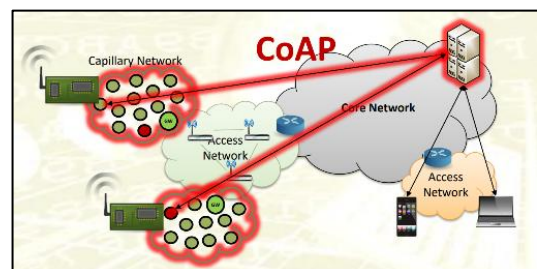
Riepiloghiamo i passaggi che ci sono stati nelle pagine precedenti.

- **Obiettivo ideale.** Implementare l'architettura orizzontale (*converged infrastructure*). MQTT, nonostante l'introduzione del *fog computing*, non è un protocollo adatto per gestire la comunicazione tra dispositivi IoT: impossibile stabilire comunicazione machine-to-machine, tempi di latenza alti, incertezza della collocazione del MQTT Broker.
- **Prima proposta.** Implementazione di una *service-oriented architecture* con ispirazione ai Web Services (e quindi il World Wide Web) visti per il Cloud Computing: *Web of Things*!
 - o Si adottano URI, HTTP e XML (questo ultimo al posto di HTML, che non è adatto).
 - o Si utilizzano gli stessi protocolli utilizzati nel Cloud Computing: SOAP e WSDL.
 - o Problema: comunicazione machine-to-machine presente, ma subordinata a un passaggio da un soggetto terzo noto come *service broker*.La proposta è *too cumbersome* per il Web of Things.
- **Seconda proposta.** Si mantiene il Web of Things, ma introduciamo i *RESTful Web Services*, dove tutti i protocolli/funzionalità superflui vengono rimossi. Approccio web-centrico dove il Web Server è il provider (nel nostro caso il dispositivo IoT) e il client il consumatore: espone risorse per mezzo di URI, a cui possono accedere i dispositivi interessati.
 - o Implementazione di API per mezzo di questo approccio.
 - o Cose belle: possibile stabilire interazioni machine-to-machine, non necessario definire ogni singola possibile interazione come con SOAP e WSDL. In generale maggiore flessibilità
 - o Problema: HTTP è troppo pesante per essere eseguito su dispositivi IoT (caratterizzati da risorse limitate), soprattutto perché si basa su TCP. Anche XML è molto pesante, vorremo ridurre la dimensione dei pacchetti trasmessi (siamo in una LLN).

Anche questa proposta è *too cumbersome* per il Web of Things.

- **Terza (e ultima) proposta.** Sostituiamo HTTP con CoAP e XML con JSON. CoAP consiste in una riscrittura *from scratch* del protocollo HTTP (lavoro del gruppo CoRE di IETF).
 - o E' pensato proprio per *constrained devices* (RAM di piccole dimensioni, batteria con scarsa autonomia) posti in una rete IoT (una rete *unreliable* e con *low bitrate*, una LLN).
 - o Permette comunicazione machine-to-machine.
 - o Non è semplicemente una versione semplificata di HTTP ("*not a blindly compressed version of HTTP*"). Diversi cambiamenti, ma l'idea alla base del *Web of Things* rimane.

- **Constrained Application Protocol**
- Generic **web protocol** for the special requirements of **constrained environment** (LLNs), especially considering **Machine-to-Machine (M2M)** applications
 - Designed by the IETF CoRE Working Group – **RFC 7252** (June 2014)
- Goal: realize an application protocol for web services (common with HTTP) but optimized for M2M applications
 - Not a blindly compressed version of HTTP



3.1.2 Novità principali

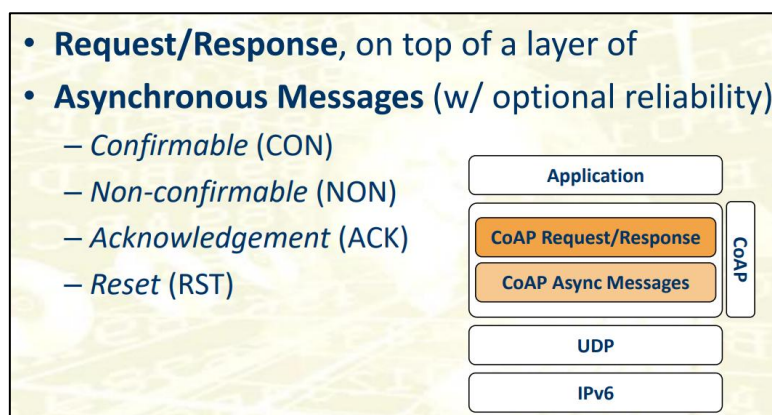
Si segnalano le seguenti novità:

- • Binding to **UDP** transport protocol
- • **Request/response** communication model, with support for **asynchronous** message exchanges
- • **Low** header **overhead** and **parsing** complexity
- Simple **proxy** and **caching** capabilities
- Stateless **HTTP** mapping
- Support for **multicast** communication
- • Support for **discovery** of resources

- Implementato sopra il protocollo di trasporto UDP (e non TCP come in HTTP). L'implementazione di UDP richiede molte meno risorse (questo perché UDP non implementa *reliability*, *ordered delivery* e *congestion control*).
- Viene mantenuto il modello di comunicazione di HTTP (*request/response*), ma si introduce anche il supporto per messaggi asincroni (in HTTP è possibili solo uno scambio sincrono, ergo il client deve attendere risposta del server prima di svolgere qualunque altra azione).
- Il protocollo è pensato per essere meno complesso: la dimensione dell'header del pacchetto è ridimensionata (*low header overhead*), la complessità del *parsing* è ridotta (questo perché CoAP è un protocollo *binary-based* e non *text-based* come HTTP).
 - Minore complessità perché non deve essere più fatto il *parsing* del testo. Protocollo pensato per dialogo tra macchine e non tra persone.
- Si introduce il supporto per l'individuazione delle risorse (*support for discovery of resources*), non presente in HTTP.

3.1.3 Modello request-response e *optional reliability* implementata in *asynchronous messages*

Abbiamo già detto che viene mantenuto il modello di comunicazione *request/response* e adottato UDP al posto di TCP. Sulla *reliability*: UDP non la implementa, ma è una cosa che potrebbe essere necessaria per specifiche applicazioni! CoAP implementa **lightweight reliability mechanisms** opzionali (la logica è spostata dal *transport layer* all'*application layer*). Sono definite quattro tipologie di messaggi:



- **Confirmable (CON).**
Messaggio che deve essere *reliable*. Il nodo destinatario riceve il messaggio *confirmable*, verifica se il messaggio è *reliable* o meno. In caso affermativo trasmette un *acknowledgement* al nodo mittente.
- **Acknowledgement (ACK).**
Messaggio trasmesso dal nodo destinatario per informare il nodo mittente che il messaggio è stato ricevuto correttamente

- **Non-confirmable (NON).**
Messaggio su cui non sono richiesti controlli per la reliability.
- **Reset (RST).**
Messaggio per resettare la comunicazione tra due nodi (Esempio: a seguito di errori).

Request e response sono caratterizzate dai seguenti elementi:

- Same model as HTTP (client/server)
- Request
 - Method
 - Request URI
 - Payload and optional meta-data (CoAP options)
- Response
 - Response Code
 - Payload and optional meta-data

- **Request.**
 - Method (i soliti di CRUD: GET, POST, DELETE e PUT, si veda 3.1.4 - Request methods)
 - Request URI, che indica il server su cui è collocata la risorsa di nostri interesse e la risorsa stessa (Esempio: `coap://sensor.com/temperatura`)
 - Può contenere ulteriori informazioni (legate alla richiesta) nel payload.
- **Response.**
 - Response code, gli stessi definiti in HTTP (si veda 3.1.5 - Classificazione dei response codes)
 - Presente il payload (solitamente quasi sempre, si restituisce l'esito della computazione).

3.1.4 Request methods

Conosciamo già i request method definiti in PHP: GET, POST, DELETE e PUT. Come decidiamo il method più appropriato da associare a una certa richiesta?

La richiesta può essere:

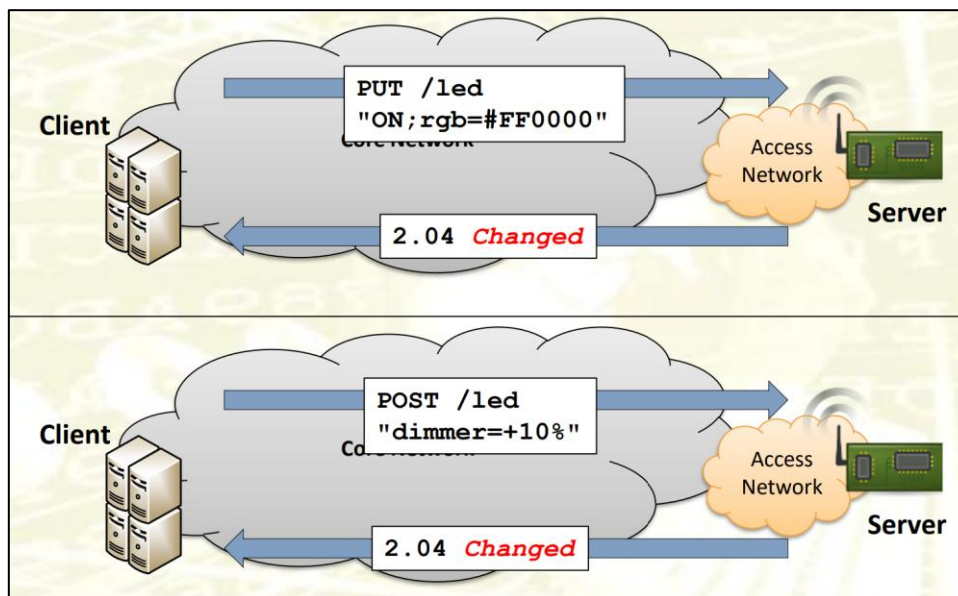
- *safe*, se non altera lo stato del sistema;
- *idempotent*, se l'output/esito della richiesta non cambia in base al numero di invocazioni.

- **GET:** retrieves a representation of the state of the resource (safe and idempotent)
- **POST:** requests that the representation enclosed be processed (neither safe nor idempotent)
 - It usually results in a new resource being created or the target resource being updated
- **DELETE:** requests the resource to be deleted (not safe but idempotent)
- **PUT:** requests that the resource be updated with the enclosed representation (not safe but idempotent)

- **GET.**
Metodo utilizzato per recuperare lo stato della risorsa (Esempio: recuperare la temperatura captata dal sensore).

- È *safe* e *idempotent*.
- **POST.**
Metodo utilizzato per l'invio di una nuova configurazione al server (Esempio: configurazione con cui si determina il livello di precisione in un sensore di temperature, si mette la nuova configurazione nel payload), in generale invio di dati che il server dovrà processare. Si creano nuove risorse o si aggiornano quelle esistenti.
 - Non è *safe* e non è *idempotent*.
- **DELETE.**
Si richiede la rimozione di una risorsa (rimozione di una certa configurazione, disattivazione di una certa funzionalità).
 - Non è *safe*, ma è *idempotent* (se una risorsa è eliminata rimane eliminata, non succede nulla di più)
- **PUT.**
Method utilizzato per cambiare lo stato di una risorsa esistente (aggiunta di una nuova configurazione).
 - Non è *safe*, ma *idempotent*.

Il dubbio che potrebbe emergere è come distinguere POST da PUT. Basta ricordarsi se il method è *idempotent*.



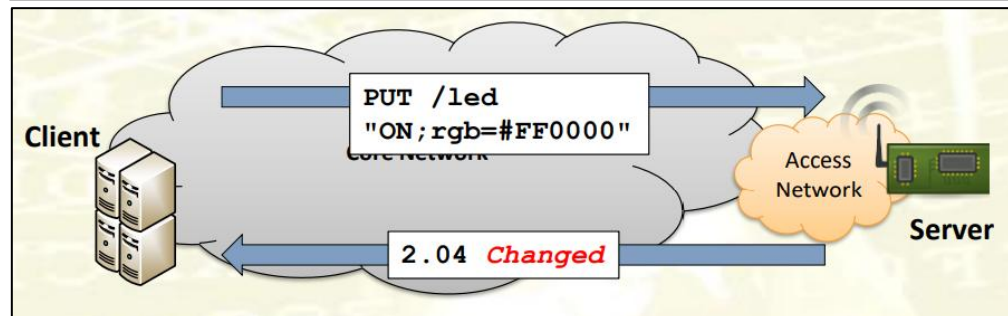
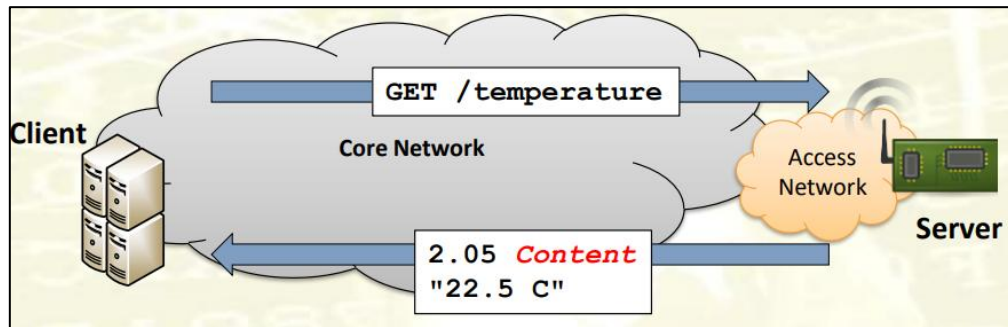
- **PUT in quanto idempotente.**
La prima richiesta (accensione del led – *an actuator* - e indicazione del colore) è PUT: se invio tale richiesta molteplici volte lo stato del sistema non cambia (il led è già acceso e rimane acceso con le stesse caratteristiche)
- **POST perché non idempotente.**
La seconda richiesta (aumento della luminosità del led) è POST: se invio tale richiesta molteplici volte lo stato del sistema cambia molteplici volte (ogni volta la luminosità viene incrementata del 10%).

3.1.5 Classificazione dei *response codes*

I *response codes* sono raggruppati in tre classi:

- **Success (2xx).**

La richiesta è stata ricevuta ed eseguita correttamente. Esempio: response code 204 o 205.



- ✓ **204 No Content**

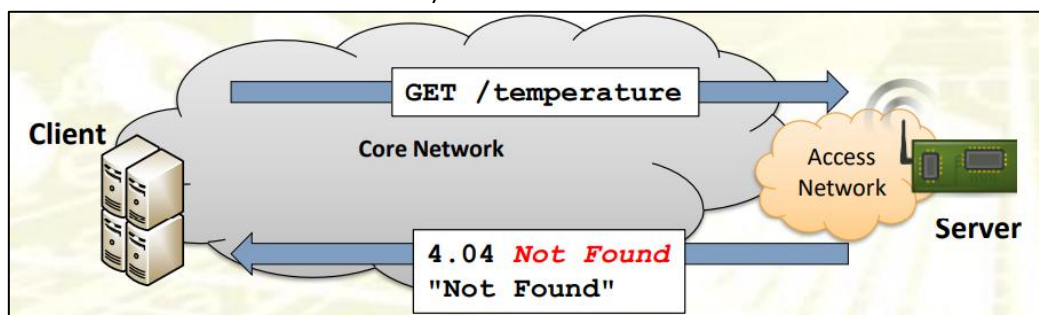
- **Significato:** Richiesta completata con successo, **nessun contenuto da restituire**.
- **Uso tipico:** DELETE, PUT o POST quando **non serve** una risposta.

- ✓ **205 Reset Content**

- **Significato:** Richiesta completata, ma il client deve **resettare l'interfaccia utente** (es. un form).
- **Uso tipico:** Dopo l'invio di un **modulo/form**, il server indica di **svuotare i campi**.

- **Client error (4xx).**

Il client (chi ha mandato la richiesta) ha commesso un errore. Esempio: errore 404, il client ha richiesto accesso a una risorsa inesistente).



- **Server error (5xx).**

C'è un errore lato server.

3.2 Request/Response transport

3.2.1 Introduzione di *lightweight reliability mechanisms*

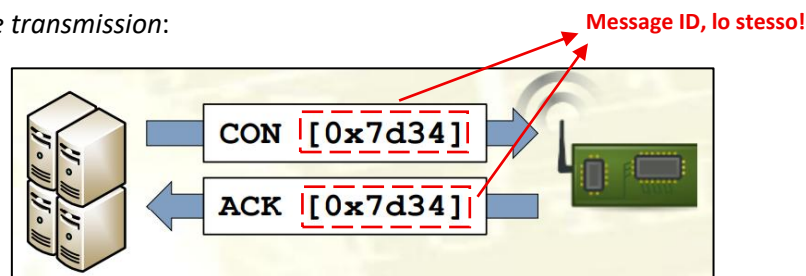
Abbiamo detto che CoAP si basa sul protocollo di trasporto UDP, che non garantisce la *reliability* dei pacchetti trasmessi.

- CoAP is bound to **non-reliable transport** (i.e., UDP)
- A **lightweight** reliability mechanism is needed
 - Stop&Wait **retransmission** with exponential back-off
 - **Duplicate detection**
- Requests and responses are carried by CoAP **asynchronous messages**, which implement optional reliability

- La reliability potrebbe essere necessaria: per questo CoAP implementa a livello applicazione *lightweight reliability mechanism*.
 - o Si adotta **Stop&Wait retransmission with exponential back-off** (si veda 3.2.2 - *Reliable transmission*)
 - o Si introduce **duplicate detection**: in caso di trasmissione di pacchetti duplicati (cosa per nulla impossibile vista la *retransmission*) questi vengono riconosciuti e scartati.
- Si introduce la possibilità di scambiare messaggi in maniera asincrona.
- **Due layer** (nel registro parla di "*request/response sub-layers*"): prima abbiamo le richieste e le risposte; tali richieste e risposte sono incapsulate in un messaggio di tipo CON e NON.

3.2.2 Reliable transmission

Introduciamo la *reliable transmission*:



- Il client invia la richiesta ponendola all'interno di un CON message.
- Il server riceve la richiesta. Tale richiesta è posta in un CON message, il server può:
 - o fare l'acknowledgement inviando l'ACK message;
 - o inviare un RST message per segnalare che qualcosa è andato storto (l'alternativa è ignorare il messaggio, ma fare aspettare il client più del dovuto non è mai una buona scelta).
- CON message e relativo ACK message sono legati tra loro dal **message ID**, un identificativo di 16 bit.

- **Stop&Wait retransmission with exponential back-off.**

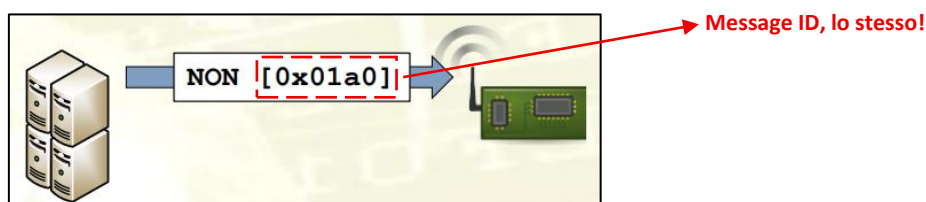
Abbiamo già anticipato l'introduzione di questo meccanismo. Cosa succede se l'ACK non arriva a destinazione?

- o Il client trasmette un messaggio al server e attende un ACK da parte di questo.
- o Se l'ACK non arriva al client entro un certo tempo un *timeout* viene triggerato e il pacchetto nuovamente ritrasmesso.
 - Quanto tempo deve aspettare il client? Viene scelto randomicamente all'interno di un intervallo che cresce esponenzialmente ad ogni *retransmission*.
 - L'intervallo iniziale è [0; 2 s], il ritardo è scelto al suo interno (Esempio: 0.5 s)

- Avviene *retransmission*: l'intervallo diventa $[0; 4 \text{ s}]$, il ritardo è scelto al suo interno (Esempio: 3 s).
- Proseguono le *retransmissions*: $[0; 4 \text{ s}] \rightarrow [0; 8 \text{ s}] \rightarrow [0; 16 \text{ s}]$.
- Si continua così fino a quando non si raggiungerà il numero massimo di *retransmissions* (che di default è 4).
- Perché serve questo meccanismo?
 - La ragione principale per la non ricezione di ACK in una LLN è la congestione (dato che la trasmissione dei pacchetti avviene con bitrate molto basso). Serve un meccanismo che riduca significativamente il numero di pacchetti immessi nella rete in caso di packet loss.
 - Il client non può fare altro finché non riceve l'ACK, ergo la trasmissione di ulteriori pacchetti da parte del client è limitata (Attenzione all'*asynchronous mode*, si legga 3.2.4 - *Modalità di trasmissione della risposta* – per i dubbi che potrebbero emergere leggendo questa frase).

3.2.3 Non-reliable transmission

La comunicazione *non-reliable* è molto più semplice da implementare:



- Il client invia la richiesta ponendola in un NON message.
- Il server non deve inviare nessun ACK!
- Il message ID rimane per gestire l'individuazione dei duplicati!

Il mittente non può sapere se il messaggio è stato ricevuto o meno.

3.2.4 Modalità di trasmissione della risposta

Esistono diverse modalità di trasmissione di una risposta. In ogni caso il server deve considerare che:

- se la richiesta è *reliable* allora anche la risposta dovrà essere *reliable*;
- se la richiesta è *non-reliable* allora anche la risposta dovrà essere *non-reliable*.

Ricordiamo che è il client a decidere se la richiesta è *reliable* o no. Le risposte sono associate alle richieste per mezzo di un ulteriore ID: il **token** (diverso dal message ID citato precedentemente). Ogni pacchetto presenterà al suo interno un *token* e un *message ID*!

- **Risposta in *reliable transmission*.**

Due strade possibili:

- ***Separate mode.***

L'ACK message si limita a comunicare l'avvenuta ricezione del messaggio di richiesta (ergo il relativo payload è vuoto). La risposta alla richiesta verrà trasmessa con un successivo CON message, a cui il client dovrà rispondere a sua volta con un altro ACK message.

- ***Piggy-backed mode.***

L'ACK message ha duplice scopo:

- comunicare l'avvenuta ricezione del messaggio di richiesta;
- rispondere alla richiesta (tale risposta è posta nel payload dell'ACK message).

Si fa tutto in un unico pacchetto: sicuramente il più efficiente tra i due metodi!

La *separate mode* si adotta in caso di comunicazione asincrona, la *piggy-backed mode* in caso di comunicazione sincrona!

- Comunicazione asincrona significa che il client può fare altre attività tra l'invio della richiesta e la ricezione della relativa risposta.

- Se la comunicazione è asincrona e *reliable* dobbiamo essere certi che richiesta e risposta arrivino a destinazione.
- Il client trasmette la richiesta al server, il server risponde al client con un ACK message (payload vuoto). Se adottato la piggy-backed mode la risposta è inclusa nell'ACK, ma questa arriverà più tardi e nel frattempo il client non potrà fare altre operazioni.
- Il server trasmette (dopo un po' di tempo) la risposta al client, il client risponde al server con un ACK message (payload vuoto).

Il server decide quale modalità adottare considerando il tempo necessario per processare la richiesta. Consideriamo un sensore: se ha bisogno di molto tempo per produrre la risposta allora adotterà la *separate mode*. Adottare la *piggy-backed mode* in questa circostanza significherebbe: tenere il client in attesa per troppo tempo; possibile superamento di timeout per non ricezione di ACK e conseguenti retransmissions.

- Risposta in *non-reliable transmission*.

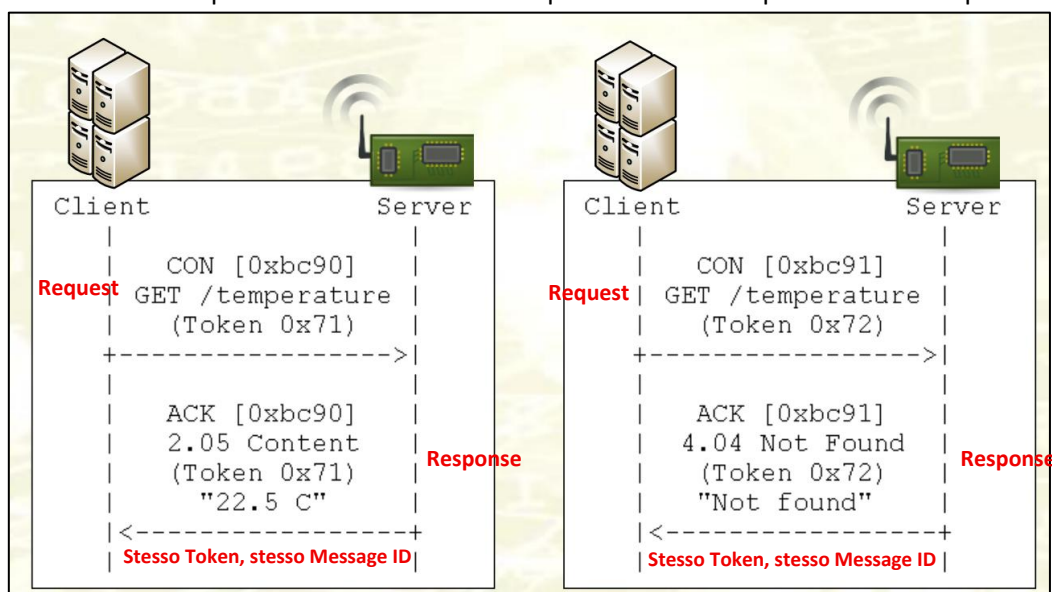
La risposta è inviata con un nuovo NON message. Non sono previsti ACK!

3.2.5 Esempi di Request/Response

Consideriamo alcuni esempi in forma di grafico:

- [Reliable transmission] Comunicazione in *piggy-backed mode*.

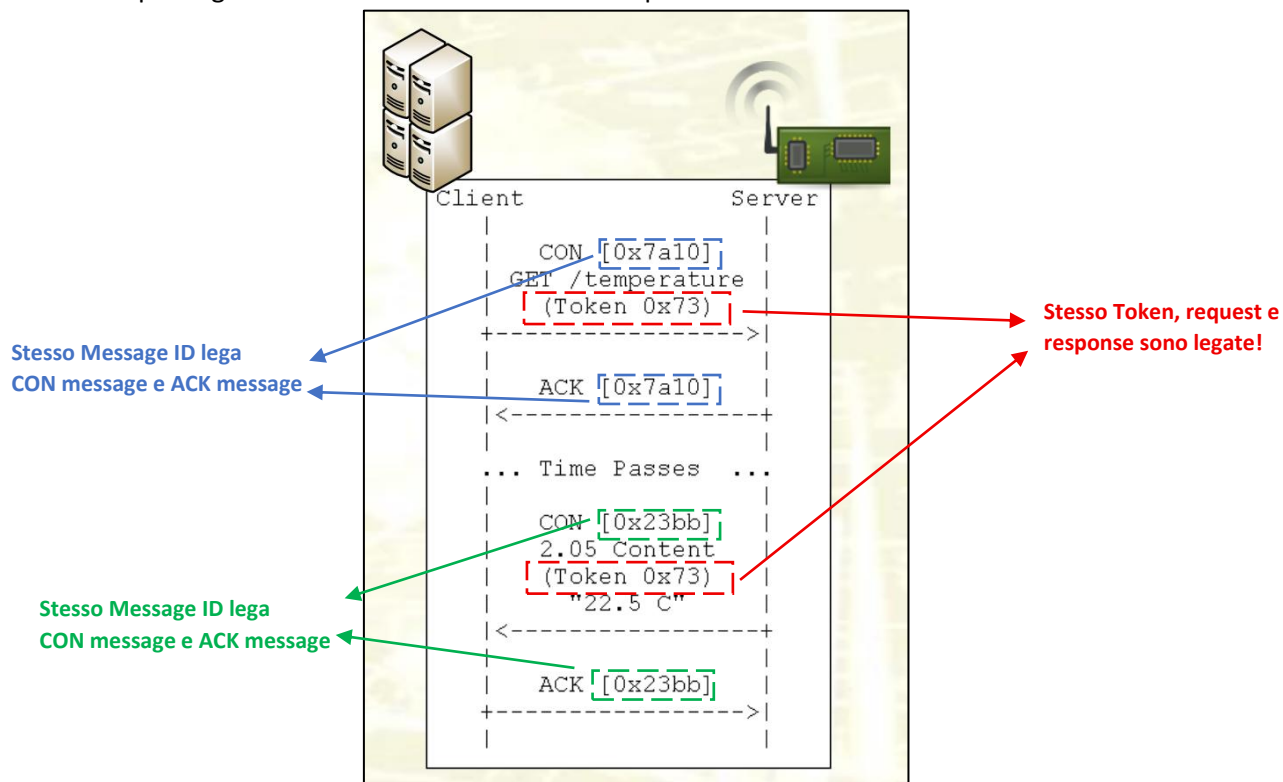
Il client trasmette una richiesta (GET /temperature) al server (il sensore) in un CON message. Il sensore risponde con un ACK message che contiene all'interno del payload la temperatura richiesta. Può anche rispondere con un errore 404 qualora non sia disponibile una temperatura.



Osserviamo (riflessioni valide sia per l'esempio a sinistra che per quello a destra):

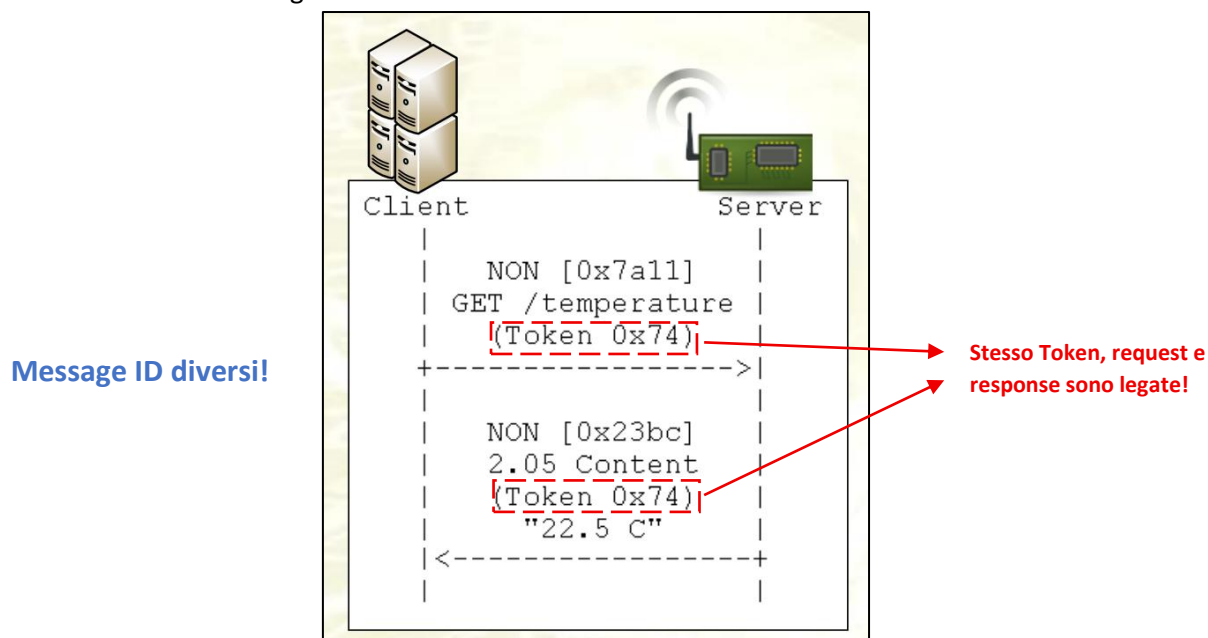
- Il message ID dei due messaggi è lo stesso (nulla di strano, si lega CON message ad ACK message).
 - Il token dei due messaggi è lo stesso (nulla di strano, si lega request a response).
 - **Non è presente una parte nell'immagine:** l'ultimo messaggio è un ACK message con payload, è necessario che si accerti l'effettiva trasmissione del contenuto del payload! Il client dovrà trasmettere un ulteriore ACK message (con payload vuoto) per confermare la ricezione delle informazioni contenute nel payload.
- #### - [Reliable transmission] Comunicazione in *separate mode*.
- Il client trasmette una richiesta (GET /temperature) al server (il sensore) in un CON message. Il sensore risponde con un semplice ACK message (payload vuoto) per segnalare l'avvenuta ricezione della richiesta al client.
 - Il sensore processa la richiesta (... Time Passes ...)

- Il sensore trasmette la risposta al client con un nuovo CON message. La risposta è posta nel payload. Il client, ricevuta la risposta, trasmette un semplice ACK message (payload vuoto) per segnalare l'avvenuta ricezione della risposta al server.



- **Non-reliable transmission.**

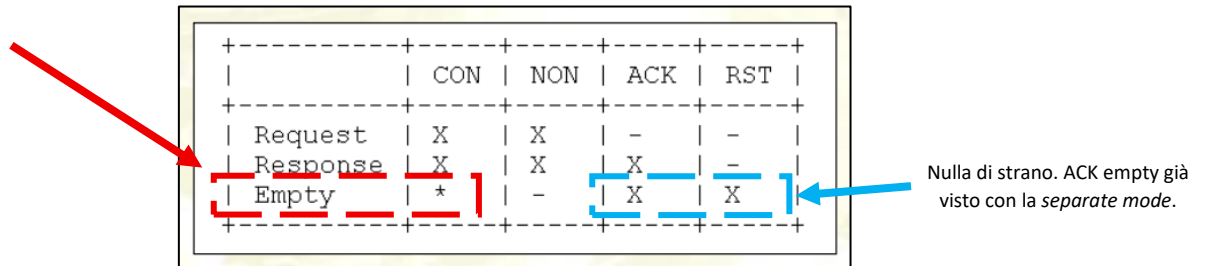
Tutto molto più semplice: il client trasmette una richiesta (GET /temperature) al server (il sensore) in un NON message. Il server non deve trasmettere ACK: comunica la risposta per mezzo di un nuovo NON Message.



Si osservi che: il *message ID* è diverso (ci torna, è pensato per legare CON e ACK messages – qua abbiamo NON messages); il *token* è lo stesso (anche questo ci torna, leghiamo *request* e *response*).

3.2.6 Empty message: definizione e CoAP ping

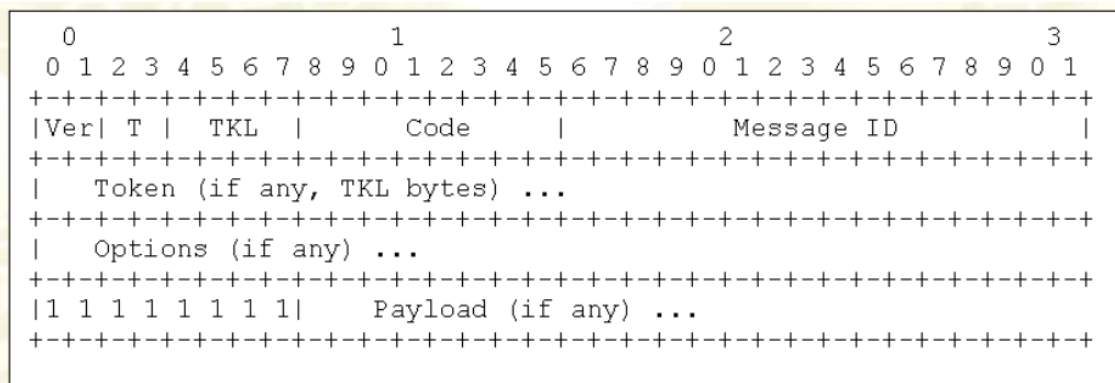
Lo standard prevede la possibilità di trasmettere *empty messages*, cioè messaggi senza method/response code e con payload vuoto.



- Abbiamo già visto negli esempi precedenti che è possibile creare empty message di tipo ACK e RST.
- Possibile trasmissione anche di *empty message* di tipo CON! Il messaggio è un ping (CoAP ping) che permette di verificare se il destinatario è ancora online. Il destinatario risponde con *RST message*.

3.2.7 Message format

Abbiamo *machine-to-machine interaction*: si pone *binary format*.



- Il pacchetto ha una dimensione fissa: questo significa che è più semplice crearli (in HTTP bisogna tenere conto della dimensione variabile del pacchetto, in particolare del payload).
- La struttura è fissa, quindi possiamo gestire ogni singolo campo utilizzando una bitmap.
- Consideriamo alcuni campi:
 - **Version (2 bit).**
Versione del protocollo. Attualmente siamo alla versione 1.
 - **T (2 bit).**
Tipologia di messaggio. Può essere CON Message (0 <- 00), NON Message (1 <- 01), ACK Message (2 <- 10), RST Message (3 <- 11).
 - **TKL (4 bit).**
TKL è acronimo di *token length*: indica la lunghezza, appunto, del campo token. Il campo deve essere lungo almeno 8 byte.
 - **Code (8 bit).**
Sequenza di 8 bit divisa in 3 bit *Class* e 5 bit *Detail*. Lo scopo varia in base al messaggio (se è *request* o *response*).

- **Code: split into 3-bit class and 5-bit detail (c.dd)**
 - Class: request (0), success response (2), client error response (4), server error response (5)
 - Detail: specifies the method (e.g., 0.01 is a GET) or the response code (e.g., 2.04 is 'Created')
 - Special code 0.00 indicates an Empty message

- **Class.**
 - **Request.** Il campo è uguale a 0.

- **Response.** Il campo può assumere uno dei seguenti valori: *success response* (2), *client error response* (4), *server error response* (5).
- **Detail.**
 - **Request.** Indica il *method* (Esempio: GET, con *Class* uguale a 0 e *Detail* uguale a 1).
 - **Response.** Indica il *response code* (Esempio: errore 404, con *Class* uguale a 4 e *Detail* uguale a 4).
- **0.00.** Se il *Code* è una sequenza di zeri allora segnaliamo che il pacchetto consiste nell'*empty message* introdotto in 3.2.6.
- **Message ID.**
Autoesplicativo, la sequenza di 16 bit introdotta nelle pagine precedenti.
- **Token.**
Autoesplicativo, la sequenza introdotta nelle pagine precedenti. Uno dei due campi la cui lunghezza è variabile (l'altro è *Options*).
- **Options.**
Uno dei due campi la cui lunghezza è variabile (l'altro è *Token*). Set di opzioni. Segnaliamo tra le opzioni:
 - *Uri-Host*, *Uri-Port*, *Uri-Path* e *Uri-Query Options* (l'URI viene diviso in più campi)
 - *Max-Age*, che permette al server di segnalare al client la durata della validità della risposta. Supponiamo che sia 60 s (default value): è un modo per dire che in caso di reiterazione della richiesta (entro 60 s) la risposta sarà la stessa.

• **coap-URI = "coap:" "://" host [":" port] path ["?" query]**

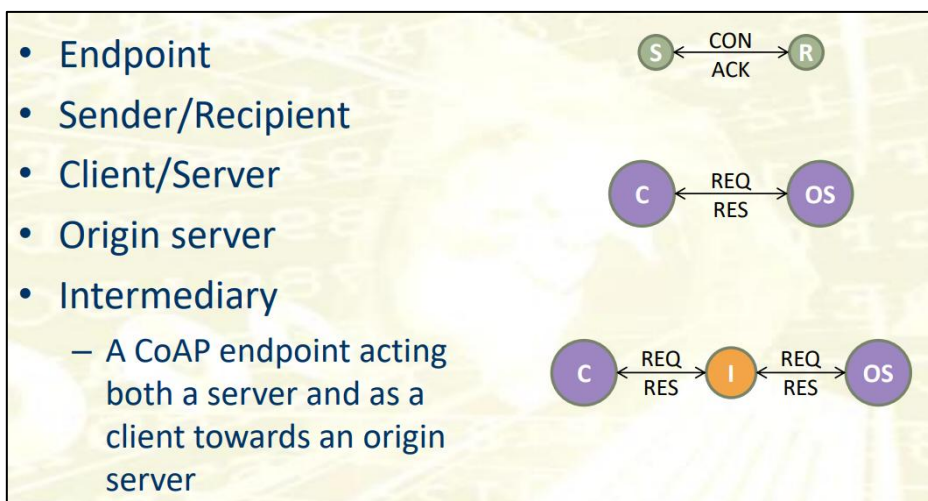
No.	C	U	N	R	Name	Format	Length	Default
1	x			x	If-Match	opaque	0-8	(none)
3	x	x	-		Uri-Host	string	1-255	(see below)
4				x	ETag	opaque	1-8	(none)
5	x				If-None-Match	empty	0	(none)
7	x	x	-		Uri-Port	uint	0-2	(see below)
8				x	Location-Path	string	0-255	(none)
11	x	x	-	x	Uri-Path	string	0-255	(none)
12		x	-		Content-Format	uint	0-2	(none)
14		x	-		Max-Age	uint	0-4	60
15	x	x	-	x	Uri-Query	string	0-255	(none)
17	x				Accept	uint	0-2	(none)
20				x	Location-Query	string	0-255	(none)
35	x	x	-		Proxy-Uri	string	1-1034	(none)
39	x	x	-		Proxy-Scheme	string	1-255	(none)
60			x		Size1	uint	0-4	(none)

Request URI is split into the Uri-Host, Uri-Port, Uri-Path and Uri-Query Options

3.3 Proxying

3.3.1 Introduzione

Il proxying era già presente in HTTP e viene introdotto anche in CoAP. L'idea di base è sempre la stessa!



Nuova terminologia	
<i>Endpoint</i>	Qualunque dispositivo che implementa il protocollo CoAP. Sinonimo sia per server che per client.
<i>Origin server</i>	Server che produce l'informazione.
<i>Intermediary</i>	Proxy che si pone come intermediario tra il <i>client</i> e l' <i>origin server</i> . - Si comporta come Server dal punto di vista del Client. - Si comporta come Client dal punto di vista dell'Origin Server. La presenza dell'intermediario permette l'introduzione di funzionalità che potenziano l'interazione tra client e server. Abbiamo i seguenti passaggi: - il Client invia una richiesta; - la richiesta viene ricevuta dall'intermediario, che ne fa il parsing; - l'intermediario inoltra la richiesta all'Origin Server; - l'Origin Server trasmette la risposta all'intermediario; - l'intermediario inoltra la risposta al Client.

Perché ci interessa questa cosa?

- **Performance issues.**

L'introduzione di un intermediario permette la riduzione del carico di lavoro per il server (che è un *constrained device*) introducendo funzionalità di *caching*. L'idea è la seguente:

- l'intermediario riceve le richieste dal client e verifica se tale risposta è presente nella cache;
- se la risposta è presente può inoltrarla immediatamente al client, senza rivolgersi al server;
- se la risposta non è presente allora viene inoltrata la richiesta al server;
- il server trasmette la risposta all'intermediario, che la memorizza nella sua cache e la inoltra al client.

L'opzione *Max-Age* è molto utile: permette di indicare al Proxy per quanto tempo deve mantenere l'informazione all'interno della memoria cache.

- **Security issues.**

Difficile implementare misure di sicurezza su un *constrained device*. L'idea è di implementare tali meccanismi (access control, firewall, ...) sull'intermediario (che è decisamente più potente).

- **Protocol translation.**

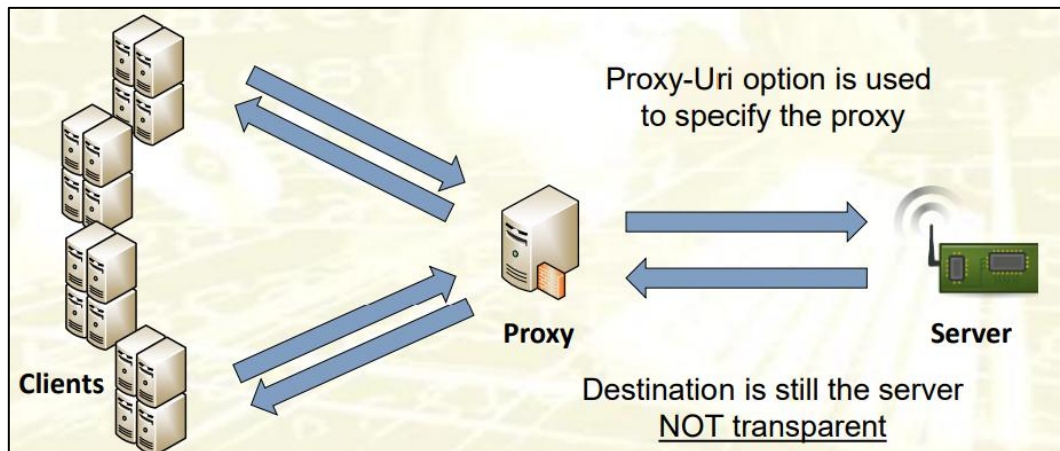
Possibile tradurre HTTP requests/responses in CoAP requests/responses e/o viceversa. Si veda sezione più avanti.

3.3.2 Tipologie di proxy: *forward proxy* e *reverse proxy*

Definiamo due tipologie di proxy:

- **Forward Proxy.**

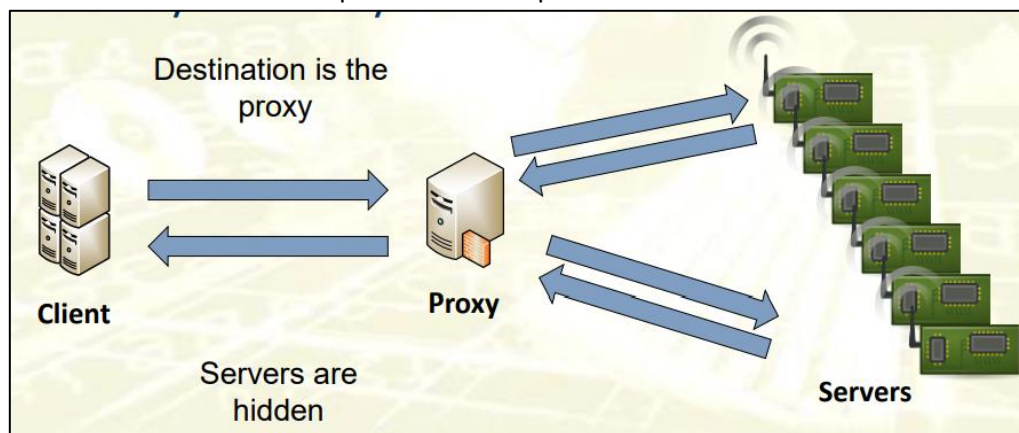
Il proxy è progettato in modo da non risultare trasparente al client, nel senso che il client è consapevole della presenza del proxy e lo contatta esplicitamente (al posto del server).



- Il client inoltra la richiesta al proxy (esplicitamente).
- Il proxy riceve la richiesta e la inoltra al server. Tra ricezione e invio possono essere svolte computazioni, ad esempio attuare i meccanismi di caching spiegati precedentemente.
- Il server trasmette la risposta al proxy.
- Il proxy inoltra la risposta al client.

- **Reverse Proxy.**

Il proxy è progettato in modo da risultare completamente trasparente al client, nel senso che questo non è assolutamente consapevole della sua presenza.

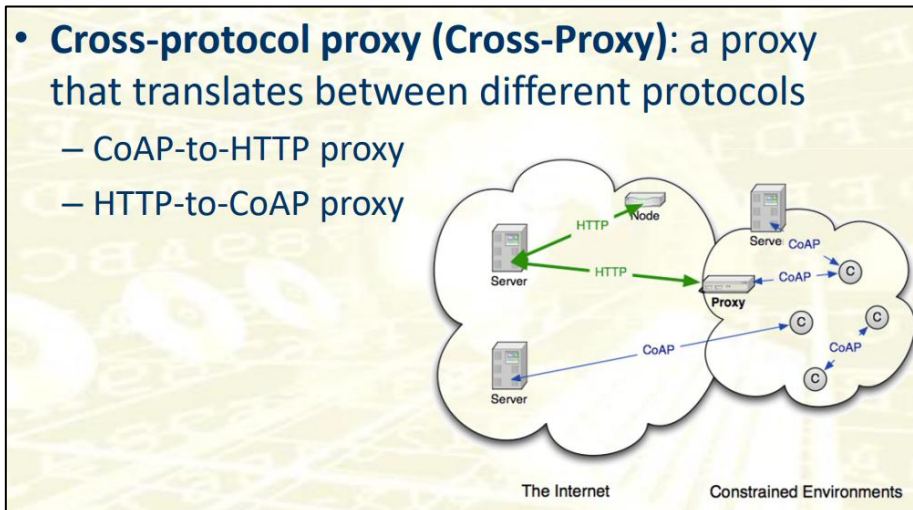


Di solito installiamo il proxy nell'edge della rete IoT. Questa tipologia di proxy è utilizzata principalmente per garantire la sicurezza della rete:

- Il proxy fa *discovery* dei dispositivi IoT presenti nella rete ed espone le risorse dei dispositivi IoT A soggetti come risorse proprie.
- Ogni volta che arrivano richieste da soggetti esterni queste vengono gestite dal proxy, che conosce le risorse presenti all'interno della rete.

3.3.3 Proxy translation

La natura delle architetture di HTTP e CoAP permettono di pensare a meccanismi di traduzione: conversione CoAP-to-HTTP e viceversa! Tale traduzione viene attuata dal proxy.

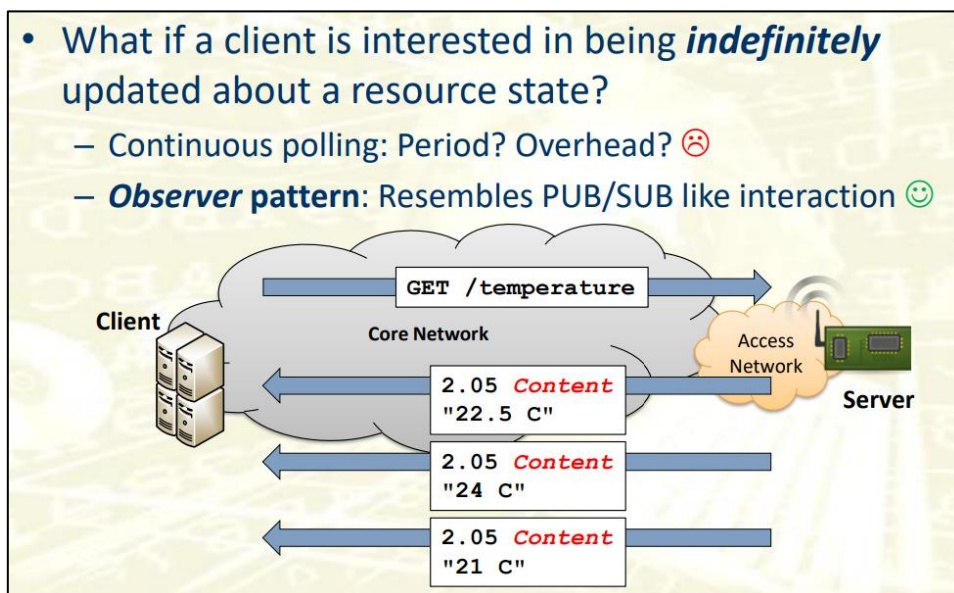


Esempio. Un sensore IoT vuole accedere a informazioni meteo (*weather forecast*) disponibili su un web service. Invia una richiesta CoAP: questa viene ricevuta dal proxy, che la traduce in una richiesta HTTP. Il server riceve la richiesta HTTP e genera una risposta HTTP, che viene inviata al proxy. Il proxy traduce la risposta HTTP in risposta CoAP e la inoltra al sensore IoT.

3.4 [Novità] Resource observing

3.4.1 Introduzione

Introduciamo la funzionalità di *resource observing*, una delle novità di CoAP rispetto a HTTP. È pensata per gestire una particolare tipica nelle reti IoT: interesse a monitorare per un tempo indeterminato i dati di particolari sensori/dispositivi.

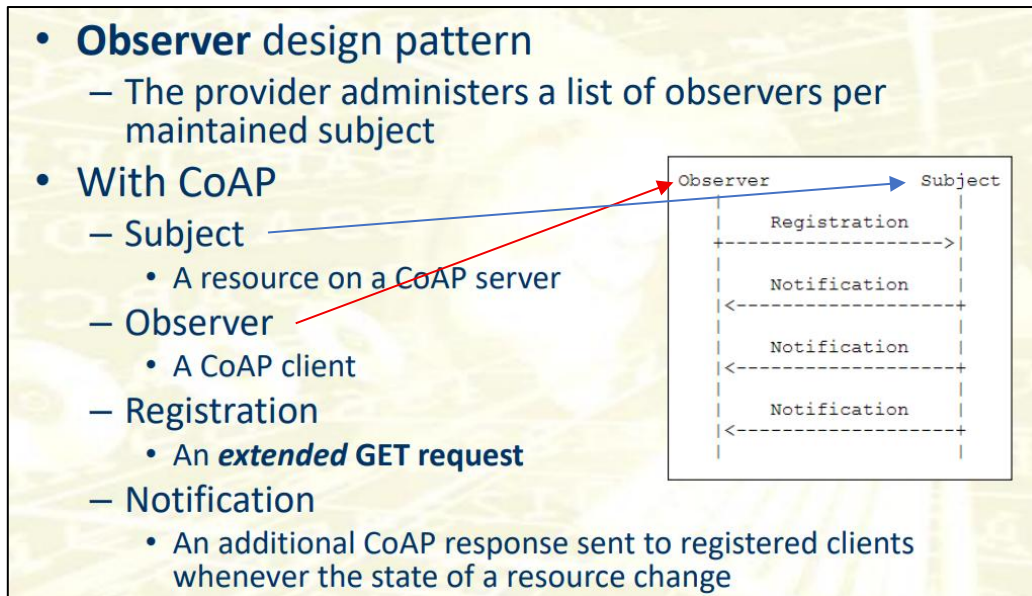


- Possiamo implementare questa cosa senza il *resource observing*?

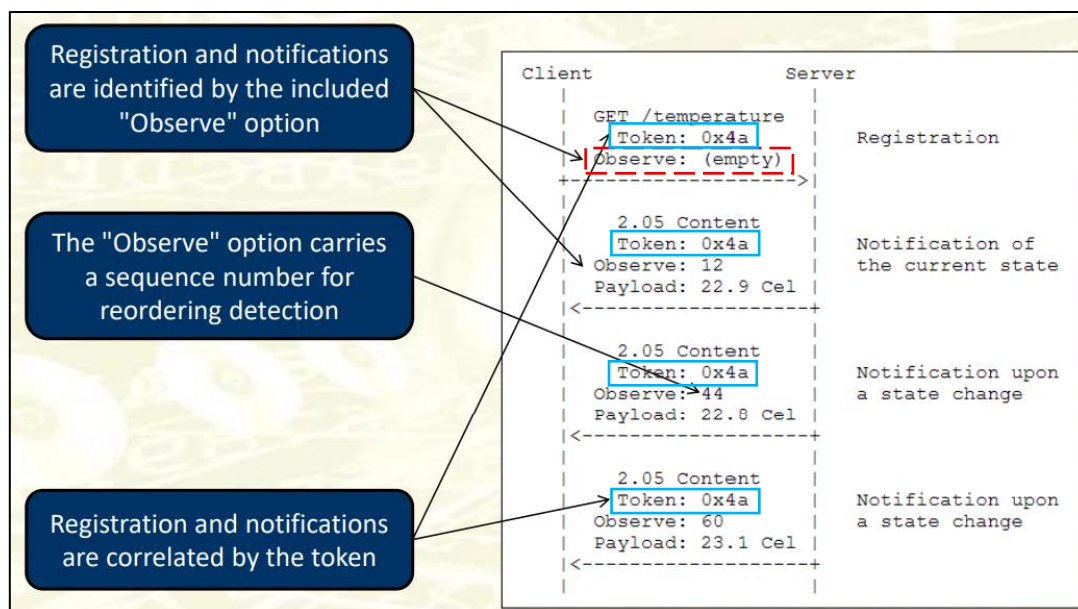
Sì, CoAP permette di gestire la situazione descritta senza *resource observing*. Supponiamo di avere un *actuator* che monitora le temperature captate da un sensore di temperatura per decidere se attivare l'aria condizionata (temperatura sale sopra un *upper threshold*) o il riscaldamento (temperatura scende sotto un *lower threshold*). Si fa *polling*, cioè richieste periodiche al sensore.

- Il client invia una richiesta GET /temperature al sensore.
- Il sensore risponde trasmettendo l'ultima temperatura captata.
- Il client rieseguita la richiesta GET /temperature ogni volta che serve un nuovo dato.

La soluzione proposta è estremamente inefficiente: invio continuo di richieste `GET /temperature`, con conseguente attesa di risposte (inefficiente sia per il client che per il server); richieste a cui segue invio degli stessi dati di risposte precedenti.



- La proposta si ispira al *publish/subscribe pattern* già visto in MQTT.
 - o Un sensore espone una risorsa, un altro nodo è interessato a tale risorsa.
 - o Il nodo segnala al sensore l'interesse verso quella particolare risorsa.
 - o Il sensore invia notifiche (*unsolicited notifications*) sugli aggiornamenti della risorsa ai nodi che hanno segnalato interesse sulla risorsa stessa.
- Il meccanismo è opzionale: si hanno *observable resources* e *non-observable resources*.
- Il client invia una *GET request* accompagnandola con un flag (chiarimento questo poco più avanti). Con questa richiesta il client si pone come *observer* presso il *server*.



- Il server pone la politica per l'invio di nuove notifiche (non è indicato nelle specifiche, ovviamente).
- **Registration.**
Il messaggio pone il contenuto della richiesta (`GET /temperature`, quello che si manderebbe anche senza il *resource observing*). In aggiunta si utilizza l'opzione *Observe* (che nella registrazione rimane vuota) per segnalare la volontà di registrazione.

- **Notifications.**

Inviata dal server al client per notificare una variazione dello stato della risorsa (in questo caso GET /temperature). Il token associa ogni notifica alla relativa registrazione (è lo stesso in tutti i pacchetti nell'esempio qua sopra). L'opzione *Observe*, che era vuota nella registrazione, viene usata per gestire l'*Ordering* (non è detto che i pacchetti vengano ricevuti nell'ordine con cui sono stati trasmessi, il numero di sequenza posto in *Ordering* permette di ricostruire l'ordine lato client).

3.4.2 Esempi di *observable resources*

Il server deve esporre le observable resources: fa ciò in una maniera che sia sensata dal punto di vista dell'applicazione.

"it is up to the server to expose observable resources that change their state in a way that is useful in application context"

Poniamo alcuni esempi di risorse che il server potrebbe mettere a disposizione del client:

- **coap://server/temperature**

Una prima idea può essere creare una risorsa che cambia il suo stato ogni secondo con l'ultima temperatura captata dal sensore. Non siamo molto convinti perché il numero di notifiche risultante è elevato e possibilmente ridondante: stiamo dando per scontato che il sensore capti un nuovo valore ogni secondo. Altro aspetto: non è detto che ci interessi ogni singola variazione di valore.

- **coap://server/temperature/critical?above=45**

Una seconda idea può essere inviare soltanto le temperature significative solo se superano una soglia critica. La soglia critica è indicata dal client nell'Uri-Query. Se la temperatura scende sotto 45° lo stato è "OK" e rimane tale: non vengono inviate ulteriori notifiche finché la temperatura rimane sotto la soglia critica definita da *above*.

- **It is up to the server to expose observable resources that change their state in a way that is useful in the application context**
- **coap://server/temperature**
 - changes its state every second to the current reading of the temperature sensor
- **coap://server/temperature/felt**
 - changes its state to "cold" when the temperature reading drops below a certain pre-configured threshold, and to "warm" when the reading exceeds a second, slightly higher threshold
- **coap://server/temperature/critical?above=45**
 - changes its state based on the client-specified parameter value: every second to the current temperature reading if the temperature exceeds the threshold, or to "OK" when the reading drops below

3.4.3 Consistency

Il client che ha stabilito una connessione *observing* col server deve tenere in considerazione che i dati ricevuti dal server potrebbero non essere fresh al 100%.

- To what extent the client view is in sync with the resource state?
 - Latency, lost and/or reordered notifications, erroneous registration cancellation
- **Best effort** approach
 - As soon as possible
 - As many states as possible
- Bound the out-of-sync time duration with a **maximum age** attached to notifications

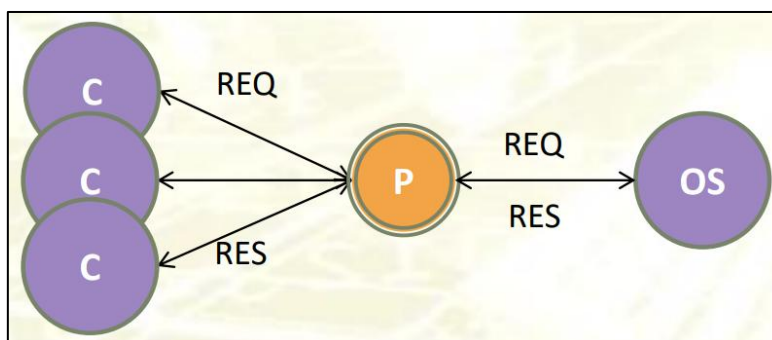
- Siamo in una LLN (che è *lossy*): alcune notifiche potrebbero non arrivare a destinazione.
- Le notifiche potrebbero essere ricevute in un ordine diverso da quello di trasmissione (possibilissimo in una *multi-hop communication*).

L'approccio è chiaramente *best-effort* (nel senso che si cerca di portare il maggior numero di pacchetti a destinazione, se poi questo non succede non è un problema). Come gestiamo questa cosa?

- Recuperiamo il valore *Max-Age*, utilizzabile per segnalare la durata della validità dell'informazione fornita. Il client verifica il valore di *Max-Age* e sa che non ha senso inviare ulteriori richieste sulla stessa risorsa finché l'informazione è ancora valida. Riduzione del numero di pacchetti trasmessi.
- Utilizziamo il *Token* per ricostruire l'ordine dei pacchetti: se l'ultimo pacchetto ricevuto ha Token 5 e ne riceviamo uno con Token 3 allora questo ultimo può essere scartato (è un'informazione vecchia, superata da dati più recenti).

3.4.4 Observing and intermediaries

Gestire il *resource observing* non è cosa sicuramente leggera: pensiamo che ogni server deve creare una struttura dati per ogni *observable resource*, memorizzando gli elenchi dei client che si sono registrati presso le varie risorse. Questo è un problema su *constrained devices*, specialmente se il numero di *observers* è significativo.



Possiamo introdurre un proxy, spostando così parte della complessità in nodi diversi dai *constrained devices*.

Siamo in *Reverse Proxy*: il proxy individua le risorse messe a disposizione all'interno della rete e le espone come proprie a soggetti esterni.

- I Clients C desiderano ricevere notifiche su variazioni di stato di risorse presso l'*Origin Server* OS.
- L'*Origin Server* OS è un *constrained device*.

- Introduciamo il Proxy P all'interno della rete.
 - o L'*Origin Server* OS deve gestire unicamente la registrazione del Proxy P (rispetto a quella risorsa). Viene mandata una notifica di variazione dello stato della risorsa solo al proxy.
 - o Le registrazioni dei Client C sono gestite dal Proxy P e non dall'*Origin Server* OS! Il proxy inoltra le notifiche di variazione di stato ricevute dall'*Origin Server* OS ai Clients C.

3.5 [Novità] *Discovery*

3.5.1 Introduzione

La funzionalità di *discovery* è un'altra novità introdotta in CoAP (non presente in HTTP). Essa permette l'individuazione dei dispositivi IoT presenti nella rete e per ciascuno di essi le risorse disponibili. Tali passaggi devono avvenire in modo automatico, senza l'intervento umano.

Si articola in: *service discovery* e *resource discovery*.

3.5.2 *Service discovery*

Service discovery significa individuare tutti i server che implementano CoAP all'interno di una certa LAN!

- CoAP ha allocato un indirizzo multicast IPv6, noto nello standard come "*All CoAP Nodes*". Inviare pacchetti a questo indirizzo significa inviarli a tutti i nodi presenti nella rete.
- L'applicazione invia una richiesta multicast a tutti i nodi presenti nella LAN.
 - o I nodi che non implementano CoAP ignorano la richiesta.
 - o I nodi che implementano CoAP devono rispondere alla richiesta informandoli della loro esistenza.
- Limite: *multicast messaging* è presente solo nelle LAN. Nel caso in cui si esca da questo confine (Esempio: Cloud Computing) è necessario cambiare approccio (prevista alternativa nello standard, non la vedremo).
- Quanto spiegato è l'equivalente del *Broker* nella *service-oriented architecture*.

- How to discover CoAP servers?
- Multicast CoAP + "All CoAP Nodes" multicast address
 - Discovery is bound to the link-local scope!
- Well-known default port number: 5683

3.5.3 *Resource discovery e CoRE Link Format*

Il passaggio successivo all'individuazione dei CoAP servers (*service discovery*) è trovare, per ciascun server, l'elenco delle risorse disponibili.

- How to discover which resources are hosted by the service?
 - URIs
 - Format and interface
- Well-known relative URI **"/.well-known/core"**
 - default entry point for requesting the list of links about resources hosted by a server
- CoRE Link Format to describe links (resources) hosted by the server

- Ogni CoAP server deve esporre la risorsa `/.well-known/core` (**è obbligatorio**).
- Tale risorsa è utilizzata ai fini della *resource discovery*: viene lanciata una richiesta `GET /.well-known/core`, la risposta del server contiene l'elenco delle risorse disponibili all'interno del relativo payload.

- L'elenco delle risorse è restituito adottando *CoRE Link Format* come *data-encoding language*. List of resources along with a short description of functionalities/data that can be provided.

Link attributes



- **Resource Type ('rt')**
 - Opaque string used to assign an application specific semantic type to a resource
- **Interface Description ('if')**
 - opaque string used to provide specific interface definition used to interact with the target resource
 - E.g.: the URI of a Web Application Description Language
- **Maximum Size Estimate ('sz')**
 - an indication of the maximum size of the resource representation
- **Content Format ('ct')**
 - provides a hint about the Content-Formats this resource returns
- **Observable ('obs')**
 - a hint indicating that the destination of a link is useful for observation

- Con *Resource Type* (*rt*) si attribuisce un “significato semantico” alla risorsa (possiamo dire che la risorsa rappresenta una temperatura in gradi celsius, un led, un contatore d’acqua...)
- Con *Interface Description* (*if*) descriviamo l’interfaccia (Esempio: “sensor” o “actuator”, questa cosa permette di capire quali operazioni possono essere fatte – GET su sensor, POST e PUT su actuator).

REQ: GET /.well-known/core

RES: 2.05 Content

</sensors/temp>;if="sensor",

</sensors/light>;if="sensor"

REQ: GET /.well-known/core

RES: 2.05 Content

</sensors>;ct=40

REQ: GET /sensors

RES: 2.05 Content

</sensors/temp>;rt="temperature-c";if="sensor",

</sensors/light>;rt="light-lux";if="sensor"

REQ: GET /.well-known/core?rt=light-lux

RES: 2.05 Content

</sensors/light>;rt="light-lux";if="sensor"

Media type	Encoding	Id.
text/plain;	-	0
charset=utf-8	-	
application/	-	40
link-format	-	
application/xml	-	41
application/	-	42
octet-stream	-	
application/exi	-	47
application/json	-	50

- Possibile filtrare l’elenco delle risorse al momento della richiesta.

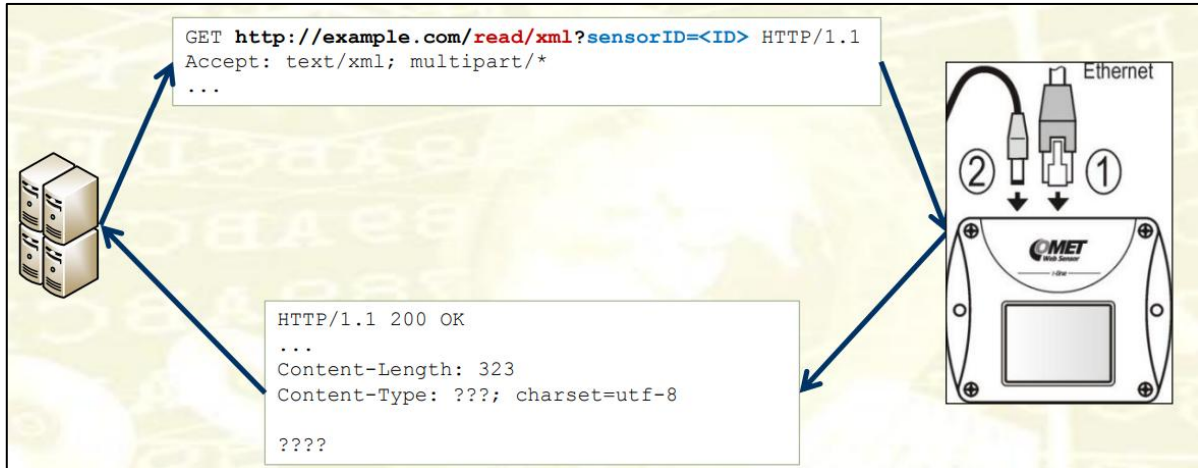
- The resource **"/.well-known/core"** is meant to be used to return links to the entry points of resource interfaces on a server
- Filtering may be performed
 - GET /.well-known/core?name=value
 - Name: 'href' (target resource) or attribute name
- Examples
 - ?href=/foo
 - ?href=/foo*
 - ?rt=temperature-c

4 INTEROPERABLE DATAENCODING

4.1 Introduzione

Abbiamo introdotto nei capitoli precedenti due protocolli application-layer per la trasmissione dei dati: MQTT e CoAP. Tali protocolli non si esprimono rispetto alla codifica dell'informazione contenuta all'interno del payload.

La scelta è voluta: CoAP deve supportare molteplici *use-cases* (ulteriori vincoli non aiuterebbero).



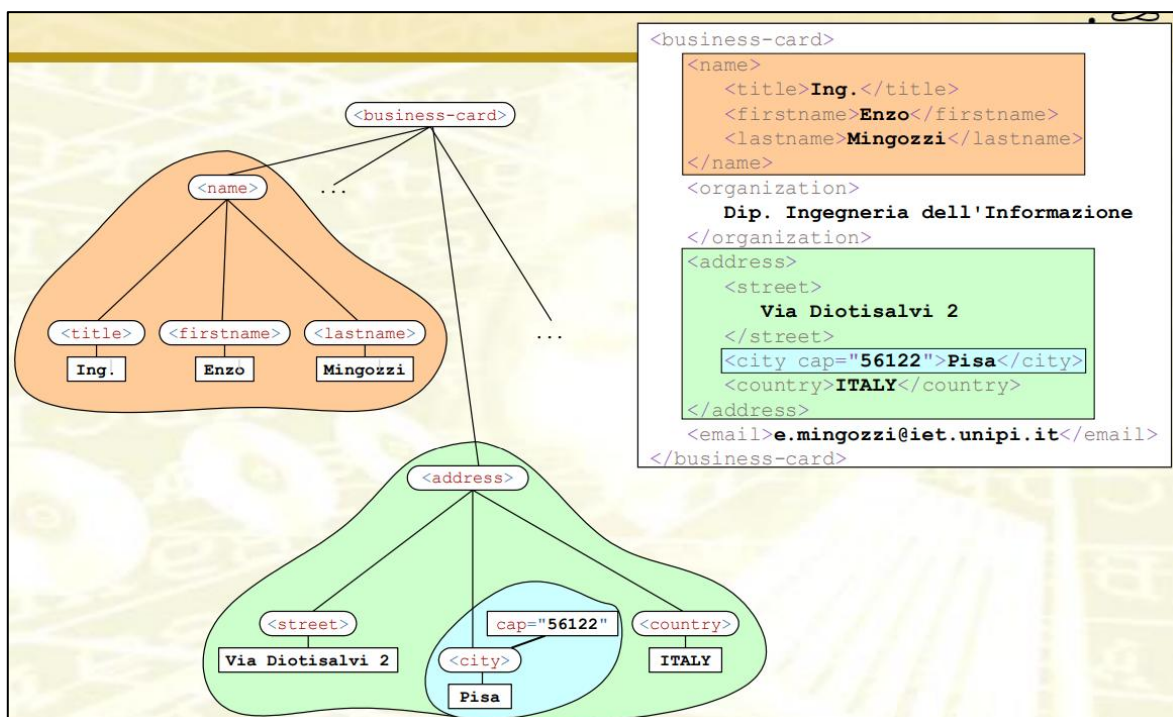
Questo non significa che non si debba stabilire un formato, specialmente se pensiamo all'interoperabilità. Immaginiamo un sistema IoT di grandi dimensioni: se non si pone un minimo di standardizzazione ogni dispositivo stabilirà le sue regole, rendendo difficile il dialogo tra essi.

Procediamo introducendo i due principali *data-encoding languages* (dal Cloud): XML e JSON! Attenzione anche a EXI e CBOR, pensati per l'IoT.

4.2 [Dal Cloud] XML

4.2.1 Introduzione

XML è un *markup language*. La codifica dei dati avviene in forma di elementi (elements), organizzati in maniera gerarchica (come evidente dall'immagine)



- An XML document is a sequence of characters
- A character is an atomic unit of text
 - Tab, Carriage Return, Line Feed
 - Legal characters of Unicode and ISO/IEC 10646
- A document consists of intermingled character data and markup
 - *Markup* takes the form of
 - Elements: `<tag>...</tag>`, `<empty_tag />`
 - Entity references: `&entity_name;`
 - Comments: `<!-- any text -->`
 - Instructions: `<?processing_instruction data ?>`
 - CDATA sections: `<![CDATA[ ... ]]>`
 - ...
 - *Character data*: all the text that is not markup

- Un documento XML è una sequenza di caratteri, il carattere è l'unità atomica del testo. Il carattere può essere: uno di quelli speciali (TAB, CR, LF) o uno della codifica Unicode.
- L'*element* contiene l'informazione.
 - o È caratterizzato da uno *start tag* e un *end tag* (devono esserci entrambi). Entrambi contengono il nome dell'elemento.
`<tag_name> ... </tag_name>`
 - o Possibile porre un elemento senza contenuto ponendo `<tag_name/>`
 - o Possono essere posti attributi all'interno di uno start tag
`<tag_name attribute_name="attribute_value" ... > ... </tag_name>`
 - o Possibile includere *elements* all'interno di altri elements.
 - o Possibile inserire riferimenti ad altre entità (presenti nel documento o addirittura in documenti esterni) usando la e commerciale: `&entity_name`.
- Possibile introdurre *CDATA sections*, cioè sezioni che contengono *raw data* (sezioni non organizzate con regole e specifiche di XML).

- To skip XML syntax rule application on a piece of text
 - CDATA sections cannot be nested into each other
- CDATA section syntax
 - Start tag `<![CDATA[`
 - Any sequence character other than `]]>`
 - End Tag `]]>`

```
<sample>
  if ( this->getX() < 5 && v[0] != 3 )
    cerr << this->displayError();
</sample>
<sample>
  <![CDATA[
    if ( this->getX() < 5 && v[0] != 3 )
      cerr << this->displayError();
  ]]>
</sample>
```

4.2.2 Well-formed document

Un documento XML è *well-formed* se tutte le regole sono soddisfatte.

- **Sintassi.**

Abbiamo introdotto nella sezione precedente le regole base per la scrittura di un documento XML. Se tali regole non sono rispettate non è possibile fare il *parsing* del documento, con conseguenti errori. Esempio dove *start tag* ed *end tag* (relativi alla città) non coincidono per colpa di un *typo*:



- **Use-case.**

XML fornisce meccanismi che permettono di verificare se il documento è sensato non solo rispetto alla sintassi, ma anche rispetto allo *use-case* (Esempio: se dentro `<address>...</address>` ci sono due elementi `city` questo non ha molto senso, dato che la città associata a un indirizzo è una sola – si può segnalare la cardinalità dell'elemento). Tali schemi non sono obbligatori!



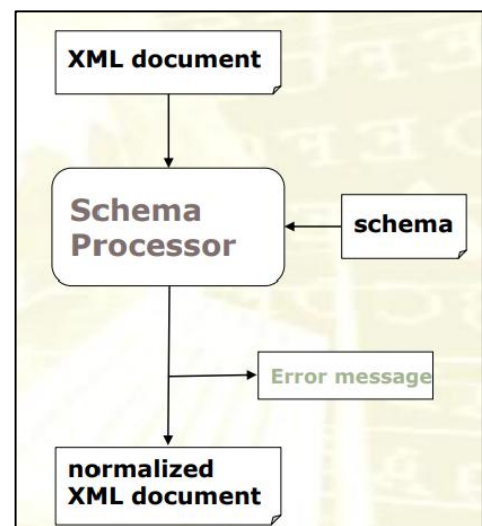
4.2.3 Document schema

Lo schema consiste in informazioni aggiuntive che possono essere associate (facoltativo) al documento XML per verificare se questo ha senso rispetto a un particolare *use-case*.

Use case → Schema

Esempio principale: XML schema, un set di costrutti attraverso cui il developer può definire lo schema e stabilire regole.
Esempio: stabilire che *address* deve prevedere al suo interno una e una sola istanza di *city*.

Consideriamo il seguente esempio:



```

<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        targetNamespace="http://businesscard.org">

  <element name="card" type="b:card_type"/>
  <element name="name" type="string"/>
  <element name="title" type="string"/>
  <element name="email" type="string"/>
  <element name="phone" type="string"/>
  <element name="logo" type="b:logo_type"/>

  <complexType name="card_type"> Definizione tipo complesso
    <sequence>
      <element ref="b:name"/>
      <element ref="b:title"/>
      <element ref="b:email"/>
      <element ref="b:phone" minOccurs="0"/>
      <element ref="b:logo" minOccurs="0"/>
    </sequence>
  </complexType>

  <complexType name="logo_type"> Definizione tipo complesso
    <attribute name="url" type="anyURI"/>
  </complexType>

</schema>

```

- Definizione dei possibili elementi presenti all'interno del documento XML: *card*, *name*, *title*, *email*, *phone* e *logo*.
- *logo* e *card* sono tipi complessi definiti all'interno dello schema XML, gli altri sono tipi *string*.
- Possibile definire proprietà per gli elementi definiti all'interno dello schema.
 - o Esempio 1: *minOccurs* su *phone* e *logo*, segnale col valore 0 che tali elementi sono facoltativi. I rimanenti (su cui non si è opposto *minOccurs*) sono obbligatori.
 - o Esempio 2: segnale che l'attributo *url* del tipo complesso
 - o Possibile ulteriore esempio: elemento *address* visto nell'esempio della pagina precedente, pongo su *city* *maxOccurs* uguale a 1 per indicare che ce ne deve essere uno e uno solo.
 - o Ovviamente $\text{minOccurs} \leq \text{maxOccurs}$

Come associamo il documento XML allo schema XML appena definito? Con l'attributo *schemaLocation*, posto rispetto a un particolare elemento del documento stesso (nell'esempio *card*).

```

<card xmlns="http://businesscard.org"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://businesscard.org business_card.xsd">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 456-1414</phone>
  <logo url="widget.gif"/>
</card>

```

- The schema for a document is specified by the value of the attribute ***schemaLocation***
 - By inserting this attribute, the author asserts that the instance document is intended to be valid with respect to the schema
- The attribute can be specified in any element besides the document element

Il destinatario del documento XML può scaricare lo schema e verificare che il messaggio sia rispettoso.

Ogni volta che avviene uno scambio di pacchetti tra due nodi (non solo da dispositivo IoT ad applicazione, ma anche viceversa) è necessario verificare la validità del documento XML posto all'interno del payload.

- Questi controlli non hanno molta utilità se l'intero sistema è progettato da un unico sviluppatore: è chiaro che la presenza di un unico sviluppatore garantisce sulla coerenza nelle interazioni tra moduli.
- Nella pratica ciò non succede mai: abbiamo moduli implementati da sviluppatori diversi.
- XML è complesso e ridondante, ma aiuta molto dal punto di vista dell'interoperabilità. Aiuta nel far emergere le cause di errori a seguito di modifiche sui moduli che compongono il sistema, specialmente i più complessi.

Si invita a lettura delle diapositive di Vallati per maggiori dettagli, ha sorvolato da 23 a 28

La necessità di avere qualcosa di più semplice porta all'adozione di un altro *data-encoding language*: JSON!

4.3 [Dal Cloud] JSON

JSON è acronimo di *JavaScript Object Notation*. L'idea è definire un *data-encoding language* non ridondante e semplice da interpretare.

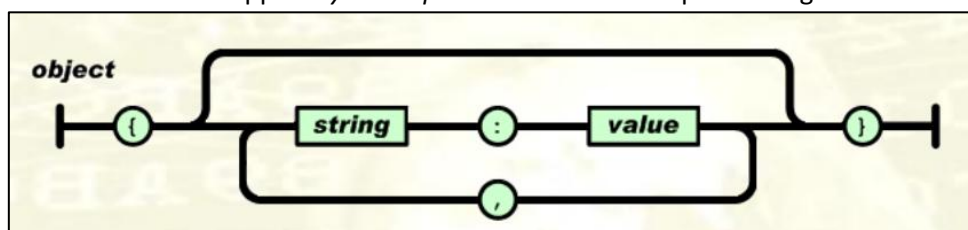
```
{
  "Image": {
    "Width": 800,
    "Height": 600,
    "Title": "View from 15th Floor",
    "Thumbnail": {
      "Url": "http://www.example.com/image/481989943",
      "Height": 125,
      "Width": "100"
    },
    "IDs": [116, 943, 234, 38793]
  }
}
```

```
[
  {
    "precision": "zip",
    "Latitude": 37.7668,
    "Longitude": -122.3959,
    "Address": "",
    "City": "SAN FRANCISCO",
    "State": "CA",
    "Zip": "94107",
    "Country": "US"
  },
  {
    "precision": "zip",
    "Latitude": 37.371991,
    "Longitude": -122.026020,
    "Address": "",
    "City": "SUNNYVALE",
    "State": "CA",
    "Zip": "94085",
    "Country": "US"
  }
]
```

JSON si basa su due particolari strutture:

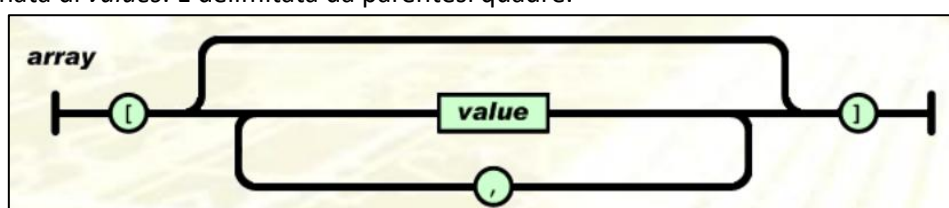
- **Oggetto.**

Collezione non ordinata di coppie *key-value pairs*. È delimitata da parentesi graffe.



- **Array.**

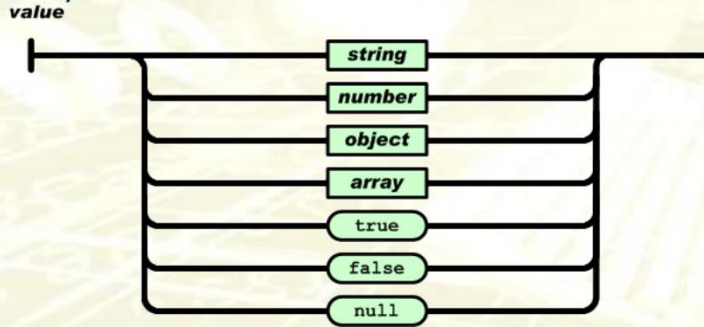
Lista ordinata di *values*. È delimitata da parentesi quadre.



Regole di sintassi molto semplici, nessuno schema da definire e/o controllare.

È comunque possibile dare luogo a strutture complesse!

- **Value:** can be
 - a string (in double quotes), a number, **true**, **false**, **null**
 - – an object, or an array (these structures can be nested)



- Un valore può essere un oggetto (quindi posso avere una lista di oggetti, un oggetto costituito a sua volta da oggetti, ...).
- Un valore può essere una lista (quindi posso avere liste di liste, oggetti con all'interno liste, ...).

Ma non esistono schemi anche per JSON (chiede Saverio)? Sono state definite cose successivamente, ma non sono parte del core di JSON. Sono nati dopo.

4.4 Scelta tra JSON e XML: come?

Come scelgo tra XML o JSON?

- **XML** è preferito quando si ha un progetto molto grande e si deve avere garanzie sull'*interoperability*.
- Se il progetto è piccolo e coinvolge pochi sviluppatori allora **JSON** è l'opzione raccomandata.

4.5 [Novità] Accenni su EXI e CBOR

Come scelgo XML o JSON?

- XML è l'opzione raccomandata con progetti molto grandi e un numero elevato di persone coinvolte, in generale un contesto dove si deve garantire la *interoperability*.
- JSON è l'opzione raccomandata con progetti piccoli e pochi soggetti coinvolti.

Nonostante questo non possiamo ritenerci soddisfatti: entrambi i linguaggi comportano overhead significativo. Se la codifica è *text-based* il dispositivo IoT deve investire molto tempo nel fare *parsing* dell'intero testo (molto spesso per estrarre una piccola informazione). Questa cosa è problematica in *constrained device*.

Si propongono due nuovi *data-encoding language*, pensati proprio per l'IoT:

- **EXI**, per rappresentazione binaria di un documento in formato XML;
- **CBOR**, per rappresentazione binaria di un documento in formato JSON.

Non ci interessano i dettagli, ma sottolineiamo come questo passaggio migliori notevolmente l'efficienza nella lettura.

Encoding	Complexity	RDF	Smart Energy	SensorML
XML	Medium	206	409	300
EXI*	Low	6 (3%)	13 (3%)	57 (19%)
BXML	Medium	177 (86%)	210 (51%)	177 (59%)
Fast Infoset	Medium	143 (69%)	200 (49%)	185 (62%)

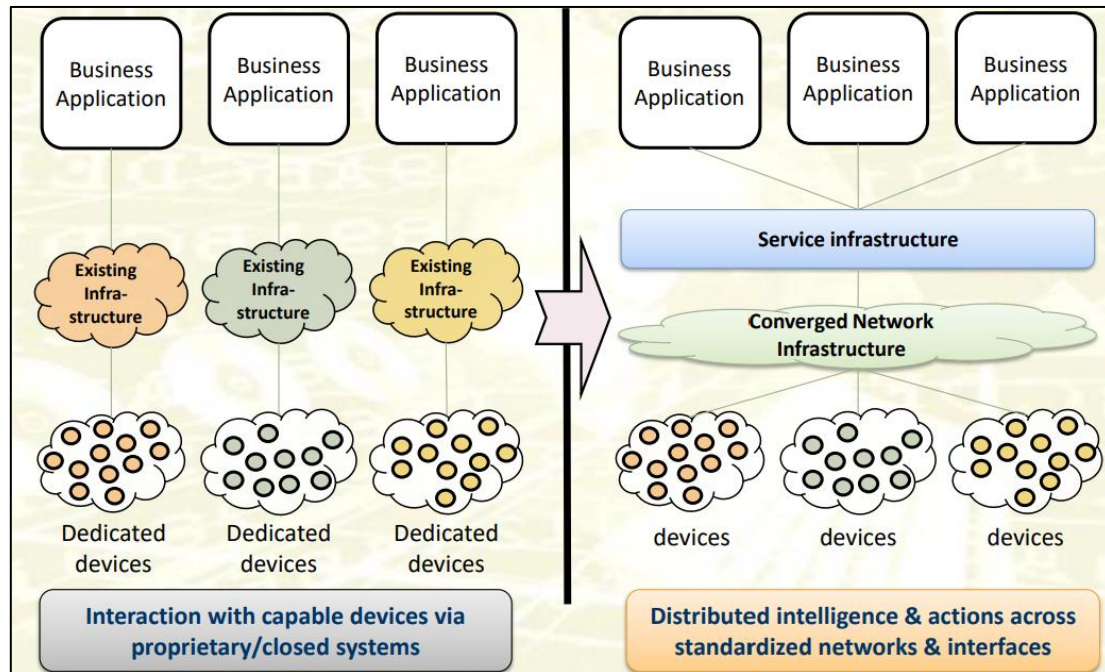
* Strict schema-informed mode with bit alignment

Table 1. Comparison of encoding results (size of encoded objects in Bytes).

5 SENSOR DATA REPRESENTATION AND MODELS

5.1 Introduzione e nuovo problema

La visione è sempre la stessa: muoverci da un'architettura verticale (a sinistra) a un'architettura orizzontale (a destra) dove tutti i dispositivi fanno riferimento a un'infrastruttura comune, anche quando prodotti da soggetti diversi: un'ecosistema!



Vogliamo favorire:

- **Interoperability.**
Le applicazioni poste in questo ecosistema devono essere in grado di comunicare tra loro.
- **Automated Discovery.**
Servono meccanismi che permettano a un'applicazione di individuare quali nodi sono disponibili all'interno della rete. Per ciascun sensore l'applicazione vuole conoscere le risorse che i sensori presenti nella rete offrono.

Abbiamo già introdotto alcuni blocchi per poter perseguire il nostro obiettivo:

- protocolli per gestire la comunicazione tra applicazioni e dispositivi IoT;
- *encoding languages* che definiscono la struttura del messaggio.

Mancano ancora due blocchi: *data semantic* e *automated discovery*. Ci poniamo su livelli superiori rispetto a quanto già visto: non ci interessa l'aspetto tecnico (già affrontato), ma l'aspetto semantico!

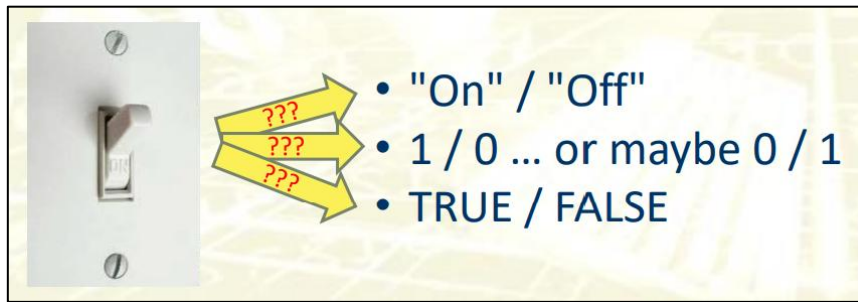
5.2 Data semantic

5.2.1 Problema

Abbiamo già visto i principali *data encoding languages* (XML e JSON) per l'organizzazione dei dati all'interno del documento.

- Un conto è ricevere i dati e interpretare la "struttura tecnica" del documento (cosa già vista con XML, JSON, ...).
- Un'altra cosa è dare un significato ai dati in modo tale da permettere al dispositivo di svolgere azioni (tenendo conto dei dati stessi). Vogliamo stabilire un *common ground* tra nodi (sul come devono essere interpretati i dati).

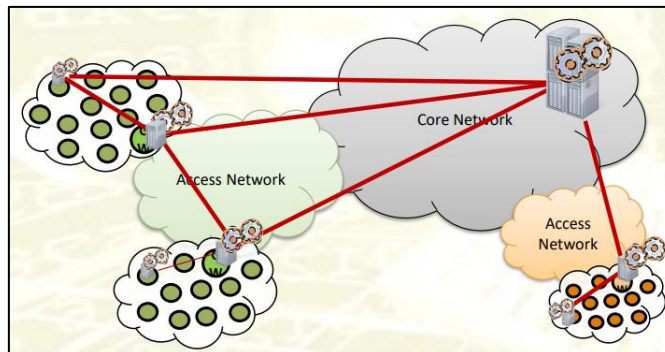
Introduciamo il seguente esempio:



- Il *data encoding language* adottato è influente: può essere XML come JSON.
- Vogliamo rappresentare lo stato della lampada (manipolabile per mezzo dell'interruttore) all'interno del documento. Come poniamo questa cosa?
 - o "On" / "Off" ?
 - o 1 / 0 ?
 - o True / False?

È chiaro che esistono molteplici modi per rappresentare questa cosa nel documento.

- Un *vendor* potrebbe adottare una strada, un secondo *vendor* un'altra strada, e così via...
- *sender* e *receiver* devono concordare sul significato attribuito ai dati.



Il problema non viene affrontato nei protocolli già noti perché il numero di possibili *use-cases* è elevato. Quello che faremo adesso è introdurre un *common ground* relativo a un particolare *use-case* (quello che ci interessa maggiormente, si pensi al progetto): *data reporting / telemetry*.

5.2.2 Proposta per lo *use-case data reporting*: Sensor Markup Language (SenML)

5.2.2.1 Spiegazione

SenML (acronimo di *Sensor Markup Language*) è uno standard introdotto per garantire un *common ground* semantico rispetto allo *use-case* del *data reporting / telemetry*.

- I dati sono organizzati utilizzando uno dei *data encoding language* noti (XML, JSON,)
- *SenML* descrive come questi dati devono essere interpretati da un punto di vista semantico.

Poniamo quanto segue:

```
{ "e": [
  { "n": "voltage", "u": "V", "v": 120.1 },
  { "n": "current", "t": -5, "v": 1.2 },
  { "n": "current", "t": -4, "v": 1.30 },
  { "n": "current", "t": -3, "v": 0.14e1 },
  { "n": "current", "t": -2, "v": 1.5 },
  { "n": "current", "t": -1, "v": 1.6 },
  { "n": "current", "t": 0, "v": 1.7 } ],
  "bn": "urn:dev:mac:0024beffffe804ff1/",
  "bt": 1276020076,
  "ver": 1,
  "bu": "A"
}
```

Tue Jun 8 18:01:16 UTC 2010

- I dati sono posti all'interno di un oggetto.
- All'interno dell'oggetto sono presenti:
 - o un set di attributi opzionali (bn, bt, ver, bu, ...);
 - o un attributo obbligatorio (e, array di entries – almeno una).

Approfondiamo quali sono gli attributi previsti:

- **Attributi non relativi alle singole entries in e.**

Abbiamo i seguenti attributi:

Base Name (optional)	A string that is prepended to the names found in the entries
Base Time (optional)	A base time that is added to the time found in an entry
Base Units (optional)	A base unit that is assumed for all entries, unless otherwise indicated
Version (default = 1)	Version number of media type format
Measurement or Parameter Entries Array (mandatory)	Values for sensor measurement or other generic parameters (such as configuration parameters)

- o **Measurement or Parameter Entries Array, obbligatorio.**
Lo abbiamo già visto (e), contiene le misurazioni trasmesse col pacchetto.
- o **Base Name (bn), opzionale.**
Nome associato alle misurazioni.
- o **Base Time (bt), opzionale.**
Istante temporale rappresentato in Epoch. Per ogni misurazione si prende il base time e si somma il relativo parametro t per ottenere l'istante temporale in cui è stata ottenuta tale misurazione

$$\text{measurement_time} = \text{bt} + t$$

```

{ "e": [
  { "n": "voltage", "u": "V", "v": 120.1 },
  { "n": "current", "t": -5, "v": 1.2 },
  { "n": "current", "t": -4, "v": 1.30 },
  { "n": "current", "t": -3, "v": 0.14e1 },
  { "n": "current", "t": -2, "v": 1.5 },
  { "n": "current", "t": -1, "v": 1.6 },
  { "n": "current", "t": 0, "v": 1.7 }],
  "bn": "urn:dev:mac:0024beffffe804ff1/",
  "bt": 1276020076,
  "ver": 1,
  "bu": "A"
}

```

Tue Jun 8 18:01:16 UTC 2010

Se t non viene definito in particolari misurazioni allora $\text{measurement_time} = \text{bt}$. Se bt non viene definito allora t non è offset (quindi si pone l'Epoch direttamente in ogni t).

- o **Base Units (bu), opzionale.**
Unità di misura adottata nelle misurazioni.
- o **Version.**
Versione di SenML, mettere 1 senza starci a pensare troppo.

- **Attributi relativi alle singole entries in e.**

Per ogni misurazione posta all'interno dell'array e indichiamo:

Name (optional if Base Name is present)	Name of the sensor or parameter. The Name attribute concatenated to the Base Name attribute must result in a <u>globally unique identifier</u> for the resource (an URI is recommended).
Units (Optional)	Units for a measurement value
Value (Optional if a Sum value is present)	Value of the entry. Only three basic data types: - Floating point numbers ("v" field for "Value") - Booleans ("bv" for "Boolean Value") and - Strings ("sv" for "String Value")
Sum (optional if a Value is present)	Integrated sum of the values over time
Time (optional)	Base Time + Time = Time when value was recorded (if zero, the time is roughly "now")
Update Time (optional)	A time in seconds that represents the maximum time before this sensor will provide an updated reading for a measurement

○ **Name (n), obbligatorio se non è stato indicato Base Name.**

Nome associato alla singola misurazione. La concatenazione tra Base Name e Name deve restituire un identificativo univoco della misurazione.

○ **Units (u), opzionale.**

Unità di misura utilizzata per quella particolare misurazione. Possibile:

- Non definire la *Base Unit* e stabilire l'unità di misura singolarmente, per ogni misurazione.

```
{ "e": [
  { "n": "temperature", "v": 27.2, "u": "Cel" },
  { "n": "humidity", "v": 80, "u": "%RH" } ],
  "bn": "http://[2001:db8::2]/",
  "bt": 1320078429,
  "ver": 1
}
```

- Definire la *Base Unit* (bu) e al tempo stesso stabilire l'unità di misura di una particolare misurazione (u). In tal caso u sovrascrive bu.

```
{ "e": [
  { "n": "voltage", "u": "V", "v": 120.1 },
  { "n": "current", "t": -5, "v": 1.2 },
  { "n": "current", "t": -4, "v": 1.30 },
  { "n": "current", "t": -3, "v": 0.14e1 },
  { "n": "current", "t": -2, "v": 1.5 },
  { "n": "current", "t": -1, "v": 1.6 },
  { "n": "current", "t": 0, "v": 1.7 } ],
  "bn": "urn:dev:mac:0024beffffe804ff1/",
  "bt": 1276020076,
  "ver": 1,
  "bu": "A"
}
```

Tue Jun 8 18:01:16 UTC 2010

○ **Time (t), opzionale.** Già spiegato nella pagina precedente.

○ **Valore (v), obbligatorio.** Valore della misurazione.

Nota sulle unità di misura. SenML prevede unità di misura relative, ad esempio %EL e %RH

%RH	Percentage (Relative Humidity)	float	[RFC8428]
%EL	Percentage (remaining battery energy level)	float	[RFC8428]

<https://www.iana.org/assignments/senml/senml.xhtml>

```

{ "sv": "E 24' 30.623", "u": "lon", "t": 120 },
{ "sv": "N 60' 7.966", "u": "lat", "t": 120 },
{ "v": 98.0, "u": "%EL", "t": 150 },
{ "v": 21.2, "t": 180 },
{ "sv": "E 24' 30.628", "u": "lon", "t": 180 },
{ "sv": "N 60' 7.967", "u": "lat", "t": 180 }],
"bn": "http://[2001:db8::1]",
"bt": 1320067464,
"bu": "%RH"
}

```

5.2.2.2 Esempio: misurazione di voltaggio e corrente

Abbiamo già riportato nelle pagine precedenti il seguente esempio:

```

{ "e": [
  { "n": "voltage", "u": "V", "v": 120.1 },
  { "n": "current", "t": -5, "v": 1.2 },
  { "n": "current", "t": -4, "v": 1.30 },
  { "n": "current", "t": -3, "v": 0.14e1 },
  { "n": "current", "t": -2, "v": 1.5 },
  { "n": "current", "t": -1, "v": 1.6 },
  { "n": "current", "t": 0, "v": 1.7 }],
"bn": "urn:dev:mac:0024beffffe804ff1/",
"bt": 1276020076,
"ver": 1,
"bu": "A"
}

```

Tue Jun 8 18:01:16 UTC 2010

- L'oggetto contiene al suo interno misurazioni di corrente: l'unità di misura di base `bu` è A (Ampere)
- Tra le misurazioni, in realtà, è presente anche un voltaggio (la misurazione dove è indicato come `u` V, Volt). La cosa è segnalata anche dal campo `n` (che distingue le misurazioni di corrente da quella della tensione)
- L'oggetto contiene, nel suo complesso, le ultime sei misurazioni di corrente (attenzione ai valori di `t`, negativi!!).

Applicazione di SenML in XML. Omessa

5.3 Automated Discovery: IPSO Smart Objects

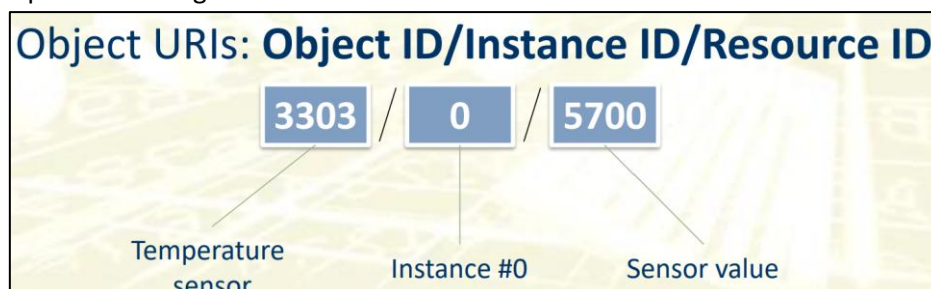
L'*automated discovery* è la capacità di un'applicazione di scoprire i dispositivi presenti, ma anche le funzionalità che ogni dispositivo può offrire per mezzo di interfacce. Si va oltre le funzionalità di discovery già introdotta in CoAP:

- In CoAP forniamo un elenco di risorse disponibili;
- qua siamo interessati a un meccanismo che permetta un'interpretazione semantica della risorsa.

La *IPSO Alliance* ha elaborato la convenzione *IPSO Smart Objects*, molto interessante per la sua semplicità.

- Si cerca di includere le informazioni direttamente nell'URI, cosa perfettamente possibile dato che lo standard alla base di URI è molto flessibile (a parte la sintassi dell'indirizzo).
- Lavoriamo con CoAP e invochiamo una risorsa per mezzo di un indirizzo del tipo `coap://ip_address/part`

Dove `part` presenta la seguente struttura



- **Object ID.**
ID che identifica un gruppo di dispositivi (una certa categoria, esempi: *temperature sensors, smart meters, smart bulbs, ...*)
 - **Resource ID.**
Ogni oggetto/dispositivo è caratterizzato da un set di risorse. Ogni risorsa deve essere identificata.
 - **Instance ID.**
Valore incrementale utilizzato quando sono presenti molteplici sensori all'interno di un particolare oggetto. Solitamente succede con schede particolarmente complesse (se non succede il valore tipico è 0).
- I possibili valori di Object ID e Resource ID sono recuperabili da un database fornito dalla IPSO Alliance. L'applicazione visualizza un particolare ID e accede alla relativa interpretazione semantica visitando il database.

Consideriamo il seguente esempio:

• Example of object spec: Light Control							
Object	Object ID	Object URN	Multiple Instances?		Description		
IPSO Light Control	3311	urn:oma:lwm2m:ext:3311	Yes		Light control object with on/off and optional dimming and energy monitor		

Resource Name	Resource ID	Access Type	Mandatory	Type	Range or Enumeration	Units	Descriptions
On/Off	5850	R, W	Mandatory	Boolean			On/off control, 0=OFF, 1=ON
Dimmer	5851	R, W	Optional	Integer	0-100	%	Proportional control, integer value between 0 and 100 as a percentage.
Colour	5706	R,W	Optional	String		Defined by "Units" resource	A string representing a value in some color space
Units	5701	R	Optional	String			Measurement Units Definition e.g. "Cel" for Temperature in Celsius.
On Time	5852	R, W	Optional	Integer		s	The time in seconds that the light has been on. Writing a value of 0 resets the counter.

- L'oggetto è il controllore di una luce, identificato con Object ID 3311.
- All'interno dell'oggetto deve essere prevista obbligatoriamente la risorsa On/Off, identificata da Resource ID 5850: questo perché una luce spenta difficilmente può offrire funzionalità.
- Le altre risorse sono opzionali, ma se previste devono essere identificate con i resource ID indicati.
- Per ogni risorsa si indicano tutte le informazioni necessarie (si osservi la descrizione, l'unità di misura, il range, il tipo...)