ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Laurea Magistrale in Ingegneria Informatica

TESI DI LAUREA

In

Sistemi Mobili M

Estensione del Framework Kura per Internet of Things tramite Supporto Distribuito CoAP.

CANDIDATO	RELATORE:
Gianvito Morena	Prof. Ing. **Paolo Bellavista**
	CORRELATORE/CORRELATORI
	Ing. **Carlo Giannelli**
	Ing. **Alessandro Zanni**

Anno Accademico 2014/2015

Sessione I

Indice

Introduzione 5
Internet of Things: Requisiti e Scenari Applicativi 7
IoT 7
Obiettivi 10
Requisiti 11
Ambito Smart City 11
Progetti Smart City esistenti 17
Tecnologie utilizzate 25
CoAP 25
Definizione 25
Architettura 25
Messaggi 26
Discovery 49
Comunicazione multicast 49
Security 51
Sottoscrizioni e notifiche 57
Esempi di comunicazione 64
Implementazioni disponibili 67
Approfondimento 76
MQTT 79
Definizione 79
Architettura 80
Messaggi 80
Topic matching 85
Application level QoS 85
Last will and testament 87
Persistence 87
Broker MQTT 87
MQTT-SN 88
Confronto tra CoAP e MQTT 97
Kura 100

Everyware Cloud 102
Progettazione 105
Introduzione 105
Requisiti 107
Considerazioni su Californium 107
Resource Directory 111
CoAPTreeHandler 114
Gestione dinamica delle risorse 115
QoS 117
Implementazione 119
Introduzione 119
Bundle 119
Service base 119
Californium Service 124
CoAPTreeHandler 126
Broker CoAP 139
RemoteResource 154
Valutazioni sperimentali 158
Indicatori di performance 158
Caso d'esempio 159
Configurazione di prova 160
Test su MQTT 162
Test sulla Resource Directory 166
JDK 8 170
Ottimizzazione degli intervalli di validità delle risorse su RD 180
Test e ottimizzazioni delle operazioni di lookup su RD 185
Considerazioni finali e sviluppi futuri 188
Conclusioni 192
Bibliografia 194
Ringraziamenti 195

1. Introduzione

Oggigiorno sono tanti i dispositivi connessi a Internet che possono scambiarsi le proprie informazioni tra di loro.
In futuro saranno ancora di più e per poter garantire loro una corretta connessione bisognerà implementare protocolli e architetture adatte a supportarli perchè se si vuole avere un'espansione completa bisognerà rispettare determinati requisiti che, nella maggior parte delle volte, vedono come protagonista principale la loro limitata capacità di elaborazione.
Le città, d'altro canto, stanno assumendo caratteri sempre più tecnologici. È impensabile prevedere che in futuro ogni singola città sarà sprovvista di sistemi che permettono di capire la reale situazione in cui imperversa.
È anche naturale pensare che il progresso tecnologico avanzi di pari passo alle necessità delle persone ed è fondamentale capire come i due concetti possano fondersi per dare vita a una vera e propria rivoluzione.
Potenzialmente saremo capaci di avere qualsiasi informazione a portata di click (o qualsiasi UI interaction sarà concepita in futuro per permettere l'interfacciamento con gli oggetti connessi) e saremo capaci di gestire la nostra vita grazie a questi dati.
Come qualsiasi grande rivoluzione, all'inizio, ci saranno dei costi alti da pagare.
Qualsiasi azienda, messa di fronte ad una possibilità, ne approfitterà per crearci il proprio business, spremendo al massimo i potenziali acquirenti. Prendendo in considerazione i recenti fatti di cronaca relativi alla privacy, con tutte le possibili conseguenze e il coinvolgimento di grandi aziende, viene naturale pensare che bisognerà rendere quanto più possibile decentralizzato il sistema di gestione delle risorse.
In questo modo sarà più difficile per le organizzazioni tracciare il comportamento degli utenti, perchè è probabile che con l'aumentare delle informazioni disponibili ci saranno più possibilità di analizzare i comportamenti quotidiani degli abitanti.
Evitando di affidarsi completamente ad un'unica azienda e magari avendo a disposizione software open-source, si riuscirebbe in parte ad eliminare il problema di rendere pubblica in maniera eccessiva la propria vita privata.
Recentemente, stanno nascendo tantissimi progetti open su piattaforme di condivisione come github, che permettono di sfruttare oggetti che utilizziamo comunemente ogni giorno rendendoli "smart". Inoltre, permettono a questi oggetti di comunicare tra di loro o di gestire

il loro ciclo di vita.
Uno di questi si chiama Kura ed è un framework per l'Internet delle Cose che è stato inizialmente sviluppato all'interno di Eurotech e poi ne è stato rilasciato il codice sorgente.
È perfettamente compatibile con tutte le nuove tecnologie, frutto della ricerca di diverse università.
Due di questi protocolli sono CoAP ed MQTT, progettati per essere veloci, flessibili e sicuri grazie all'integrazione con SSL e DTLS.
Lo scopo della tesi è quello di utilizzare la piattaforma Kura ed estenderla implementando un supporto CoAP che permetta facilmente di aggiungere un qualsiasi tipo di oggetto ed esporlo esternamente come risorsa su cui si possono applicare metodi di tipo REST.
Sarà configurabile in maniera approfondita, grazie all'utilizzo dei metatype presenti all'interno di Kura per la Web UI. Si avrà la possibilità di impostare il livello del dispositivo all'interno della gerarchia dei nodi e anche la capacità di attivare e disattivare dinamicamente il supporto ai protocolli di sicurezza.
Inoltre il supporto dovrà essere capace di far fronte a eventuali failure nelle connessioni dei dispositivi, notificando la nuova situazione ai nodi interessati.

2. Internet of Things: Requisiti e Scenari Applicativi

2.1. IoT

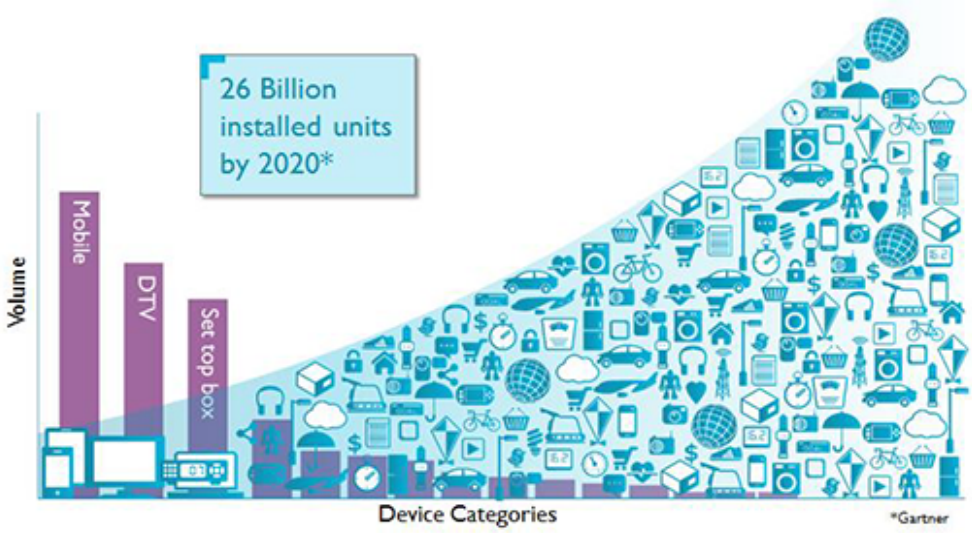
La Internet of Things consiste nella interconnessione di dispositivi di basso livello che permette di ottenere reti con un alto numero di nodi che possono scambiarsi informazioni e trasmetterle al mondo esterno.

Per ottenere queste funzionalità abbiamo bisogno di protocolli standard.
Due dei più promettenti per l'utilizzo su dispositivi a capacità limitata sono MQTT e CoAP.
Entrambi:

- Sono standard open.
- Sono stati progettati appositamente per ambienti vincolati dal punto di vista delle capacità dei dispositivi e della rete (non come HTTP).
- Forniscono meccanismi per la comunicazione asincrona.
- Usano lo stack TCP/IP.
- Hanno un ampio range di implementazioni.
- MQTT inoltre fornisce flessibilità nella comunicazione con l'approccio publish-subscribe, mentre CoAP è stato progettato per avere interoperabilità massima con il Web.

<u>Scenario attuale</u>

IoT copre diverse categorie di prodotto: oltre ai sensori presenti all'interno delle case, abbiamo anche smart grid, automobili connesse ad Internet e più in generale dispositivi di piccole dimensioni che possono essere presenti praticamente dappertutto.



Ci sono circa 7 miliardi di dispositivi connessi al giorno d'oggi, secondo una stima di Gartner, e nel 2020 ce ne saranno addirittura 26 miliardi.
Stiamo parlando di una fetta di mercato potenzialmente notevole su cui le aziende possono investire per aumentare il loro fatturato.

Prima di passare alla descrizione degli obiettivi e dei protocolli bisogna comunque fare delle riflessioni sui vari aspetti chiave legati ad IoT:

1. Gestione delle connessioni
Gli objects connessi a Internet devono effettuare sempre delle azioni di creazione di connessioni. Sono disponibili diversi pattern di messaggistica per ottenere questo comportamento (publish-subscribe, request only o request callback) e quindi è necessario riuscire ad inserire uno strato di sicurezza che permetta una comunicazione affidabile senza intrusioni esterne.

2. Proliferazione dei protocolli
Oltre ai due citati in questo documento (CoAP e MQTT) sono disponibili anche AMQP e XMPP (insieme a molti altri). Il modo giusto di agire per evitare un'espansione troppo elevata del loro numero è quello di utilizzare tecnologie già esistenti adattandole ai nuovi requisiti.

3. Chiavi di accesso
Mettere al sicuro le chiavi è fondamentale per la sicurezza dei dispositivi che altrimenti sarebbero vulnerabili ad attacchi esterni. Un possibile punto debole è la memorizzazione sui

server che introduce molteplici difficoltà per aspetti come autenticazione, integrità e confidenzialità.

4. Naming

L'identificazione degli utenti è ragionevolmente accurata con l'utilizzo di Active Directory, LDAP e in genere database specifici utilizzati nelle applicazioni.
L'equivalente, invece, per ogni tipo di oggetto della IoT non esiste. Questo aspetto è molto importante per quanto riguarda l'attribuzione delle policy di accesso ai vari nodi e alle risorse in essi contenute.

5. Dispositivi a capacità limitata

Le operazioni di deploy sui dispositivi molte volte sono limitate dalla capacità di elaborazione disponibile. I progettisti dei sistemi di IoT devono tenere conto di questa realtà e devono basare le loro assunzioni sul fatto che non è possibile avere le stesse capacità di storage o di banda che si possono trovare nelle normali reti ad alto livello.
Il pattern corretto sarebbe sempre quello di effettuare le operazioni più costose all'esterno lasciando al singolo dispositivo quelle più semplici da eseguire.

6. Servizi a tempo

La maggior parte dei protocolli di sicurezza si basa sull'utilizzo di eventi temporali ben precisi. Dato che stiamo comunque lavorando con dei dispositivi molto limitati anche dal punto di vista della memoria non possiamo aspettarci di riuscire ad utilizzare con efficacia gli stessi meccanismi. La soluzione ideale sarebbe quella di affidarsi non ad istanti temporali ben precisi ma utilizzare ad esempio dei contatori per identificare gli eventi.

7. Gateway

L'utilizzo di gateway è decisivo sia per quanto riguarda aspetti di sicurezza ma anche per la risoluzione di bug che, purtroppo, sono sempre presenti anche nei sistemi progettati nel miglior modo possibile. La presenza di un intermediario permette di agire direttamente su di esso e non sui dispositivi, diminuendo così il tempo necessario per risolvere i problemi e il deployment di nuove patch.

8. Modalità di gestione dei fallimenti

È molto importante conoscere lo stato in cui si trova un dispositivo o più in generale un

sistema quando ci sono dei fallimenti nel suo funzionamento normale.
Il pattern corretto da seguire sarebbe quello di cercare di anticipare la situazione di failure (quindi agendo in modo proattivo) così da trovare nel minor tempo possibile una soluzione.

2.2. Obiettivi

Riuscire a sfruttare le tecnologie presenti in IoT in modo sinergico permette di ottenere tantissime funzionalità con requisiti di elaborazione molto bassi dato che sono protocolli che sono stati progettati appositamente per ambienti con dispositivi a bassa capacità.
In particolare si vuole sviluppare un supporto scalabile per CoAP (protocollo client-server molto leggero) per Kura (framework open-source per IoT):

1. Che permetta di ricevere informazioni di dominio pubblico e privato in maniera veloce, economica e distribuita.
2. Flessibile: sfrutta la capacità del Raspberry Pi di interfacciarsi con tantissimi tipi di sensori a basso costo.
3. Semplice da configurare: dopo aver configurato il service basta sviluppare una semplice classe Java con l'override dei metodi REST.
4. Che utilizzi tecnologie open-source: dato che è basato su Kura il cui codice sorgente è disponibile su github.
6. Che gestisca automaticamente la gerarchia di nodi: compresa la rilevazione dei dispositivi che si disconnettono in modo imprevisto e conseguente riconfigurazione dell'albero.
7. Che permetta l'accesso ai dati da remoto: basta conoscere l'URL di un nodo Kura e si possono effettuare operazioni di:
    - Query: ad esempio si possono inserire i seguenti URI per le richieste

      coap://trasporti.it/bologna/rd-remote?linea=11
      coap://bologna.it/parchi/rd-remote?nome=giardini_margherita&sensore=temp

    - Ricerca: in questo caso invece una URI tipo sarebbe la seguente

      coap://trasporti.it/bologna/rd?linea=11

Più in generale è possibile sviluppare un'app che sfrutti librerie CoAP per le varie piattaforme per realizzare automaticamente queste chiamate.

2.3. Requisiti

Di seguito vengono indicati i requisiti principali di un'applicazione che utilizza il supporto CoAP per Kura:

- Accesso a informazioni di tipo pubblico e privato (ovviamente con l'utilizzo di DTLS)
- Si vuole ottenere una modalità di accesso standard a tutte i dati forniti dai sensori.
- I nodi su cui è installato Kura possono avere disconnessioni abbastanza frequenti.
- Le richieste di informazioni sui sensori possono essere numerose: dato che viene utilizzato CoAP, otteniamo prestazioni di gran lunga superiori rispetto ad HTTP per richieste REST.
- È possibile utilizzare i duplicati per operazioni di load balancing: quando un determinato nodo è sotto carico elevato possiamo programmare la restituzione dal broker di gestione centrale di un altro nodo su cui fare le richieste.

2.4. Ambito Smart City

Con il termine *Smart City* si intende caratterizzare una città che utilizza la tecnologia con lo scopo di migliorare la vita degli abitanti.
Riuscire ad avere informazioni in tempo reale sullo stato degli edifici oppure sui consumi di determinate aree, permette una gestione efficiente della città, con conseguente riduzione dei costi ed ottimizzazione delle attività quotidiane. Una di queste tecnologie consiste nell'utilizzare reti di sensori. Possiamo in questo modo andare a misurare molti parametri e in base al loro valore, prendere determinate decisioni.
I dati raccolti possono essere disponibili pubblicamente in modo tale da rendere partecipi anche i cittadini dello stato attuale di zone o edifici. Possono utilizzare queste informazioni per regolare la propria vita quotidiana e scegliere sempre la mossa migliore per ottimizzare i propri tempi.
Ad esempio se una persona deve fare jogging in un parco pubblico, necessita di sapere le condizioni atmosferiche in tempo reale (temperatura, umidità, …) per potersi adeguare e prepararsi nel miglior modo possibile all'attività fisica.
Di seguito sono elencati i vari possibili utilizzi delle WSN.

Un primo utilizzo potrebbe essere all'interno dei parchi pubblici.

Questo ambito di applicazione si sposa benissimo con tutti quei cittadini che svolgono regolarmente attività fisica.
È infatti utilissimo avere a disposizione informazioni (ad esempio se si vuole fare running in un parco) del:

- Livello di umidità
- Temperatura
- Livello di raggi UV
- Livello di rumore
- Numero di persone attualmente presenti

Infatti a seconda dei valori che assumono questi indicatori, è possibile scegliere:

- L'abbigliamento appropriato.
- L'intensità dell'allenamento che si vuole effettuare.
- Se scegliere un parco rispetto ad un altro perché è più tranquillo e ci si può muovere più liberamente.

Un servizio simile a quello dei parchi pubblici si può applicare anche alle stazioni balneari.

Possiamo inserire, infatti, in ogni stazione una serie di sensori per il calcolo del:

- Livello medio di rumore.
- Temperatura.
- Numero di posti disponibili.
- Indice UV.

Un cittadino prima di recarsi al mare può tranquillamente controllare questi valori e decidere dove andare in base ai propri gusti:

- Se c'è musica (o bambini che urlano) in un determinato lido e magari ci si vuole sdraiare in un ambiente tranquillo.
- Com'è l'indice UV in modo tale da uscire solo in determinati orari.
- Se ci sono troppe persone o addirittura se non ci sono ombrelloni disponibili.

Simile al caso precedente, vogliamo controllare le condizioni all'interno di una biblioteca.

Si potrebbe pensare di avere diversi sensori per misurare:

- Livello di rumore presente.
- Numero di persone.
- Temperatura.

Avendo a disposizione queste informazioni si può scegliere di svolgere altri compiti evitando di andare a controllare di persona lo stato attuale.

Un'altra potenziale applicazione, in ambito smart city, è quella relativa alla gestione delle code negli edifici pubblici.

Possiamo esporre come informazioni:

- Differenze tra ultimo numero e numero attuale (per comprendere il flusso medio di elaborazione delle operazioni).
- Numero attuale di persone.

Sono dati che possono essere di interesse per:

- Poste.
- Biglietterie.
- Banche.
- Mense.

Avendo a disposizione gli indicatori elencati in precedenza, si ridurrebbero di molto i disagi provocati dalle attese, ogni volta che dobbiamo recarci in uno di questi edifici.

Un altro ambito di utilità è quello dei ristoranti.

Ogni ristorante, infatti, potrebbe avere dei sensori che indicano:

- Numero attuale di persone presenti all'interno.
- Livello medio di rumore.
- Flusso medio di persone che entrano ed escono (questa informazione può essere utilizzata per capire in media fra quanto tempo ci saranno tavoli liberi).
- Temperatura.

Dal punto di vista del comune, invece, potrebbe essere interessante installare sensori presenti in punti strategici della città.

Si potrebbe calcolare:

- Livello di inquinamento acustico in dB (se viene rilevato un livello alto può significare che sono avvenuti incidenti o magari ci sono ingorghi in determinati punti).
- Livello di inquinanti (inquinamento atmosferico).
- Livello di inquinamento elettromagnetico.
- Livello di vibrazioni.

Tutte queste informazioni che si integrano benissimo con l'obiettivo di rendere efficienti i servizi erogati all'interno della città in base ai valori assunti dai sensori.

Consideriamo, ora, il caso in cui vogliamo automatizzare un edificio privato in maniera economica e veloce.

Abbiamo a disposizione una serie di sensori che vogliamo interfacciare con un sistema gerarchico di rilevazione dei valori e impostazione di eventuali attuatori:

- Sono presenti sensori per la misurazione del:
  - Livello di rumore.
  - Livello di inquinamento dell'aria.
  - Livello di consumo dei dispositivi all'interno di un ufficio.
- Sono presenti attuatori per l'accensione e lo spegnimento (o più in genere il controllo remoto) di:
  - Luci.
  - Ventole.
  - Apparecchi elettronici.
  - Qualsiasi dispositivo che è possibile interfacciare anche con switch connessi in WiFi.
  - Dispositivi a bassissimo consumo che utilizzano CoAP come protocollo di comunicazione.

Questi apparecchi possono essere visti come risorse che possono inviare e ricevere informazioni attraverso operazioni REST.
Quello che si vuole ottenere è il controllo totale di varie zone dell'edifico connesse gerarchicamente attraverso dispositivi a basso consumo e a ridotte capacità computazionali come il Raspberry Pi.
A loro volta sono presenti diversi uffici distanti parecchi km tra di loro e dato che possono presentarsi failure nelle connessioni, bisogna essere capaci di gestire queste situazioni.

2.5. Progetti Smart City esistenti

Alcuni tra i progetti Smart City esistenti che meritano una menzione particolare, anche perché si sono posizionati tra i primi posti nell'IoT Awards 2015 (competizione a cui possono partecipare i vari progetti che hanno come scopo quello di rendere la vita di tutti migliore introducendo la tecnologia in posti che non erano stati considerati finora), sono:

Array of Things

È un progetto che è già presente all'interno della città di Chicago e consiste in una serie di box installati in diversi punti strategici che permettono di memorizzare dati in real-time sull'ambiente cittadino e sulle infrastrutture per attività di ricerca e anche di utilizzo pubblico. All'interno di una box sono presenti sensori di temperatura, umidità, luce, monossido di carbonio, diossido di nitrogeno e vibrazioni. Sono presenti inoltre altri tipo di sensori ambientali ed è probabile che in futuro siano aggiunte anche le misurazioni sulle precipitazioni e sul vento.
I dati vengono pubblicati minuto per minuto su un database esterno e sono gestiti direttamente dalle autorità di Chicago che assicurano che non verranno utilizzati per arricchire le casse della città ma per un utilizzo prettamente pubblico.

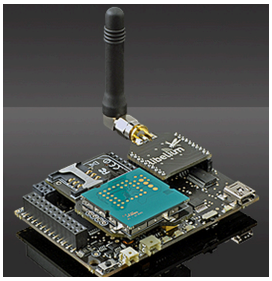
BigBelly

Si tratta di un progetto che permette di rendere smart tutti i cassonetti di rifiuti presenti in una città. È una soluzione che permette la gestione remota delle informazioni sull'attuale situazione interna di ogni stazione di riciclaggio per favorire la corretta ed efficiente rimozione successiva da parte degli addetti.
Infatti si stima che grazie alle ottimizzazioni apportate dall'aggiunta di stazioni BigBelly è possibile ridurre i costi dell'80%, evitando, tramite l'accesso sicuro al cassonetto, di rendere visibili i rifiuti e contaminare con odori sgradevoli la zona.

Kiunsys

Un aspetto che non è stato considerato finora è quello dei parcheggi. Attraverso l'utilizzo di una serie di sensori e di access gates è possibile conoscere in tempo reale la presenza di posti auto disponibili e addirittura sapere di quanto sarà il costo associato.
Attraverso l'analisi tramite le videocamere già installate in moltissimi parcheggi sarà possibile riconoscere il numero di targa e quindi fornire informazioni utilizzabili per rintracciare magari un individuo che ha commesso un'infrazione.
La soluzione cloud utilizzata internamente permette di ottenere un quadro completo della situazione attuale e potrebbe anche essere utilizzata dalle autorità per capire se i parcheggi sono stati pagati, evitando così controlli inutili.

Waspmote



È un dispositivo utilizzabile di dimensioni molto ridotte che può essere utilizzato per la connessione di un numero elevatissimo di sensori e tra le caratteristiche principali ci sono:

- Bassissimo consumo di energia: circa 0.7 uA.
- Circa 70 sensori disponibili.
- 15 tecnologie radio supportate tra cui 3G. ZigBee, 802.15.4, WiFi, RFID, NFC e Bluetooth 4.0. È supportata anche una nuova tecnologia chiama LoRA (868/915 Mhz) che permette una comunicazione a lungo raggio (fino a 20-30 km), anche attraverso gli edifici.
- Programmazione e configurazione remota: OTA.
- Supporto a librerie per la sicurezza: AES e RSA.

Due degli utilizzi più importanti che è possibile ottenere tramite questo dispositivo sono:

l'integrazione delle informazioni ricavate dai sensori con invio su Google Maps e il monitoraggio della qualità dell'acqua in fiumi, laghi e mari.

Strawberry Tree

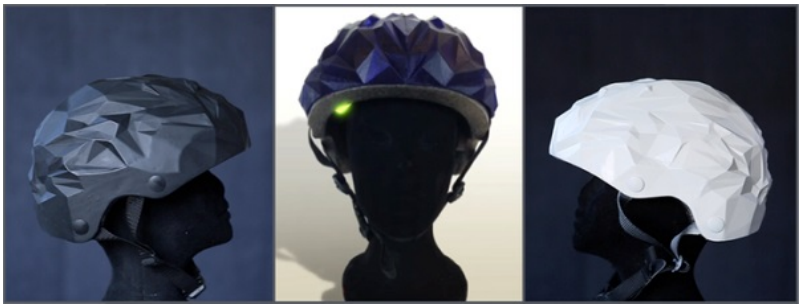
È un dispositivo urbano per la raccolta dell'energia solare che permette, utilizzandola, di ricaricare dispositivi elettronici connessi ad essa, di fornire connettività Internet WiFi, informazioni sullo stato ambientale circostante e su sensori presenti in diverse parti della città (dove è presente "l'albero").
È capace di funzionare fino a 20 giorni senza energia solare e contiene al proprio interno un sistema operativo che permette in modo semplice l'aggiunta di funzionalità successivamente.
In maniera simile all'Array of Things permette di inviare tutte le informazioni ad una piattaforma Cloud utilizzando il proprio WiFi interno, rendendo disponibili questi dati ai cittadini che utilizzano l'app associata.
È disponibile al momento in diverse città della Serbia.

Oxford Flood Network

Si tratta di un progetto presente nella città di Oxford che permette il rilevamento delle piene in città che hanno fiumi nelle vicinanze, basando le proprie previsioni sui livelli dell'acqua attuali, le falde sotterranee e conoscenze locali, perlopiù provenienti da eventi passati.

L'obiettivo degli sviluppatori era quello di creare un progetto che permettesse l'utilizzo di sensori a basso costo con l'utilizzo di una piattaforma che, combinando i dati provenienti da diverse sorgenti, permettesse di controllare e prevenire i danni provocati dalle piene dei fiumi.

MindRider Helmet



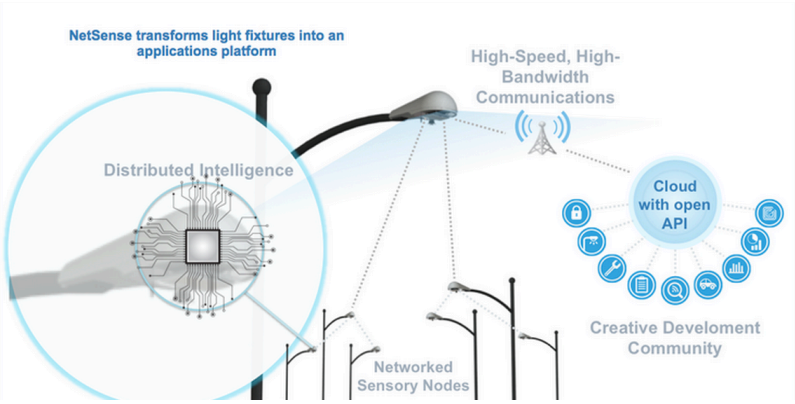
È un progetto sviluppato originariamente al MIT e consiste in un elmetto che permette di tracciare in tempo reale il modo in cui un utente utilizza la propria bicicletta, conoscere i propri movimenti e i posti che attraversa durante il tragitto.
Permette quindi a tutti i ciclisti di avere informazioni dettagliate sui posti migliori (definiti come SweetSpot all'interno della piattaforma) dove riposarsi e ammirare il panorama e dove invece sono presenti degli ingorghi oppure zone non ciclabili (definiti come HotSpot) e quindi ci sono dei peggioramenti importanti nella propria esperienza ciclistica.
È stato effettuato un mapping della città di Manhattan attraverso MindRider ed è possibile consultarne i risultati online per capire come ottimizzare il proprio allenamento in base alla posizione geografica.

Placemeter

È un progetto del 2013 utilizzato nella città di New York e permette di estrarre informazioni

da live feed video come il numero di persone che camminano su una determinata strada, che entrano ed escono dai negozi e il numero di auto presenti, in modo tale da poter capire in tempo reale il traffico presente.
Lo scopo è quello di analizzare le abitudini delle persone e ottimizzare il transito urbano in determinate aree coperte da videocamere (presenti ormai dappertutto a New York). Un altro importante utilizzo è quello della prevenzione dei furti, mantenendo così alto il livello di sicurezza della città.
Per quanto riguarda la privacy, non vengono rilevati dati che potrebbero essere utilizzati per il riconoscimento delle persone dato che nell'algoritmo (proprietario) si cerca di conteggiare solo il numero di individui utilizzando la silhouette tipica di un essere umano.

Sensity



Un'altra applicazione in ambito Smart City riguarda l'utilizzo dell'illuminazione pubblica per fornire diversi servizi alla città. È possibile andare ad agire sui pali della luce, rendendoli smart, e introducendo al loro interno una serie di sensori che comunicano tra di loro per fornire un quadro generale del traffico, della presenza di eventuali guasti sulla rete stradale, rilevamento di attività sospette e di infrazioni e analizzare la situazione ambientale attraverso la presenza di sensori integrati all'interno del palo.

Tzoa

Una categoria di device che si sta diffondendo in maniera capillare è quella dei wearable (cioè dei dispositivi smart indossabili). Interessante è capire come questo tipo di oggetti possa essere utilizzato in ambito Smart City.
Tzoa è un dispositivo indossabile composto da una serie di sensori interni che permettono di misurare la qualità dell'aria, la temperatura, l'umidità, la pressione atmosferica e il livello di raggi UV quando si è esposti al sole. Utilizzando il proprio smartphone è possibile accedere a queste informazioni e ricevere delle raccomandazioni sulle azioni da prendere nei diversi casi. Ad esempio si può notificare di aprire la finestra per far circolare l'aria oppure scegliere le direzioni da seguire con un livello di inquinamento basso quando si fa jogging e la quantità di luce solare a cui si è sottoposti soprattutto durante i periodi estivi.
Si può inoltre costruire una mappa della zona con tutti i dati ricevuti dai vari dispositivi e fornire comunque a chi non ha Tzoa la situazione corrente nella city.

Enevo

In maniera molto simile alla soluzione BigBelly, Enevo permette il monitoraggio dei cassonetti dei rifiuti portando ad un risparmio di circa il 50% nelle aziende che li gestiscono. Funziona con qualsiasi tipo di cassonetto (vetro, bio, metalli, plastica, …) e una volta installato è possibile controllare direttamente online lo stato di tutti gli Enevo presenti in città tramite una comoda interfaccia Web. I servizi Web permettono di notificare ad esempio se ci sono degli eventi inaspettati come un repentino cambio di temperatura su un determinato dispositivo.

Il funzionamento è semplice: i sensori (alimentati a batteria) effettuano il monitoraggio del livello di riempimento e i dati ricavati vengono inviati via rete cellulare ai server Enevo per le analisi ed è possibile visualizzarne i risultati attraverso il sito web. Ad esempio si possono anche effettuare previsioni sull'istante di riempimento totale e utilizzare strumenti di pianificazione per creare mappe di raccolta, utilizzate dagli autisti dei camion per il raccoglimento dei rifiuti.

Soofa

In maniera molto simile allo Strawberry Tree, Soofa permette di avere a disposizione un dispositivo che permette di sfruttare l'energia solare per ricaricare i vari dispositivi elettronici in commercio come smartphone o smartwatch.
L'unica differenza è che invece di assumere le sembianze di un albero altro non è che una panchina "smart".
Permette inoltre di condividere anche le informazioni sull'ambiente circostante attraverso sensori che misurano la qualità dell'aria.

3. Tecnologie utilizzate

3.1. CoAP

CoAP è l'acronimo di *Co*nstrained *A*pplication *P*rotocol ed è un protocollo ottimizzato per la comunicazione tra dispositivi con capacità limitate.
È stato sviluppato dal gruppo CoRE (*Co*nstrained *R*esource *E*nvironments) IETF.

Il modello di interazione è molto simile a quello di HTTP (client-server). La differenza principale sta nel fatto che nella comunicazione M2M i nodi possono essere sia client che server e quindi non è specificato univocamente il ruolo delle entità.
Le caratteristiche principali di CoAP sono:

- Protocollo web che rispetta i requisiti di M2M per gli ambienti limitati sia dal punto di vista delle capacità elaborative dei nodi sia per reti non ad alta velocità.
- Aggiunta del supporto di UDP a richieste unicast e multicast con affidabilità opzionale.
- Scambio di messaggi asincrono.
- Overhead sull'header basso e complessità delle operazioni di parsing molto bassa.
- Supporto ad URI e Content-Type.
- Capacità di proxy e caching implementate in modo semplice.

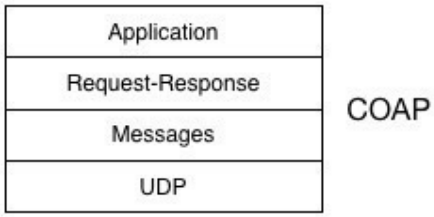
3.1.1. Introduzione e Architettura

Come HTTP, CoAP è un protocollo per il trasferimento di documenti.
I pacchetti CoAP sono molto più piccoli di quelli che fanno parte dei flussi TCP di HTTP. Il mapping da stringhe ad interi è utilizzato in modo molto esteso per risparmiare spazio. Sono semplici da generare e ne può essere fatto il parsing senza consumare RAM extra.

CoAP gira su UDP e non su TCP (i client e i server comunicano attraverso datagrammi,

quindi connectionless). Ritrasmissioni e ordinamento dei pacchetti sono implementati nel livello applicazione. La rimozione del requisito di utilizzare TCP può infatti permettere la realizzazione di reti IP in piccoli microcontroller.
Inoltre permette l'indirizzamento sia broadcast che multicast.
Segue un modello client-server dove i client effettuano richieste verso i server e i server creano e inviano risposte all'indietro verso il mittente. I client hanno la possibilità di utilizzare le primitive GET, PUT, POST e DELETE sulle risorse. È possibile operare con HTTP e oggetti RESTful attraverso dei semplici *proxy*.

Dato che CoAP è basato sull'utilizzo di datagrammi può essere utilizzato anche come cross-protocol per l'aggancio ad SMS e ad altri protocolli (basati sempre su UDP).
Viene utilizzato un approccio a due livelli in cui abbiamo:



1. Livello di messaging : si interfaccia con il livello di trasporto UDP
2. Livello con interazioni request-response : utilizzano Method e Response Codes.

Questi due livelli convergono in un unico protocollo con messaging e richiesta-risposta inseriti nell'header CoAP.

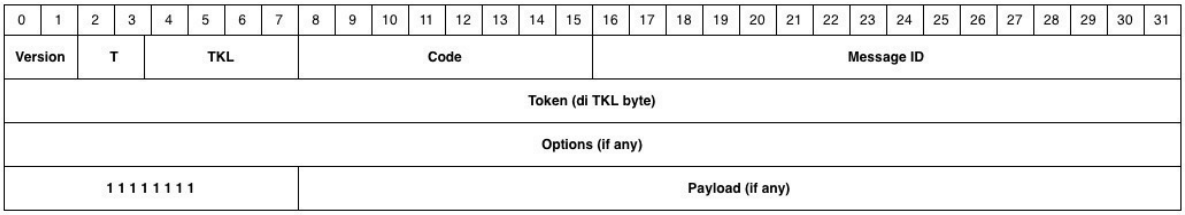
3.1.2. Messaggi

Come già detto CoAP è basato sullo scambio di messaggi compatti che, di default, sono trasmessi sopra UDP (ogni messaggio CoAP occupa la sezione dati di un datagramma UDP). Può essere anche usato sopra DTLS (Datagram Transport Layer Security) per ottenere maggiore sicurezza, sopra TCP oppure SCTP.
I messaggi sono codificati in un formato binario semplice.

3.1.2.1 Formato dei messaggi

È composto da un header iniziale di 4 byte a cui poi segue un Token che ha una lunghezza

variabile (tra 0 e 8 byte).



Questi sono i campi:

- **Campi IP**
  - *Source Address* : indirizzo unicast della sorgente
  - *Destination Address* : indirizzo unicast della destinazione ma può anche essere usato un multicast in UDP
- **Campi dell'header**
  - *Version* (intero a 2 bit) : indica il numero di versione CoAP. Il valore deve essere uguale a uno. Altri valori sono riservati per versioni future.
  - *T* (intero a 2 bit) : indica il tipo di messaggio che può essere Confirmable (0), Non-confirmable (1), Acknowledgement (2) o Reset (3).
  - *TKL* (Token length, intero a 4 bit) : indica la lunghezza del campo token (da 0 a 8 B).
  - *Code* (intero a 8 bit) : è composto da una classe da 3 bit e poi 5 bit vengono utilizzati per il dettaglio. Una classe può essere: una richiesta, una risposta con successo, una risposta con errore da parte del client e una risposta con errore da parte del server. Per quanto riguarda il dettaglio invece è presente una tabella con tutte le specifiche.
  - **Message ID** (intero a 16 bit) : Viene utilizzato per rilevare la duplicazione dei messaggi ACK o RST rispetto ai messaggi CON e NON. Lo stesso valore non deve essere riutilizzato all'interno dello stesso EXCHANGE_LIFETIME. È comunque consigliato che il valore iniziale sia scelto in modo random in modo tale da rendere meno probabile un tipo di attacco off-path sul protocollo.

La tabella per quanto riguarda le specifiche sui messaggi è la seguente:

	CON	NON	ACK	RST

Request	si	si	no	no
Response	si	si	si	no
Empty	*	no	si	si

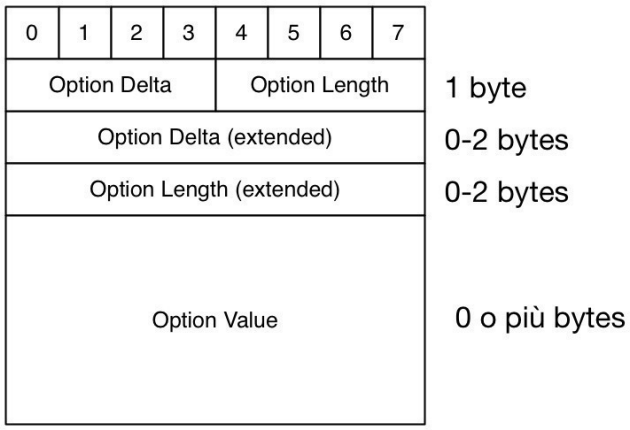
* --> la combinazione non è utilizzata di solito ma l'obiettivo è simulare un messaggio RST (operazione anche detta di CoAP ping).

3.1.2.2 Formato delle opzioni

CoAP definisce un numero elevato di opzioni che possono essere incluse in un messaggio.
Viene specificato:

1. Numero
2. Lunghezza del valore
3. Valore

Lo schema è il seguente:



Invece di specificare il numero dell'opzione direttamente, le istanze sono ordinate rispetto al loro numero di opzione e ad un valore delta di encoding.
Questo valore viene calcolato come la somma dei delta dell'istanza e il numero di opzione delle precedenti istanze nel messaggio (se è la prima il valore è 0).
I campi delle opzioni sono quindi:

- *Option Delta* (intero a 4 bit) : È un valore compreso tra 0 e 12. Viene utilizzato come differenza tra l'Option Number dell'opzione e quello della precedente. Ci sono tre valori riservati per costrutti speciali:
  - 13 : il byte iniziale è seguito da un altro byte che indica il delta -13.
  - 14 : il byte iniziale è seguito da altri 2 byte che indicano il delta -269.

  - 15 : riservato.
- *Option Length* (intero da 4 bit) : un valore tra 0 e 12 indica la lunghezza dell'Option Value in byte. Ci sono tre valori riservati per costrutti speciali:
  - 13 : 8 bit precedono l'Option Value e indicano una Option Length -13.
  - 14 : 16 bit precedono l'Option Value e indicano una Option Length -269.
  - 15 : riservato.
- *Value* : non è altro che una sequenza di byte di lunghezza Option Length. I formati principali sono:
  1. Empty: tutti zero.
  2. Opaque: sequenza opaca di bytes.
  3. Uint: intero non negativo rappresentato in ordine di byte di rete utilizzando il numero di byte dato dal campo Option Length.
  4. String: codificata in UTF-8.

3.1.2.3 Dimensione del messaggio

La specifica CoAP fornisce solo un limite superiore alla dimensione del messaggio.
Infatti dovrebbe rientrare all'interno di un singolo pacchetto IP per evitare la frammentazione e dato che viene inserito nel payload UDP deve rientrare ovviamente in un singolo datagramma IP. Se l'MTU non è conosciuto per la destinazione deve essere di 1280 bytes.

Un'importante considerazione da fare riguarda la frammentazione relativa a molte reti a banda limitata. Ad esempio in un pacchetto L2 di 6LowPAN siamo limitati a soli 127 bytes inclusi i vari overhead e quindi bisogna comunque trovare un modo per effettuare delle trasmissioni in modo efficace.
Una soluzione potrebbe essere quella della trasmissione a blocchi descritta in *"Bormann, C. and Z. Shelby, "Blockwise transfers in CoAP", October 2013"* e trattata successivamente nel paragrafo associato. Nel caso in cui un destinatario abbia un buffer molto ridotto e riceve un messaggio che non rientra in quello spazio di memoria, può rispondere con un messaggio di tipo *Request Entity Too Large*. Il mittente successivamente suddivide il pacchetto originale in blocchi della giusta dimensione e invia più messaggi, ognuno con un blocco nel payload.

3.1.2.3.1 Trasferimento a blocchi

Utilizzando UDP il payload di un messaggio CoAP può avere dimensione massima di 64 KiB dopo il quale avviene la frammentazione IP.
Un altro motivo per cui è stato introdotto il trasferimento blocchi è quello di evitare la frammentazione anche a livello adaptation in cui il cui pacchetto massimo (in 6LowPAN) è 60-80 byte. Per supportare l'invio di pacchetti di dimensione maggiore non si fa altro che aggiungere l'opzione Block ad ogni pacchetto.
Il server in questo modo può gestire ogni trasferimento di blocco separatamente senza il bisogno di instaurare una connessione o avere memoria dei blocchi trasferiti precedentemente. L'obiettivo è quello di trasferire la rappresentazione di risorse molto grandi senza creare conversazioni con stato ogni volta che vengono inviati messaggi GET.
Il trasferimento di ogni blocco è ACKed e quindi è presente la ritrasmissione individuale se richiesta.

3.1.2.3.1.1 Opzione Block1 e Block2

Questo è lo schema delle opzioni:

No.	C	U	N	R	Name	Fornat	Length	Default
23	C	U	-	-	Block2	uint	0-3 B	(no)
27	C	U	-	-	Block1	uint	0-3 B	(no)

Sia *Block1* che *Block2* possono essere presenti sia nel messaggio di richiesta che nel messaggio di risposta. *Block1* si riferisce al payload della richiesta mentre *Block2* a quello della risposta. Entrambe specificano le proprietà del payload relativamente all'intero messaggio che sarà trasferito.
Naturalmente l'implementazione di quest'opzione non è mandatory ma, quando è presente deve essere presa in considerazione ed elaborata.

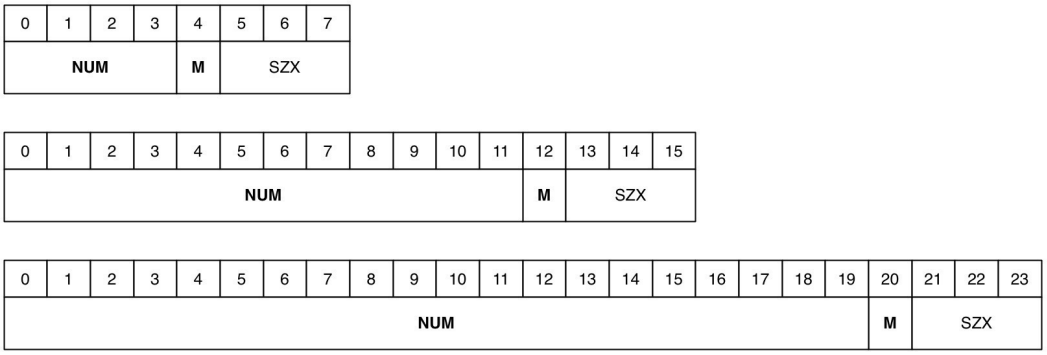
3.1.2.3.1.2 Struttura dell'opzione

Le tre informazioni principali per trasferire un blocco di dimensioni elevate sono:

1. La dimensione del blocco (*SZX*, SiZe eXponent).
2. Se ci sono blocchi successivi a quello corrente (*M*, More).
3. Il numero relativo del blocco all'interno della sequenza (*NUM*).

La dimensione dell'opzione è variabile (da 0 a 3 byte).

I valori sono indicati come nel seguente schema:



Per mantenere l'implementazione più semplice possibile viene supportato solo un piccolo intervallo di dimensioni uguali a potenze di 2 ($2^4 \ldots 2^{10}$).
Ci sono tre possibili utilizzi delle opzioni:

1. Utilizzo descrittivo : il Block2 viene utilizzato nella risposta e il Block1 nella richiesta.
2. Utilizzo di Block2 per il controllo in una richiesta : il campo *NUM* nell'opzione Block2 indica il numero di blocco che è stato richiesto per essere restituito nella risposta. In questo caso il bit M non ha una funzione e deve essere impostato a 0.
3. Utilizzo di Block1 per il controllo in una risposta : il campo *NUM* nell'opzione Block1 indica il numero di blocco che è stato ACKed. Il blocco *SZX* indica invece la dimensione del blocco più grande preferita dal server che successivamente il client dovrebbe adottare (può comunque utilizzare pacchetti di dimensione minore).

3.1.2.3.1.3 Block2

Quando una richiesta ottiene una risposta che include un'opzione Block2 con il bit *M* impostato ad uno, il mittente può ricevere blocchi addizionali della rappresentazione della risorsa inviando richieste successive con le stesse opzioni del messaggio originale con il Block2 che indica il numero di blocco e la dimensione desiderata.
Per influenzare la dimensione del blocco di una risposta il mittente può comunque inserire le proprietà nel Block2 nella richiesta iniziale. Tutte le risposte successive dovrebbero utilizzare questa dimensione oppure un valore minore.
Si può usare in congiunzione con l'opzione *ETag* per assicurarsi che i blocchi che sono stati riassemblati facciano parte della stessa versione della rappresentazione.
Se nel momento del riassemblaggio di un pacchetto abbiamo un payload con l'opzione *ETag*

diversa da quella richiesta, il client non fa altro che richiedere di nuovo il pacchetto mancante.

3.1.2.3.1.4 Block1

Per quanto riguarda invece l'opzione Block1 in risposta ad una richiesta con payload (PUT o POST) la dimensione del blocco indica la preferenza sulla dimensione dello stesso sul server. Il client quindi dovrebbe successivamente attenersi alla preferenza indicata nell'opzione.
Nel momento in cui il server va a ricomporre il pacchetto originale con tutti blocchi, se non sono disponibili tutti i payload necessari deve restituire uno messaggio di tipo *4.08 (Request Entity Incomplete)*. Un server comunque potrebbe anche restituire un errore *4.08* per ogni trasferimento che non si trova nella sequenza.
Non è comunque possibile per un singolo endpoint effettuare richieste di trasferimenti a blocchi multiple in modo concorrente per la stessa risorsa.

3.1.2.3.1.5 Utilizzo dell'opzione Block con l'opzione Observe

Le risorse osservate da un client possono essere tranquillamente trasferite in più messaggi CoAP. L'osservazione si applica sempre sull'intera risorsa e mai su un singolo blocco.
Quando si invia una notifica di tipo *2.05 (Content)* il server ha bisogno solo di mandare il primo blocco della rappresentazione e il client può ottenere i pacchetti successivi utilizzando messaggi GET con l'opzione Block2 che contiene valori di *NUM* più grandi di zero (aumentati progressivamente).

3.1.2.3.1.6 Codici di risposta

Sono stati aggiunti altri codici di risposta rispetto a quelli predefiniti di CoAP:

- *2.31 (Continue)* : indica che il trasferimento di questo blocco è stato eseguito con successo e che il server incoraggia l'invio di blocchi successivi.
- *4.08 (Request Entity Incomplete)* : indica che il server non ha ricevuto i blocchi di cui ha bisogno per procedere alla ricomposizione del messaggio originale.
- *4.13 (Request Entity Too Large)* : indica al client che il server non ha le risorse disponibili per memorizzare blocchi del pacchetto inviato dal client.

In molti casi quando trasferiamo una rappresentazione di una risorsa molto grande è

vantaggioso conoscere la dimensione totale all'inizio del processo.
Per questo ci vengono in aiuto altre due opzioni:

- *Size1* che viene utilizzata nei messaggi di richiesta
- *Size2* che viene invece utilizzata nei messaggi di risposta.

Oltre ad informare il server riguardo la dimensione di una determinata rappresentazione della risorsa le opzioni Size1 e Size2 non hanno altri scopi.
Questo è lo schema delle due opzioni:

No.	C	U	N	R	Name	Fornat	Length	Default
60			x		Size1	uint	0-4 B	(no)
28			x		Size2	uint	0-4 B	(no)

L'intermediario HTTP deve provare ad inviare il payload di tutti i blocchi ricevuti attraverso un trasferimento block-wise all'interno di un'unica richiesta HTTP. Se è disponibile un buffer di dimensioni utili per quel determinato pacchetto, il trasferimento inizia quando viene ricevuto l'ultimo blocco CoAP.

3.1.2.3.1.7 Considerazioni sulla sicurezza

Quando viene fornito accesso ai blocchi all'interno di una risorsa possiamo avere delle vulnerabilità. Infatti un attaccante potrebbe confondere un server inducendolo ad utilizzare una risorsa parzialmente aggiornata.Nella specifica si raccomanda di utilizzare un IDS in modo tale che abbia la conoscenza dell'intera rappresentazione della risorsa. Ci sono approcci che permettono di trasmettere blocchi pari su un percorso e blocchi dispari su un altro, e questo limita la visibilità del sistema di rilevazione delle intrusioni.

Un altro possibile attacco è il buffer overflow tramite indicazioni fuorvianti sulla dimensione di una rappresentazione.

Alcune richieste di trasferimento blocchi possono indurre il server a creare dello stato locale e tutti i meccanismi che utilizzano questo tipo di approccio creano opportunità per degli attacchi DoS. Dove possibile, quindi, i server dovrebbero minimizzare la creazione di stato per sorgenti non autorizzate, utilizzando degli approcci stateless.
Un'altra soluzione sarebbe quella di non accumulare stato dopo un'intervallo di tempo. I client dovrebbero quindi effettuare il trasferimento blocchi in modo tale da minimizzare la

possibilità di incorrere in un timeout.

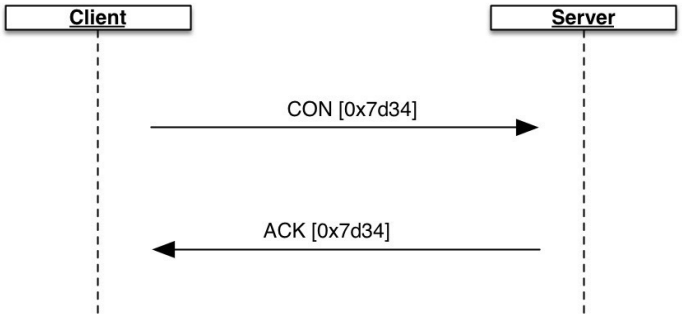
Un altro attacco (descritto nella sezione sicurezza di CoAP) è quello di amplificazione.
Un attaccante per viene limitato in questo tipo di exploit dal fatto che il server offre il trasferimento di rappresentazioni grandi di risorse solo in blocchi di dimensione piuttosto bassa.

3.1.2.4 Messaging model

Il Message ID viene utilizzato per rilevare i duplicati e per l'affidabilità (opzionale).

3.1.2.4.1 Comunicazione affidabile e non affidabile

L'affidabilità si ottiene marcando un messaggio come CON (CONfirmable). Il messaggio così composto viene ritrasmesso utilizzando un timeout di default e un backoff esponenziale tra le ritrasmissioni, fino a quando il destinatario non invia un messaggio ACK con lo stesso Message ID.
Esempio:



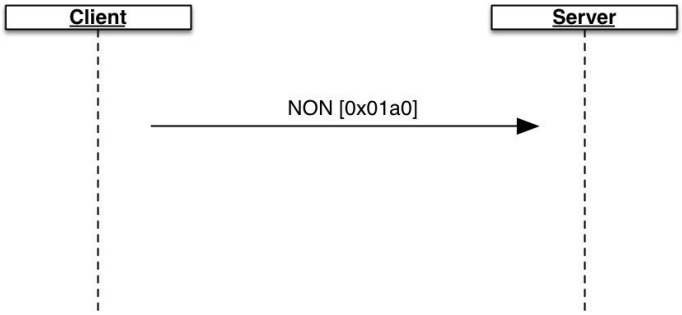
Quando invece il destinatario non è in grado di elaborare un messaggio CONfirmable risponde con un RST message (ReSeT).
Se il messaggio non è ACKed il client re-invia il messaggio CON con un intervallo che aumenta esponenzialmente fino a quando non riceve il messaggio ACK dal server.
Per ogni nuovo messaggio CON si deve tenere conto di:

1. Timeout iniziale che è impostato ad una durata random
2. Contatore di ritrasmissione (impostato a 0)

Ogni volta che c'è bisogno di ritrasmettere il contatore viene aumentato di 1 e il timeout viene

raddoppiato.

Nel caso in cui non abbiamo bisogno di una comunicazione affidabile (quando ad esempio dobbiamo inviare il risultato di una misura da un sensore) marchiamo il messaggio come NON (NON-confirmable-message).
Esempio:



Come nel caso precedente nel caso in cui il destinatario non è in grado di elaborare questo messaggio deve rispondere con messaggio RST.

3.1.2.4.2 Messaggi ed endpoint

Un endpoint è identificato in base alla security mode utilizzata. Nel caso non dobbiamo rendere sicura la comunicazione lo si identifica univocamente attraverso un indirizzo IP e un numero di porta UDP. In caso contrario identifichiamo l'endpoint secondo la modalità di sicurezza scelta.

3.1.2.4.3 Duplicazione dei messaggi

Un destinatario può ricevere lo stesso messaggio CON più volte all'interno di un *EXCHANGE_LIFETIME*. In questo caso dovrebbe:

- Fare l'ACK di ogni copia duplicata di messaggio CON con lo stesso messaggio ACK.
- Elaborare ogni richiesta o risposta solo una volta.

Si può rilassare questo requisito per 2 motivi:

1. Un server può decidere di rilassare il requisito di rispondere con lo stesso messaggio in modo tale da non mantenere stato per ogni Message ID. Infatti in questo caso potremmo elaborare separatamente ogni volta la richiesta nel caso questa operazione

sia meno costosa rispetto al mantenimento dello stato.

2. Il vincolo potrebbe essere rilassato anche nel caso in cui la semantica dell'applicazione stessa consenta di farlo.

3.1.2.4.4 Controllo della congestione

Il controllo base di congestione viene effettuato con il meccanismo di back-off esponenziale. La specifica suggerisce di mantenere il numero di interazioni con un dato server al massimo di un valore chiamato NSTART (di default 1). Per interazione si intende l'attesa di un ACK dopo aver inviato un messaggio CON.
L'algoritmo secondo il quale un client smette di attendere una risposta ad una richiesta non è definito nella specifica. L'endpoint dovrebbe comunque non eccedere il PROBING_RATE che è il data-rate medio nell'invio di un messaggio ad un altro endpoint che non risponde.
Una considerazione da fare in questo caso è che CoAP riversa l'onere di gestione della congestione per la maggior parte sui client. Essi comunque possono avere dei malfunzionamenti oppure essere sotto attacco e quindi è stata una scelta poco felice.
Una soluzione sarebbe quella di implementare lato server una limitazione del data-rate in modo tale da rispettare i requisiti dell'applicazione e alleggerire il carico di gestione sui client.

Tabella dei parametri di trasmissione dei messaggi

Nome	Valore predefinito
ACK_TIMEOUT	2 secondi
ACK_RANDOM_FACTOR	1.5
MAX_RETRANSMIT	4
NSTART	1
DEFAULT_LEISURE	5 secondi
PROBING_RATE	1 byte/secondo

La modifica di questi parametri è possibile ma:

- Non bisognerebbe diminuire l’ACK_TIMEOUT in modo significativo oppure aumentare il valore di NSTART senza utilizzare meccanismi che garantiscono la

safety del controllo di congestione.

- ACK_RANDOM_FACTOR non deve essere inferiore a 1.0 e dovrebbe essere abbastanza diverso da 1.0 per garantire protezione da effetti di sincronizzazione.
- MAX_RETRANSMIT può essere modificato liberamente ma un valore troppo piccolo riduce la probabilità di ricezione di un messaggio CON e un valore troppo alto invece introduce troppa attesa tra una trasmissione e l'altra.

La combinazione dei tre fattori definiti sopra influenza il timing della ritrasmissione che a sua volta influenza il tempo di storage di un'informazione.

I tempi derivati dai precedenti sono:

Nome	Valore predefinito
MAX_TRANSMIT_SPAN	45 secondi
MAX_TRANSMIT_WAIT	93 secondi
MAX_LATENCY	100 secondi
PROCESSING_DELAY	2 secondi
MAX_RTT	202 secondi
EXCHANGE_LIFETIME	247 secondi
NON_LIFETIME	145 secondi

- MAX_TRANSMIT_SPAN : tempo massimo dalla prima trasmissione di un messaggio CON alla sua ultima ritrasmissione.
- MAX_TRANSMIT_WAIT : tempo massimo dalla prima trasmissione di un messaggio CON al tempo in cui il mittente rilascia l'attesa di ricezione di un messaggio ACK o RST.

Di default vengono fatte delle assunzioni riguardo alle caratteristiche della rete e dei nodi:

- MAX_LATENCY : tempo massimo che un datagramma impiega dall'inizio della trasmissione al completamento della sua ricezione. È stato infatti calcolato in base al Maximum Segment Lifetime (definito nella RFC0793) ed è di 2 minuti. Definisce il worst-case scenario e permette anche ai timer per il tempo di vita dei Message ID di essere rappresentati solo da 8 bit.
- PROCESSING_DELAY : tempo che un nodo impiega per creare un messaggio ACK da un messaggio CON. Tenendo conto che l'invio di un ACK deve avvenire prima che ci sia il time out dal mittente allora viene impostato al valore ACK_TIMEOUT.

- MAX_RTT : round-trip time massimo

Altri valori derivati dai precedenti sono:

- EXCHANGE_LIFETIME : tempo dall'invio del messaggio CON al tempo in cui un ACK non è più atteso. Definito in questo modo:
  MAX_TRANSMIT_SPAN + (2 * MAX_LATENCY) + PROCESSING_DELAY
- NON_LIFETIME : tempo dall'invio di un messaggio NON al tempo in cui il Messaggio ID può essere riutilizzato in modo safe. Bisogna tenere conto che un mittente CoAP può inviare un messaggio NON molteplici volte in applicazioni multicast e quindi il riconoscimento del duplicato nel destinatario ha un ordine di tempo relativo a MAX_TRANSMIT_SPAN e quindi è più sicuro utilizzare questo valore:

  MAX_TRANSMIT_SPAN + MAX_LATENCY

3.1.2.4.5 Semantica di richiesta-risposta

Rispetto ad HTTP i messaggi non sono inviati su una connessione stabilita precedentemente ma sono scambiati in modo asincrono attraverso messaggi di tipo CoAP.

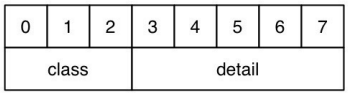
3.1.2.4.5.1 Richiesta

CoAP supporta i metodi base GET, POST, PUT e DELETE che possono essere facilmente mappati su HTTP.
Le proprietà rispetto ai metodi di HTTP sono le stesse e infatti sono:

1. Idempotenti (possono essere invocati più volte con lo stesso effetto, POST non è idempotente perché i suoi effetti dipendono dal server di origine e dipendono dalla risorsa target).
2. Safe (funzionano in modalità retrieval, come il metodo GET).

3.1.2.4.5.2 Risposta

Dopo aver ricevuto una richiesta il server risponde con un messaggio di risposta CoAP che fa match con la richiesta rispetto ad un token generato dal cliente.
Viene tenuto conto del Response Code che è un campo a 8 bit contenuto nell'header CoAP.



1. *Class* : rappresenta la classe di risposta che può essere di tre tipi
    - *Success* : la richiesta è stata ricevuta correttamente, capita e accettata
    - *Client error* : la richiesta contiene sintassi sbagliata
    - *Server error* : il server non è riuscito a creare un messaggio di risposta
2. *Detail* : aggiungono informazioni alla classe di risposta

La risposta può essere inviata in diversi modi

1. *Piggybacked* : può essere inserita direttamente nel messaggio ACK indipendentemente se è una risposta di successo o meno. La specifica non fornisce informazioni riguardo a se la risposta deve essere piggybacked o meno, ma viene consigliato che il server debba implementare questa modalità ovunque sia possibile in modo tale da minimizzare l'utilizzo della rete.
2. *Separata* : utilizzata quando la richiesta viene effettuata tramite un messaggio NON oppure quando il server ha bisogno di più tempo per ottenere la risposta e non può mandare indietro il messaggio ACK prima che il client non re-invii la richiesta. In questo caso viene reinviato prima il messaggio ACK come Empty message. Dal punto di vista implementativo bisogna dire che dato che il protocollo di trasporto non preserva la sequenzialità potrebbe arrivare prima il messaggio di risposta che quello di ACK. In questo caso bisogna considerare anche il messaggio di risposta come ACK.
3. *Non-confirmable* : potrebbe essere una risposta ad un messaggio CON e quindi un endpoint dovrebbe comunque essere preparato ad un messaggio del genere.

3.1.2.4.5.3 Token

La regola principale per la comunicazione è che quando due endpoint devono comunicare devono fare riferimento ad uno stesso token per identificare la "sessione".
Il client dovrebbe generarli in maniera tale che essi siano unici per una stessa coppia sorgente-destinazione. Quindi lo stesso valore può essere riutilizzato se abbiamo 2 destinazioni diverse.
Una considerazione va fatta per quanto riguarda la sicurezza. Se non utilizziamo TLS allora la specifica consiglia di scegliere in modo random e non banale il token per proteggersi dallo spoofing di risposte.
Se il client è connesso a Internet abbiamo un livello di protezione basso avendo a disposizione

solo 8 bit del Message ID. Bisogna quindi aumentare il livello di casualità andando a prendere in considerazione token che abbiamo un numero di bit molto più elevato (e inserirli magari nel payload in modo tale da gestirli a livello applicativo).

3.1.2.4.5.4 Regole per il matching richiesta-risposta

Per ottenere una corretta conversazione tramite CoAP:

1. L'endpoint sorgente della risposta deve essere lo stesso del destinatario della richiesta originale.
2. In una risposta piggybacked il Message ID della richiesta deve essere lo stesso del messaggio ACK e i token della richiesta e della risposta devono essere identici. Nel caso invece di risposte separate solo i token devono essere identici.

Un cliente che riceve una risposta ad un messaggio CON e ripulisce lo stato appena ha inviato l'ACK, perde lo stato della comunicazione nel caso l'ACK non arrivi al server (dato che non può re-inviarlo). In questo caso dovrebbe inviare un messaggio RST in modo tale che il server non gli re-invii più altre risposte.

3.1.2.4.6 Options

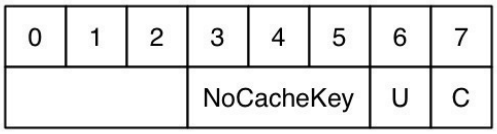
Le opzioni si dividono in due categorie:

1. *Elective* : alla ricezione un'opzione non riconosciuta deve essere ignorata.
2. *Critical* : alla ricezione di un'opzione non riconosciuta in una richiesta CON deve causare la restituzione di una risposta di tipo *Bad Response*, invece in risposta ad una CON oppure piggybacked in un ACK deve essere respinta. In un messaggio NON deve essere respinta.

Queste opzioni sono valide per endpoint che non utilizzano proxy. Nel caso di proxy invece le opzioni sono elaborate secondo classi *Unsafe* e *Safe-to-forward*.
Un'opzione è definita da un numero di opzione che fornisce anche informazioni riguardo la semantica della stessa:

- Numero dispari : opzione *Critical*
- Numero pari : opzione *Elective*

Lo schema (per i bit meno significativi) è questo:



- *C* (Class) : 0 : *Elective* e 1 : *Critical*
- *U* (Unsafe) : 0 : *Safe-to-forward e* 1 : *Unsafe-to-forward*
- *NoCacheKey* : Solo se i bit 3-4-5 sono posti a uno.

3.1.2.4.7 Payload

Il payload tipicamente è una rappresentazione della risorsa richiesta.
Il formato è specificato dall'opzione *Content-Format*. Nel caso non sia presente è l'applicazione stessa che deve fare guessing del formato (comportamento non consigliato dalla specifica). Se il payload consiste nella descrizione di un errore deve essere codificato in UTF-8 e deve essere quindi leggibile da un operatore umano.
Il client può decidere quale tipo di rappresentazione preferisce andando ad indicare l'opzione *Accept-Option*.

3.1.2.4.8 Caching

Gli endpoint CoAP possono effettuare caching delle risposte in modo tale da ridurre il tempo di risposta e il consumo di banda. Il caching non è altro che il riutilizzo di una risposta precedente per una nuova richiesta.
Sono presenti due meccanismi per garantire la freschezza dell'informazione e la validità.
Per una richiesta un endpoint CoAP non deve utilizzare una risposta in cache a meno che:

- Il metodo di richiesta e quello per ottenere la risposta memorizzata coincidono.
- Tutte le opzioni fanno match tra quelle nella richiesta al server e quelle della richiesta utilizzata per ottenere la risposta memorizzata. Non c'è bisogno che ci sia match tra qualsiasi tra le opzioni impostate come *NoCacheKey*.
- La risposta memorizzata è aggiornata o è stata validata in modo corretto.

La entry nella cache viene identificata da *Cache-Key*.
A differenza di HTTP la possibilità di inserire risposte CoAP nella cache non dipende dal metodo di richiesta ma dal Response Code.

Il meccanismo di aggiornamento si basa sul fatto che il server di origine fornisce un tempo di

expiration esplicito utilizzando l'opzione *Max-Age*. Il valore predefinito è di 60 secondi. Quindi una risposta memorizzata dopo 60 secondi non è più valida.

Nel modello di validazione quando un endpoint ha uno o più risposte per una richiesta GET ma non può usare nessuna di esse (perché non sono aggiornate), può essere utilizzata l'opzione *ETag* per dare al server di origine la possibilità di selezionare una risposta memorizzata e aggiornarla. L'endpoint aggiunge alla richiesta la entity-tag di ogni risposta memorizzata valida.
Una risposta che viene utilizzata per soddisfare una richiesta può rimpiazzare quella memorizzata.

3.1.2.4.9 CoAP URI

Gli *URI-Scheme* utilizzati da CoAP sono "coap" (per le normali richieste) e "coaps" (per le richieste coap su DTLS).
La sintassi per coap è definita in questo modo:

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

Se l'host è un registered name allora l'endpoint deve utilizzare un servizio esterno di risoluzioni di nomi per trovare il riferimento assoluto al server. Se non viene specificata la porta di default è la 5683. Un argomento è di solito nella forma *"chiave=valore"*.
Nella specifica viene consigliato di fare uso di URI descrittivi e di dimensione ridotta proprio per l'utilizzo in ambenti molto vincolati sia per banda che per energia disponibile (WSN).

La sintassi per coaps è invece definita in questo modo:

coaps-URI = "coaps:" "//" host [":" port] path-abempty ["?" query]

La porta predefinita in questo caso è la 5684.
Rispetto allo scheme le risorse rese disponibili non hanno la stessa identità condivisa anche se gli identificatori indicano lo stesso host che sta in ascolto sulla stessa porta UDP. Questo perché coap e coaps hanno namespace differenti.

Alcune regole sono specificate nell'RFC di CoAP per quanto riguarda gli URI e sono:

- Se una porta è uguale a quella di default allora è meglio non specificarla
- Gli scheme e gli host sono case-insensitive e normalmente sono specificati in lettere minuscole mentre tutti gli altri componenti invece sono case-sensitive

I passi da eseguire nell'algoritmo di decomposizione delle URI in opzioni sono:

1. Se la stringa URI non è un URI assoluto allora termina l'algoritmo con una fail.
2. Risolvi l'URI attraverso un processo di risoluzione dei riferimenti.
3. Se l'URI non ha un componente scheme come coap o coaps allora termina l'algoritmo con una fail.
4. Se l'URI ha un componente fragment allora termina l'algoritmo con una fail.
5. Viene preso il componente host e viene inserito nella opzione *Uri-Host* e successivamente converti tutti gli encoding percent.
6. Viene preso il componente port e viene inserito nell'*Uri-Port*. Se non è presente viene preso il valore di default.
7. Se è presente un componente "/" allora vai al passo successivo dopo aver convertito ogni encoding percent.
8. Se è presente un componente query allora includi *Uri-Query* e inserisci il valore del componente all'interno.

Per quanto riguardo l'algoritmo di composizione delle URI dalle opzioni è esattamente lo stesso procedimento ma al posto di inserire i componenti nelle opzioni si va a creare la stringa prendendo le stesse opzioni come sorgenti di informazione.

Di seguito alcuni esempi riguardanti gli URI CoAP:

- *Destination IP Address* = [2001:db8::2:1] e *Destination UDP Port* = 5683

```
coap://[2001:db8::2:1]/
```

- *Destination IP Address* = [2001:db8::2:1], *Destination UDP Port* = 5683 e Uri-Host = "example.net"

```
coap://example.net/
```

- *Destination IP Address* = [2001:db8::2:1], *Destination UDP Port* = 5683, *Uri-Host* = "example.net", *Uri-Path* = ".well-known" e *Uri-Path* = "core"

```
coap://example.net/.well-known/core
```

- *Destination IP Address* = [2001:db8::2:1], *Destination UDP Port* = 5683, *Uri-Host* = "xn--18j4d.example" e *Uri-Path* = la stringa composta da caratteri Unicode U+3053 U+3093 U+306b U+3061 U+306f, di solito rappresentata in UTF-8 come E38193E38293E381ABE381A1E381AF esadecimale

```
coap://xn--18j4d.example/%E3%81%93%E3%82%93%E3%81%AB%E3%81%A1%E3%81%AF
```

- *Destination IP Address* = 198.51.100.1, *Destination UDP Port* = 61616, *Uri-Path* = "", *Uri-Path* = "/", *Uri-Path* = "", *Uri-Path* = "", *Uri-Query* = "//", *Uri-Query* = "?&"

```
coap://198.51.100.1:61616//%2F//?%2F%2F&?%26
```

3.1.2.5 Proxying

Un proxy è un endpoint CoAP che ha il compito di elaborare richieste per conto di altri endpoint. Ci sono diversi tipi di proxy:

1. *Forward-proxy* : può essere selezionato esplicitamente dai client
2. *Reverse-proxy* : inserito per gestire i server di origine
3. *CoAP-to-CoAP proxy* : effettuano il mapping delle richiesta da CoAP a CoAP
4. *Cross-proxy* : permettono la traduzione tra protocolli differenti

3.1.2.5.1 Operazioni sui proxy

Un proxy ha bisogno di determinare i parametri delle richieste per una destinazione. Questo è specificato esplicitamente in un forward-proxy mentre per i reverse-proxy dipende dalla configurazione specifica. In particolare abbiamo bisogno di qualche forma di traduzione nello spazio dei nomi.
Relativamente alla cache, le opzioni che sono marcate come *Safe-to-forward* indicano che saranno incluse nella *Cache-Key*. Se una risposta è generata fuori dalla cache il valore di *Max-Age* viene impostato secondo la formula: *proxy-max-age = original-max-age - cache-age*.
Tutte le opzioni presenti in una richiesta proxy devono essere elaborate dal proxy. Un proxy CoAP-to-CoAP deve fare il forward dal server di origine di tutte le opzioni Safe-to-forward che non riconosce. Invece se ci sono opzioni unsafe che il proxy non riconosce bisogna che venga restituita la risposta *5.02 (Bad Gateway)*.

3.1.2.5.2 Forward proxy

CoAP distingue tra le richieste fatte direttamente ad un server e richieste fatte attraverso un proxy.

- Quelle indirizzate ai proxy specificano l'URI di richiesta come stringa nell'opzione *Proxy-URI*.

- Mentre in quella fatta al server l'URI è suddiviso in *URI-Host*, *URI-Port* e *URI-Path* e *URI-Query*.

3.1.2.5.3 Reverse proxy

I reverse-proxy non fanno uso di *Proxy-Uri* o *Proxy-Scheme* e per determinare la destinazione devono fare guessing sulle informazioni contenute nella richiesta.
Può crearsi un proprio namespace in cui vengono identificate queste risorse in modo tale da utilizzarle nelle richieste successive.
Può anche permettere al client di avere maggiore controllo su dove debba andare la richiesta.
Bisogna comunque stare attenti che nel processing della risorsa i valori di *ETag* siano differenti per ogni risorsa in modo tale da rispettare la specifica.

3.1.2.6 Metodi

I metodi utilizzati dal protocollo CoAP (nel rispetto del modello RESt) sono:

1. *GET* : permette di ricevere una rappresentazione per l'informazione che corrisponde alla risorsa identificata dall'URI. Se la richiesta contiene un *ETag* allora viene richiesto che l'*ETag* sia validato e che la rappresentazione della risorsa sia trasferita solo se la validazione fallisce.
2. *POST* : richiede che la rappresentazione racchiusa nella richiesta sia elaborata. La funzione che viene elaborata dipende sia dal server di origine che dalla risorsa target. La funzione base sarebbe quella di creare una nuova risorsa o effettuare un'operazione di update sulla risorsa presente. La risposta dovrebbe anche restituire l'URI della nuova risorsa creata.
3. *PUT* : richiede che la risorsa identificata dall'URI nella richiesta sia aggiornata oppure creata con la rappresentazione inclusa (data dall'opzione Content-Format).
4. *DELETE* : richiede che la risorsa identificata dalla richiesta URI sia cancellata.

3.1.2.6.1 Codici di risposta

Ogni metodo può restituire diversi codici di risposta:

- *2.xx* : Codici success
    - *2.01 (Created)* : usato in POST e PUT. Ha come risultato che la risposta è

creata all'URI richiesta o specificata nella Location-Path e non è cacheable.

- *2.02 (Deleted)* : usato in DELETE o POST. Il payload restituito è la rappresentazione del risultato dell'azione. La risposta non è cacheable e se è presente una cache bisogna marcare ogni risposta per la risorsa relativa memorizzata come non aggiornata.
- *2.03 (Valid)* : utilizzata solo per indicare che la risposta identificata dall'entity-tag incluso in *ETag* è valida. La risposta deve includere un'opzione *ETag* e non deve includere un payload.
- *2.04 (Changed)* : usata in POST e PUT. Il payload restituito è una rappresentazione del risultato dell'azione. La risposta non è cacheable e se è presente una cache bisogna marcare ogni risposta per la risorsa relativa memorizzata come non aggiornata.
- *2.05 (Content)* : ha la stessa funzione di HTTP 200 "OK" in GET. Il payload restituito è una rappresentazione del risultato dell'azione. Stesso comportamento associato alla cache come in precedenza.

- *4.xx* : Codici di errore client applicabili ad ogni metodo di richiesta e il server dovrebbe includere un payload di diagnostica.
  - *4.00 (Bad Request)*
  - *4.01 (Unauthorized)*
  - *4.02 (Bad Options)*
  - *4.03 (Forbidden)*
  - *4.04 (Not Found)*
  - *4.05 (Method Not Allowed)*
  - *4.06 (Not Acceptable)*
  - *4.12 (Precondition Failed)*
  - *4.13 (Request Entity Too Large)* : può includere un'opzione size che indica la dimensione massima che il server può gestire.
  - *4.15 (Unsupported Content-Format)*
- *5.xx* : Codici di errore server applicabili ad ogni metodo di richiesta e il server dovrebbe includere un payload di diagnostica.
  - *5.00 (Internal Server Error)*
  - *5.01 (Not Implemented)*
  - *5.02 (Bad Gateway)*

- *5.03 (Service Unvailable)* : utilizza l'opzione Max-Age per indicare il numero di secondi dopo il quale riprovare il tentativo.
- *5.04 (Gateway Timeout)*
- *5.05 (Proxying Not Supported)*

3.1.2.6.2 Riassunto delle opzioni

Questa è la tabella delle opzioni disponibili:

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

Con C=Critical, U=Unsafe, N=NoCacheKey, R=Repeatable

Queste invece sono le descrizioni:

- *Uri-Host* : host della risorsa richiesta
- *Uri-Port* : numero della porta della risorsa per il livello trasporto
- *Uri-Path* : specifica un segmento sul path assoluto verso la risorsa
- *Uri-Query* : specifica un argomento che parametrizza la risorsa richiesta
- *Proxy-Uri* : utilizzato per effettuare una richiesta verso un forward-proxy. Per evitare

loop un proxy, deve essere in grado di riconoscere tutti i nomi dei server. Ha la precedenza sulle opzioni precedenti (Uri-xxx). Come caso particolare può essere costruito a partire dalle opzioni precedenti (Uri-xxx) solo quando è presente il Proxy-Scheme.

- *Content-Format* : formato di rappresentazione del payload del messaggio. Non ci sono valori di default e i formati possibili sono *Text/plain, Charset=utf-8, Application/link-format, Application/xml, Application/octet-stream, Application/exi e Application/json*.
- *Accept* : utilizzato per indicare quale *Content-Format* è accettabile per il client. Non ci sono valori di default.
- *Max-Age* : tempo massimo che una risposta può restare in cache prima di essere considerata non più aggiornata.
- *ETag* : identificatore locale per la risorsa che permette di differenziare tra le rappresentazioni della stessa risorsa nel tempo. Generato dal server che fornisce la risorsa, quando viene ricevuto da un endpoint bisogna che venga trattato come *opaque* e non si facciano assunzioni sulla sua struttura o contenuto. Può essere considerato:
    1. Come opzione di risposta : valore corrente dell'entity-tag per la rappresentazione dell'oggetto
    2. Come opzione di richiesta : in una richiesta GET un endpoint ottiene una o più rappresentazioni tramite ETag response e può specificare quale è di suo gradimento.
- *Location-Path* : indicano una Uri che consiste in un path assoluto
- *Location-Query* : indicano una Uri che consiste in una query string. Più in genere indica un argomento che parametrizza la risorsa. È possibile in futuro definire altri modi per specificare l'Uri relativa attraverso i numeri di opzione 128, 132, 136 e 140 e costituiscono le opzioni *Location-**.
- *Opzioni di richiesta condizionali* : abilitano un client a chiedere ad un server di effettuare delle richieste se e solo se alcune condizioni indicate nelle opzioni sono soddisfatte. Con *If-Match* effettua la richiesta condizionale se il valore di un ETag corrisponde ad una o più rappresentazioni della risorsa e può essere utilizzata una o più volte. Con *If-None-Match* : come la precedente con la differenza che se la risorsa target esiste con quel determinato *ETag* allora la condizione non è soddisfatta.
- *Size1* : fornisce indicazioni sulla dimensione della rappresentazione della risorsa in una richiesta. Utilizzata nei trasferimenti a blocchi per indicare la dimensione massima di un blocco che il server può gestire in modo corretto

3.1.3. Discovery

3.1.3.1 Service discovery

Come parte dei servizi di discovery un client ha bisogno di conoscere gli endpoint utilizzati da un server. Un server viene trovato da un client conoscendo la URI di riferimento di una risorsa nel namespace del server oppure attraverso una comunicazione multicast CoAP.
Per quanto riguarda la porta utilizzata sia per coap che per coaps, gli endpoint possono utilizzarne di differenti in quanto lo spazio delle porte è dinamico.

3.1.3.2 Resource discovery

CoAP definisce un meccanismo standard per la resource discovery. I server forniscono una lista delle loro risorse (insieme ai metadati associati) in /*.well-known/core*.
Questi link sono di tipo *application/link-format* e permettono ad un client di capire quali risorse sono fornite e di che tipo sono.
Per massimizzare l'interoperabilità in un ambiente CoRE un endpoint CoAP dovrebbe supportare il formato CoRE Link per le risorse discoverable. Viene infatti definito un attributo "ct" che permette di ottenere informazioni sul Content-Format della risorsa ed è un identificatore intero a 16 bit.

3.1.4. Comunicazione multicast

I nodi che vogliono fornire dei servizi discoverable sugli indirizzi multicast di CoAP devono effettuare il join su di essi.

3.1.4.1 Messaging layer

Una richiesta multicast viene trasportata in un messaggio CoAP che è indirizzato ad un IP multicast invece che ad un endpoint specifico.
Le richieste devono essere messaggi NON e quando un server è a conoscenza che una richiesta CON è arrivata in multicast non deve effettuare una reply con un messaggio RST.

Se così fosse il messaggio RST inviato sarebbe uguale a quello in risposta di un messaggio unicast. Di conseguenza il client deve evitare di usare un Message ID che è ancora attivo da se stesso verso qualsiasi altro endpoint che potrebbe ricevere il messaggio multicast.
I messaggi multicast possono essere inviati solo su UDP e non in DTLS. Quindi le modalità di sicurezza non possono essere applicate in multicast.

3.1.4.2 Livello di richiesta-risposta

Un server quando riceve una richiesta in multicast può decidere di non rispondere immediatamente e scegliere un periodo di tempo entro il quale rispondere. Questo intervallo di tempo si chiama *Leisure*.

- La formula per il calcolo del Leisure (lowerbound) è:

$$Leisure_{LB} = \frac{S \cdot G}{R}$$

Dove G è la dimensione del gruppo, R è il target data transfer rate e S è la dimensione stimata della risposta. Ad esempio per una richiesta multicast con scope link-local su una rete IEEE 802.15.4 (6LowPAN) su 2.4 GHz: G impostato in modo conservativo su 100, R pari a 1 kB/s e S uguale a 100 bytes abbiamo un leisure di 10 secondi.
Quando viene creata la risposta solo il token deve fare match. Infatti non è assolutamente necessario che l'endpoint sorgente sia lo stesso dell'endpoint destinazione della richiesta originale (problema di sicurezza).
Il server comunque può anche decidere di non rispondere e questa scelta può dipendere ad esempio dall'analisi dei filtri che sono legati ad una certa query che il server riceve.

3.1.4.3 Caching

Quando un client fa una richiesta multicast deve comunque farla sempre sull'indirizzo multicast perché possono esserci altri membri che si sono iscritti successivamente.Fatto questo il client analizza sia le informazioni salvate sulla cache che quelle del risultato della query.
Una risposta ricevuta in risposta ad una richiesta GET su un indirizzo multicast può essere riutilizzata per una richiesta unicast. Una cache può rivalidare una risposta andando ad effettuare una richiesta GET sul relativo URI di richiesta unicast. Comunque una richiesta

GET in multicast non può contenere un'opzione ETag.

3.1.4.4 Proxying

Quando un proxy riceve le richieste su un indirizzo multicast non fa altro che ottenere una serie di risposte come descritto prima e restituirle al client. La specifica poi lascia all'implementazione il compito di gestire il Base-URI relativo all'URI di provenienza della richiesta che è stato modificato.

3.1.5. Security

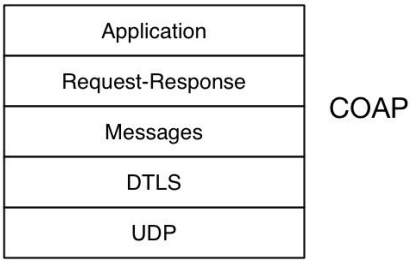
Dato che CoAP utilizza UDP non è possibile sfruttare SSL-TLS per aggiungere uno strato di sicurezza alle applicazioni. Viene utilizzato infatti DTLS (Datagram Transport Layer Security) che si appoggia ad UDP. Tipicamente i dispositivi che sono capaci di sfruttare DTLS supportano RSA-AES oppure ECC-AES.
Per mettere in sicurezza un dispositivo all'inizio è prevista una fase di provisioning in cui un dispositivo viene dotato con tutte le informazioni di cui ha bisogno. Alla fine di questa fase il dispositivo si troverà in una delle quattro modalità di sicurezza:

1. *NoSec* : non esiste un protocollo di sicurezza correntemente utilizzato dal dispositivo e bisogna andare ad utilizzare un meccanismo esterno.
2. *PreSharedKey* : DTLS abilitato ed è presente una lista di nodi con cui è possibile comunicare. Di solito si ha un rapporto 1:1 tra nodi e chiavi ma è possibile che 2 o più nodi condividano la stessa chiave perché facenti parte di un gruppo e quindi vengono autenticati come gruppo.
3. *RawPublicKey* : DTLS abilitato e il dispositivo ha una coppia di chiavi asimmetriche senza certificato.
4. *Certificate* : DTLS abilitato e il dispositivo ha una coppia di chiavi simmetriche con un certificato X.509 che è signed su una root di fiducia in comune.

CoAP non fornisce primitive di protocollo che permettono l’autenticazione e l’autorizzazione e quindi o bisogna utilizzare le primitive fornite da IPSec o DTLS oppure bisogna sfruttare gli oggetti (secured) all’interno del payload.
Se ci sono intermediari, CoAP non supporta la loro gestione per le operazioni di autenticazione.

3.1.5.1 DTLS secured CoAP

Questa è l'architettura di CoAP quando si utilizza DTLS:



DTLS è in pratica come TLS (utilizzato da HTTP) ma con funzionalità in più per gestire in modo corretto la natura inaffidabile del protocollo di trasporto UDP. In molte applicazioni come le WSN però tutte le modalità di DTLS non sono applicabili, infatti alcuni cifrari aggiungono una complessità troppo elevata per questi tipi di rete. Una volta che è stato completato l'handshake verrebbe aggiunto un overhead di 13 bytes per ogni datagramma più altri byte per gestire l'integrità.
Bisogna quindi scegliere cifrari specifici per ogni tipo di applicazione tenendo conto accuratamente del livello di sicurezza che si vuole ottenere.
DTLS come detto in precedenza non è applicabile a comunicazioni multicast.

3.1.5.2 Livello dei messaggi

Un endpoint che fa da client CoAP deve anche essere un client DTLS.

1. Dovrebbe infatti iniziare una sessione con il server sulla porta appropriata.
2. Quando l'handshake è completato poi può inviare i messaggi successivi come DTLS application data.
3. Nel momento in cui abbiamo creato una sessione le regole di matching sono che un messaggio è uguale ad un altro se fa parte della stessa sessione DTLS, ha la stessa epoch (che è in pratica un valore incrementato ad ogni cambiamento di stato del cifrario) e lo stesso Message ID. Quando si re-invia un messaggio CON si cambia il numero di sequenza DTLS (aumentato di uno) anche se il Message ID rimane lo stesso e la ritrasmissione non può avvenire assolutamente tra epoch diverse.

Nel caso di connessioni DTLS in RawPublicKey e Certificate le autenticazioni mutue possono essere riutilizzate per scambio di messaggi futuri in entrambe le direzioni.
Quando ci troviamo in una rete WSN dovremmo cercare di mantenere la connessione attive

per un tempo più elevato possibile in modo tale da non dover rinegoziare ogni volta una nuova connessione. Se così non fosse avremmo una comunicazione molto inefficiente.

3.1.5.3 Livello di richiesta-risposta

Dato che la sessione DTLS e la epoch devono essere le stesse per poter inviare una risposta significa che le risposte ad una richiesta DTLS devono essere sempre sicure e qualsiasi tentativo di utilizzare una risposta NoSec verrà rifiutata perché non fa match con la richiesta.

3.1.5.4 Identità dell'endpoint

I dispositivi devono supportare SNI (Server Name Indication) con cui devono indicare la loro Authority nel campo SNI HostName. Il motivo di questa scelta è che quando il dispositivo si trova ad gestire Authority multiple sa sempre quali chiavi usare per creare la connessione DTLS.

3.1.5.5 PreShared Key

Nel momento in cui effettuiamo una connessione verso un nuovo nodo allora il sistema seleziona una chiave appropriata e crea una sessione DTLS utilizzando la modalità PSK (Pre-Shared Key). Il supporto mandatory è TLS_PSK_WITH_AES_128_CCM_8.

3.1.5.6 RawPublicKey

Per quanto riguarda questa modalità abbiamo a disposizione una coppia di chiavi asimmetriche senza un certificato X.509. Il tipo e la lunghezza della chiave dipende dal cifrario utilizzato. La cypher suite mandatory in questo caso è TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8. L'algoritmo hash è invece SHA-256.

3.1.5.7 Provisioning

La modalità RawPublicKey è stata progettata in modo tale da essere facilmente gestibile in ambienti M2M. Come detto in precedenza bisogna supportare SHA-256-120 (che sarebbe SHA-256 troncato a 120 bit). I dispositivi possono supportare anche un numero di bit

maggiore (e non è consigliato invece l'utilizzo di un numero di bit minore). L'identificatore principale di un dispositivo può essere sia binario che human-readable.
Queste sono le fasi di provisioning per la comunicazione M2M:

1. L'identificatore di ogni nodo è immagazzinato con la lettura di un barcode sul dispositivo stesso.
2. Successivamente vengono inseriti su un corrispondente endpoint come ad esempio un data collection server M2M.
3. Alla fine viene creata una access control list (ACL) per tutti i dispositivi attraverso la quale poi potranno essere create sessioni DTLS.

3.1.5.8 Certificati X.509

Per quanto riguarda l'utilizzo di certificati X.509 la cypher suite mandatory è TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8. La firma deve utilizzare come algoritmo di hashing SHA-256. Quando creiamo una nuova connessione bisogna andare a verificare il certificato del dispositivo remoto e se il nodo remoto ha una sorgente di tempo assoluto deve verificare la data di scadenza del certificato.

3.1.5.9 Considerazioni sulla sicurezza

3.1.5.9.1 Parsing del protocollo e processing degli URI

Un'applicazione che si affaccia sulla rete può avere delle vulnerabilità nella logica di elaborazione che possono portare al crash di nodi remoti o anche a poter eseguire del codice malevolo su di essi (sempre da remoto). CoAP tenta di risolvere questo problema andando a realizzare un parser molto semplice in cui ogni opzione ha un suo significato e non è ambigua. Il processing degli URI viene spostato verso i client riducendo la possibilità di introdurre vulnerabilità lato server. L'implementazione per quanto riguarda il controllo degli accessi deve assicurare che non vengano introdotte discrepanze dirette tra:

- Codice che gestisce il controllo degli accessi.
- Codice che fornisce la risorsa indirizzata dall'URI.

Rimane comunque da dire che esiste un parser abbastanza complesso che potrebbe avere al suo interno bug di difficile scoperta come ad esempio quello per il CoRE Link Format.

3.1.5.9.2 Proxying e caching

I proxy per loro natura sono dei men-in-the-middle e quindi sono degli elementi importanti per quanto riguarda la ricerca di vulnerabilità all'interno del protocollo CoAP dato che spezzano qualsiasi protezione IPSec e DTLS tra il client e il server.
Questo aspetto diventa molto rilevante nel caso in cui il proxy abbia anche la funzione di cache. Infatti in quel caso dato che CoAP non ha nessuno meccanismo di Cache-Control (che invece ha HTTP) non riesce a gestire in modo appropriato la sicurezza degli elementi presenti nella cache. In una possibile implementazione si dovrebbe tener conto sia del controllo degli accessi per quanto riguarda le richieste che generano una entry nella cache e sia dei valori che vengono memorizzati nella cache. Inoltre non si dovrebbe rendere disponibile la possibilità di salvare in cache alle richieste che hanno proprietà di sicurezza di livello trasporto molto basse.
Le risposte per quanto riguarda le richieste che viaggiano su coaps non devono essere condivise, a meno che non si riesca a fare controllo degli accessi ad un livello più basso per quanto riguarda i contenuti della cache stessa.

3.1.5.9.3 Rischio di amplificazione

I server CoAP generalmente rispondono ad un pacchetto di richiesta con un pacchetto di risposta. Questi pacchetti sono normalmente più grandi di quelli di richiesta e quindi un attaccante potrebbe utilizzare i nodi CoAP per creare un attacco di tipo DoS (Denial of Service) attraverso l'amplificazione data dalle risposte, per ottenere un flooding di pacchetti di dimensione elevata verso una determinata destinazione.
È un problema particolare in quanto possono esserci nodi che hanno un tipo di accesso NoSec, sono accessibili dall'attaccante e possono accedere a innumerevoli vittime potenziali su Internet in quanto attraverso l'utilizzo del protocollo UDP non c'è modo di verificare l'indirizzo sorgente di un determinato pacchetto.
Un modo per evitare questo tipo di attacco è quello di fornire dei fattori di amplificazione elevati solo se le richieste sono autenticate. Un'alternativa sarebbe quella di fornire rappresentazioni delle risorse elevate solo sottoforma di blocchi per diminuire la dimensione del pacchetto (e di conseguenza il flooding).
Il fatto che CoAP supporti l'uso di indirizzi multicast è sorgente anch'essa di attacchi di tipo DoS soprattutto su reti limitate. Di nuovo, per limitare questo tipo di possibilità non si

dovrebbero accettare richieste multicast se non sono autenticate.

3.1.5.9.4 Attacchi di spoofing sull'indirizzo IP

Data la mancanza di un handshake in UDP, un endpoint è capace di leggere qualsiasi informazione circolante sulla rete e quindi può eseguire qualsiasi azione a danno degli altri endpoint. Può infatti effettuare:

- Lo spoofing di un messaggio RST in risposta ad un messaggio CON o NON e quindi rendere un endpoint ignaro di una richiesta.
- Lo spoofing di un ACK in risposta ad un messaggio CON e quindi fa in modo che il mittente non ritrasmetta il messaggio.
- Lo spoofing di un'intera risposta con payload e opzioni modificate.
- Lo spoofing di una richiesta multicast per un nodo target e questo potrebbe portare ad un collasso o congestione di rete.
- Osservazione dei messaggi che circolano nella rete.

Una possibile soluzione a questo problema è quello di utilizzare dei token che seguano un certo algoritmo in modo tale che se un messaggio di risposta ad un messaggio CON non ha un determinato token allora significa che è stato intercettato e modificato.
Con o senza spoofing della sorgente un client può tentate di fare l'overloading di un server inviando richieste molto complesse. Questo può risultare efficace soprattutto verso gli endpoint molto vincolati che hanno una durata di batteria molto limitata. L'attacco viene chiamato battery depletion attack.
Inoltre un attaccante potrebbe effettuare lo spoofing di un messaggio andando ad inserire un indirizzo sorgente reale togliendo così la possibilità a quel determinato endpoint di comunicare con il server perché di default la proprietà NSTART (numero di comunicazioni simultanee) è limitata ad 1.

3.1.5.9.5 Attacchi cross-protocol

La possibilità di poter utilizzare un endpoint CoAP per inviare pacchetti ad un indirizzo sorgente fake può essere utilizzata anche per inviare attacchi su altri protocolli che sono in listening di pacchetti UDP ad un dato indirizzo.
La procedura di un attacco sarebbe questa:

1. L'attaccante invia un messaggio ad un endpoint CoAP con un fake source address.

2. L'endpoint CoAP risponde con un messaggio ad un indirizzo sorgente specificato.
3. La vittima riceve un pacchetto UDP che interpreta secondo un altro protocollo e non secondo CoAP.

In generale dato che i protocolli basati su UDP non hanno informazioni relative al contesto in cui vengono utilizzati sono molto facilmente utilizzabili come tramite per attacchi tra protocolli diversi.
Una possibile soluzione sarebbe quella di controllare in modo stretto la sintassi dei pacchetti ricevuti. Ci sono comunque delle possibilità per gli attaccanti di far sembrare un pacchetto di un certo protocollo come un pacchetto di un protocollo diverso.
Per quanto riguarda invece le considerazioni riguardo la sicurezza in DTLS (sempre per attacchi cross-protocol) bisogna dire che se la stessa connessione DTLS viene utilizzata per trasportare informazioni di protocolli multipli la sicurezza viene comunque a mancare.

3.1.5.9.6 Nodi vincolati

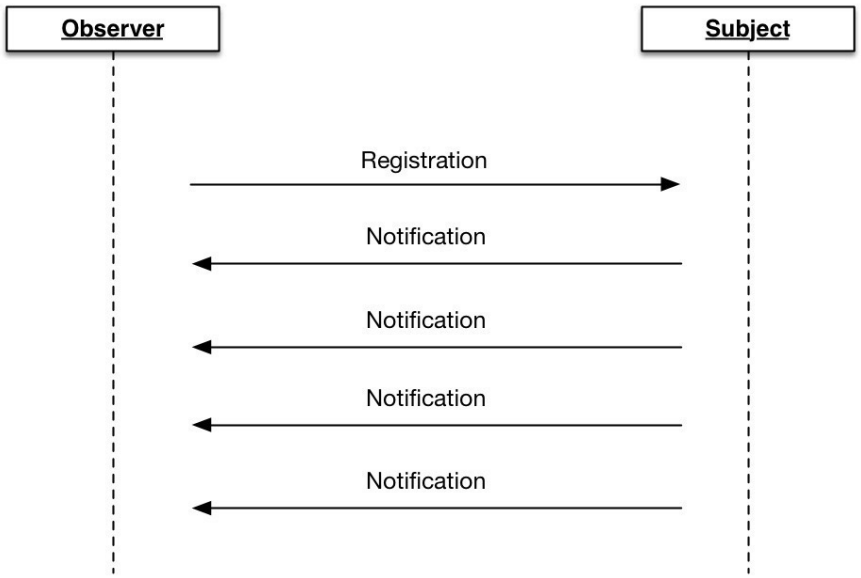
Chi deve realizzare implementazioni su nodi limitati che fanno parte ad esempio di una WSN si trova comunque senza una buona fonte di entropia (necessaria per una protezione efficace). Quindi il nodo non deve essere utilizzato per elaborare processi che hanno bisogno di un grado elevato di entropia (come ad esempio la generazione di chiavi per cifrare) e dovrebbe affidarsi a chiavi generate esternamente e consegnate (sempre in modo sicuro).
Poi bisogna gestire, appunto, in modo efficace la consegna delle chiavi per non cadere in situazioni di tampering. In particolare se andiamo ad assegnare una chiave condivisa ad un gruppo di nodi si può rendere un entità qualsiasi tra di esse capace di rompere la sicurezza dell'intera rete.
Infine dato che i nodi hanno comunque risorse di elaborazione molto limitate sono particolarmente suscettibili ad attacchi sul timing.

3.1.6. Sottoscrizioni e notifiche

CoAP non funziona bene quando un client è interessato ad una risorsa su un periodo di tempo prolungato. È quindi presente un'estensione che permette di ricevere una rappresentazione e futuri aggiornamenti sulla stessa da parte del server.

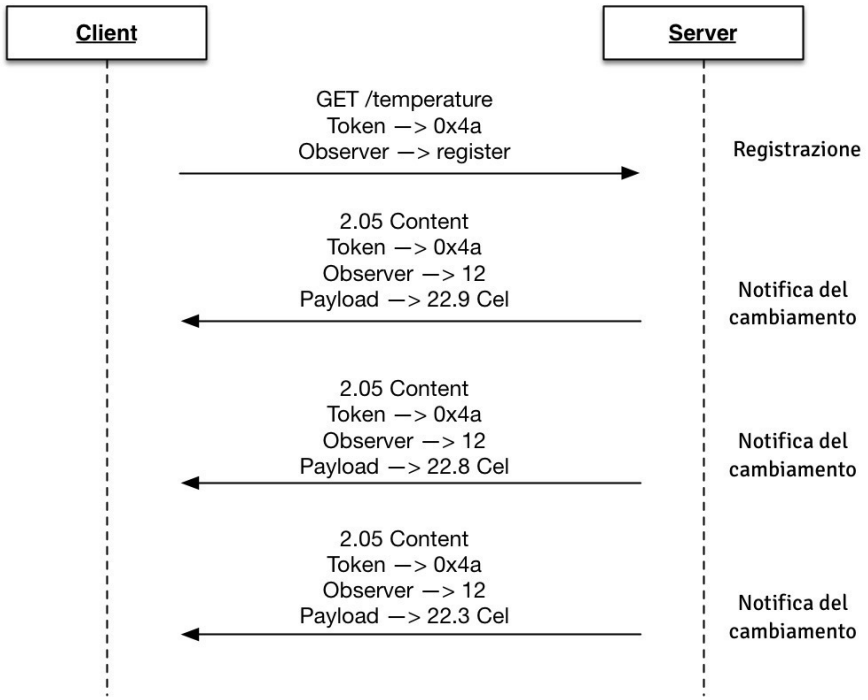
3.1.6.1 Design Pattern

Il design pattern observer consiste in componenti chiamati observers registrati su uno specifico argomento chiamato subject su cui sono interessati. Il subject è il responsabile dell'amministrazione degli observer registrati. Questo è lo schema del design pattern:



- *Subject* : in CoAP non è altro che la risorsa nel namespace del server.
- *Observer* : client che registra l'interesse nella risorsa inviando un messaggio GET di richiesta al server.
- *Registration* : il client registra il suo interesse nella risorsa con il messaggio GET
- *Notification* : quando lo stato di una risorsa cambia, il server notifica ogni client nella lista degli observers della risorsa.

Questo è lo schema di una comunicazione generale:



- Sia la richiesta di registrazione che le notifiche sono identificate da un'opzione *Observe*.
- Se invece il client si de-registra, rifiuta una notifica o la trasmissione di una notifica eccede il timeout considerato, allora l'observer viene rimosso dalla lista contenuta nel subject.

Secondo il modello di consistenza il client ed il server possono non essere sincronizzati per quanto riguarda la situazione attuale della risorsa interessata per diversi motivi:

- C'è sempre latenza tra il cambiamento di stato della risorsa e la ricezione della notifica.
- I messaggi CoAP possono andare persi e quindi il client può credere che ci sia uno stato vecchio fino a quando non arriva una nuova notifica.
- Il server può erroneamente pensare che il client non sia più interessato alle notifiche di una risorsa e quindi smette di inviare le notifiche.

Il protocollo risolve questi problemi in diversi modi:

- Segue un approccio best-effort per inviare la rappresentazione corrente al client dopo un cambiamento di stato.
- Aggiunge alle notifiche una durata massima affinché lo stato attuale sia accettabile e dopo il quale è considerato non sincronizzato con il server.
- Segue il principio della consistenza eventuale e garantisce che se la risorsa non ha un

cambiamento nello stato allora tutte gli observer registrati abbiano la rappresentazione corrente dell'ultimo stato della risorsa.

L'autorità che determina sotto quali condizioni le risorse cambiano il loro stato è il server. Il protocollo non specifica comunque possibili trigger o soglie. Quindi dal punto di vista dell'implementazione è il server che deve tenere conto del contesto dell'applicazione e agire di conseguenza.

3.1.6.2 Opzione Observe

L'opzione Observe ha questo formato:

No.	C	U	N	R	Name	Fornat	Length	Default
6		x	-		Observe	uint	0-3 B	(no)

[C (Critical), U (Unsafe), N (No-Cache-Key), R (Repeatable)]

Il valore dell'opzione è di 24 bit. Non fa parte della cache-key e quindi quando abbiamo una risposta cacheable ottenuta con un'opzione *Observe* dobbiamo rimuovere l'opzione prima di restituire la risposta.

3.1.6.3 Lato client

Un client registra il proprio interesse nella risorsa inviando un messaggio GET con l'opzione *Observe* impostata su register (0). Se il server restituisce una risposta *2.xx* allora il client può essere sicuro che è stato aggiunto alla lista degli observer e sarà notificato del cambiamento dello stato della risorsa.

Le notifiche sono risposte addizionali inviate dal server in risposta ad un messaggio GET. Ogni notifica include un token specificato dal client nel messaggio GET e inoltre è presente un numero di sequenza che permette il corretto ordinamento dei pacchetti.
Quando la risorsa non è più disponibile, il server invia un messaggio *4.04* e cancella i sottoscrittori dalla lista degli observer.

Per quanto riguarda il caching sono supportati sia il modello di aggiornamento che il modello

di validazione.

Secondo il modello di aggiornamento un client può memorizzare una notifica come una risposta nella sua cache e utilizzare la notifica memorizzata senza ricontattare il server.
Un client comunque non può essere interessato ad ogni singolo stato di una risorsa e il server utilizza l'opzione Max-Age per indicare un intervallo di tempo dopo il quale il client deve considerare la risorsa non più aggiornata. In questo caso deve inviare un nuovo messaggio GET con lo stesso token del messaggio originale. È comunque consigliato che il client non invii questo messaggio se ha ancora disposizione una risposta valida nella propria cache.
È consigliato anche che il client attenda altri 5 o 15 secondi dopo il Max-Age time per evitare di inviare il messaggio in concorrenza con altri client (non provocando così lo storming di richieste).

Secondo il modello di validazione quando un client ha una o più modifiche memorizzate nella propria cache per una determinata risorsa può utilizzare l'opzione *ETag* nel messaggio GET per indicare al server quale notifica utilizzare. Se la risorsa osservata viene identificata da un'opzione *ETag* il server può inviare un messaggio *2.03 (Valid)* invece di una notifica *2.05 (Content)*.

Le notifiche possono arrivare in un ordine differente rispetto a quello di invio. Dato che l'obiettivo è quello di mantenere sincronizzato lo stato della risorsa tra client e server, un client non deve considerare come aggiornata una notifica che arriva dopo un'altra che invece riguarda un aggiornamento successivo.
Ci sono tre condizioni che devono essere soddisfatte affinché una notifica in arrivo sia più aggiornata rispetto all'ultima notifica ricevuta:

1. $(V1 < V2 \text{ e } V2 - V1 < 2^{23})$
2. $(V1 > V2 \text{ e } V1 - V2 > 2^{23})$: le prime due condizioni verificano che V1 è più piccolo di V2
3. (T2 > T1 + 128 secondi) : assicura che il tempo trascorso tra i due messaggi non sia così grande rispetto alla differenza che può avere il cambiamento da V1 a V2. La durata di 128 secondi stata scelta come il numero arrotondato più grande di MAX_LATENCY. Quindi dopo 128 secondi non viene controllato il numero di versione.

Un client che non è più interessato a ricevere notifiche per una risorsa può semplicemente inviare un messaggio RST al server in modo tale che lo cancelli dalla lista degli observer.
È possibile anche effettuare una de-registrazione attraverso un messaggio GET con all'interno un'opzione Observe con il valore posto a deregister (1).

3.1.6.4 Lato server

La entry nella lista degli observers ha una chiave che specifica il client nel messaggio di GET. Se è presente già un altro client con la stessa chiave il server non deve aggiungere una nuova entry ma deve rimpiazzare o aggiornare quella esistente. In questo caso la risposta restituita al client non deve avere l'opzione *Observe* in modo tale che il client capirà che non sarà notificato delle modifiche successive alla risorsa.

Se sono presenti più osservatori nella lista nel server, l'ordine con il quale i client sono notificati non è definito. Il server infatti è libero di scegliere quale ordine preferisce.
Il *Content-Format* specificato nella notifica *2.xx* deve essere lo stesso di quello utilizzato nella risposta iniziale al messaggio GET.

Quando la risorsa non ha cambiamenti e il client ha una rappresentazione corrente memorizzata, il server non ha bisogno di mandare notifiche. Comunque se il client non riceve notifiche non può sapere se lo stato osservato e lo stato attuale siano sincronizzati. Per risolvere questo problema viene utilizzato il valore di *Max-Age* dopo il quale la risorsa viene considerata stale.

Per facilitare il lavoro del client nel riordinare i messaggi arrivati, il server imposta il valore di ogni notifica ad un intero di 24 bit in modo incrementale. La sequenza può iniziare a qualsiasi valore ma non deve aumentare così velocemente da avere più di 2^{23} incrementi in meno di 256 secondi.
La scelta di questo valore permette di avere al massimo 65.536 notifiche al secondo. Dato che comunque utilizziamo questo protocollo su dispositivi limitati il massimo numero di notifiche è pari a 32.768.

3.1.6.5 Trasmissione

Il server può scegliere di evitare di inviare una notifica se sa già che ne manderà subito un'altra (come per esempio quando una risorsa cambia frequentemente). Il server però deve comunque assicurarsi che riceva l'ACK dell'ultima notifica inviata al client. Infatti se il client invia l'ACK allora significa che è interessata a ricevere ulteriori notifiche.
Per accelerare la trasmissione delle notifiche il server può inviarle in un messaggio di tipo NON rispetto ad un messaggio di tipo CON. Deve comunque inviare un messaggio di tipo CON ogni 24 ore in modo tale da capire se un client è ancora interessato a ricevere notifiche per quella risorsa e quindi se deve essere rimosso dalla lista degli observer.

Il controllo della congestione di base in CoAP è ottenuto tramite il meccanismo di back-off esponenziale facendo ricadere la responsabilità del controllo di questa operazione solo sui client.
Quando una singola richiesta porta ad un numero infinito di notifiche allora la responsabilità addizionale deve essere posta sul server. Infatti il server deve strettamente limitare il numero di connessioni simultanee al valore NSTART (1 di default). Il server comunque non dovrebbe inviare più di una notifica NON per RTT (Round Trip Time) in media ad un client. Se non si riesce a stimare l'RTT allora non deve inviare più di una notifica ogni 3 secondi.

Anche se il server sta attendendo la fine di una trasmissione di una vecchia notifica e nel frattempo lo stato di una risorsa è cambiato, allora dovrebbe smettere di inviare la versione precedente e trasmettere la nuova rappresentazione al client.
Quindi può effettuare l'abort della vecchia notifica e iniziare una nuova trasmissione per lo stato aggiornato. Se la risorsa cambia stato più volte si tiene conto solo dell'ultimo.
La nuova notifica è trasmessa in questo modo:

1. Viene creato un nuovo Message ID
2. Se la trasmissione precedente stava per raggiungere il timeout allora bisogna copiare i parametri del messaggio, incrementare il contatore di ritrasmissione e raddoppiare il timeout.

3.1.6.6 Intermediari

Tra il client e il server possono esserci uno o più intermediari CoAP. In questo caso il client

registra il proprio interesse con il primo intermediario e agisce come se stesse comunicando col server. É ora compito dell'intermediario mantenere il client aggiornato sullo stato della risorsa. Affinché tutto vada a buon fine l'intermediario dovrebbe immedesimarsi nel client. La comunicazione tra ogni coppia di hops è indipendente. Quindi ogni hop deve generare i propri valori per quanto riguarda i messaggi di richiesta.
Se due o più client hanno registrato il loro interesse ad una risorsa con un intermediario, l'intermediario deve registrarsi solo una volta con l'hop successivo ed effettuare il broadcast delle notifiche a tutti i client registrati. Questo tipo di comportamento migliora la scalabilità. Un intermediario, comunque, non deve sempre agire per conto di un client ma può anche osservare una risorsa di sua iniziativa per mantenere la propria cache aggiornata.

3.1.6.7 Web Linking

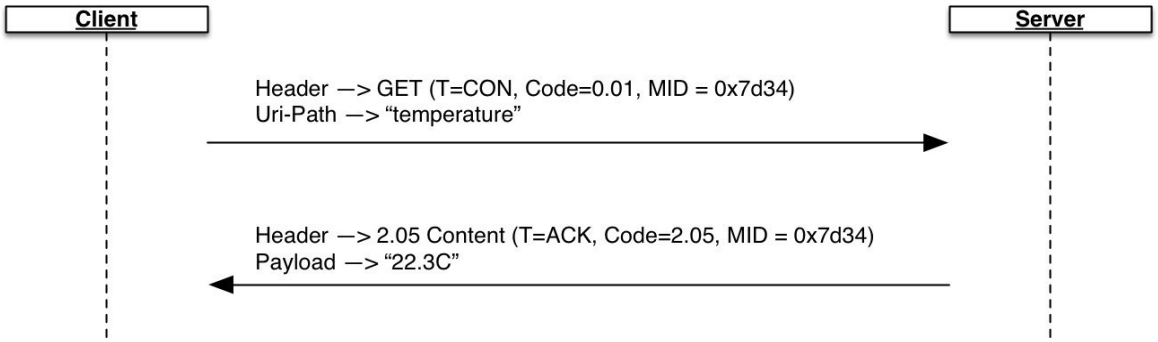
Un Web Link ad una risorsa può includere un attributo "obs". Questo permette di indicare che la destinazione di un link è utile per l'osservazione. L'attributo non ha valore e quindi ogni valore presente dev'essere ignorato.

3.1.6.8 Considerazioni sulla sicurezza

Le risorse osservabili possono drammaticamente aumentare l'effetto negativo degli attacchi di amplificazione. Non solo i messaggi e le notifiche possono essere molto più grandi dei messaggi di richiesta ma possono essere generati in un numero molto elevato per notificare il cambiamento di uno stato.
Una soluzione a questo problema è quella di autenticare il client. In questo modo il server può limitare il numero di notifiche che puoi inviare ad un client che non è stato autenticato.
Come ogni protocollo che crea stato in una macchina, gli attaccanti possono tentare di esaurire le risorse che il server ha a disposizione. Per evitare questo problema bisogna applicare una access control list per la creazione di uno stato.
Gli intermediari devono essere implementati in modo ottimale per assicurarsi che non siano impiegati per creare un ciclo. Per spezzare qualsiasi tipo di ciclo bisogna implementare cache specifiche per il forwarding delle notifiche negli intermediari.

3.1.7. Esempi di comunicazione

3.1.7.1 GET con risposta piggybacked

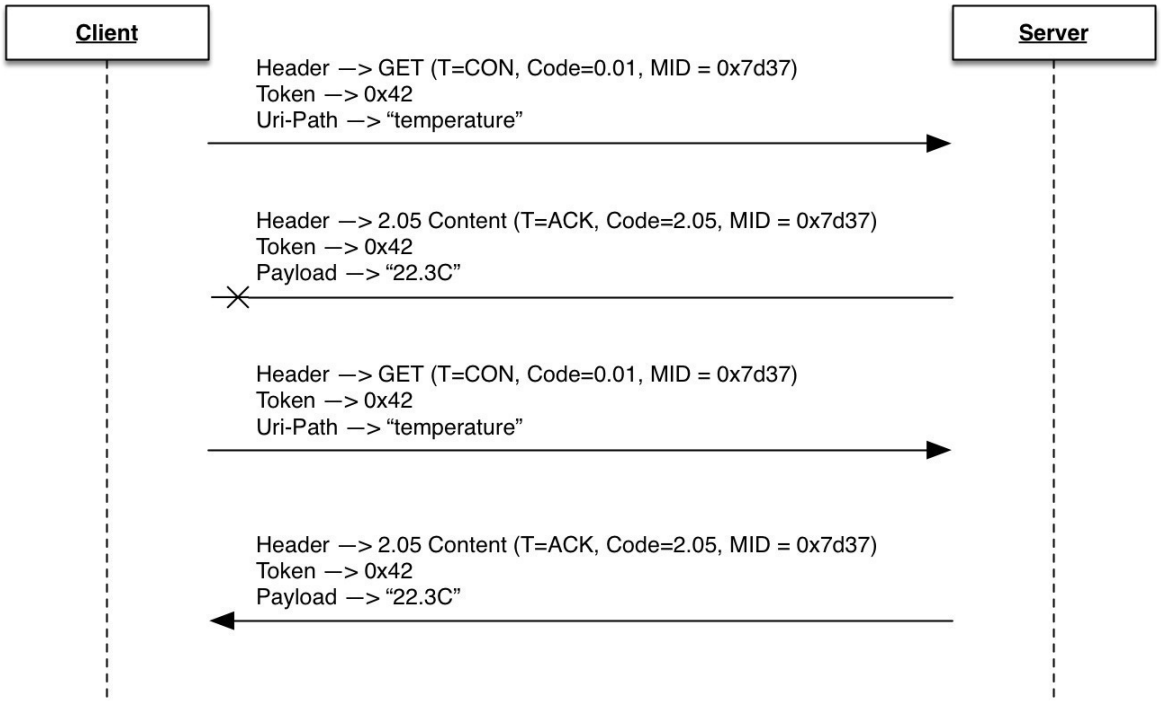


Indirizzo della risorsa : coap://server/temperature

Dimensione del payload della GET : 11 Byte (dim. totale della richiesta : 16 Byte)

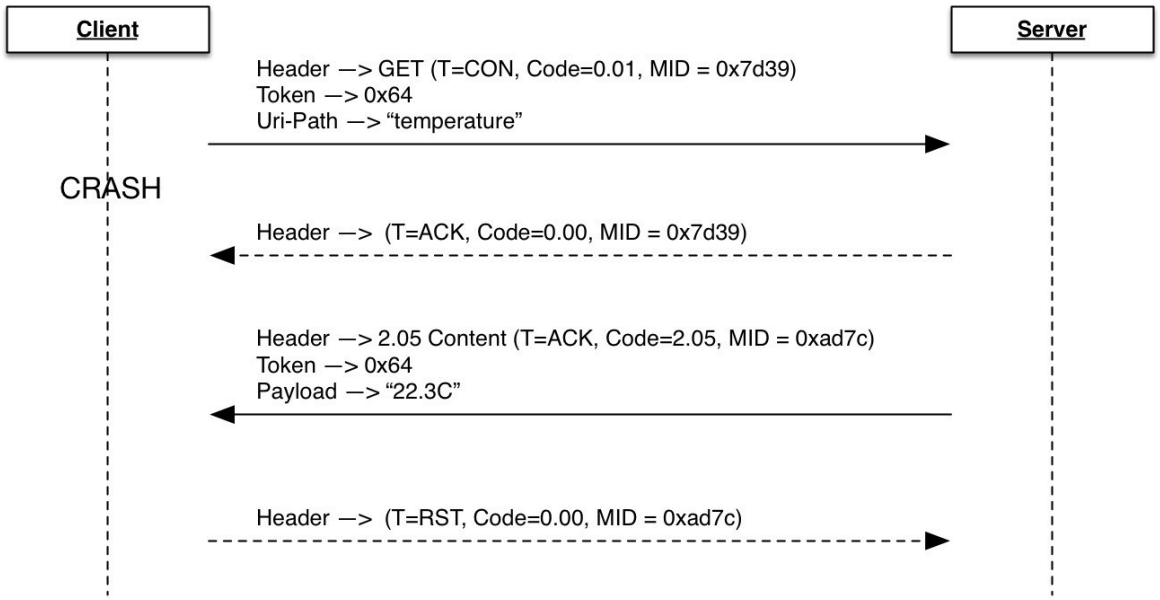
Dimensione del payload della 2.05 Content : 6 Byte (dim. totale della risposta : 11 Byte)

3.1.7.2 GET con perdita del primo ACK



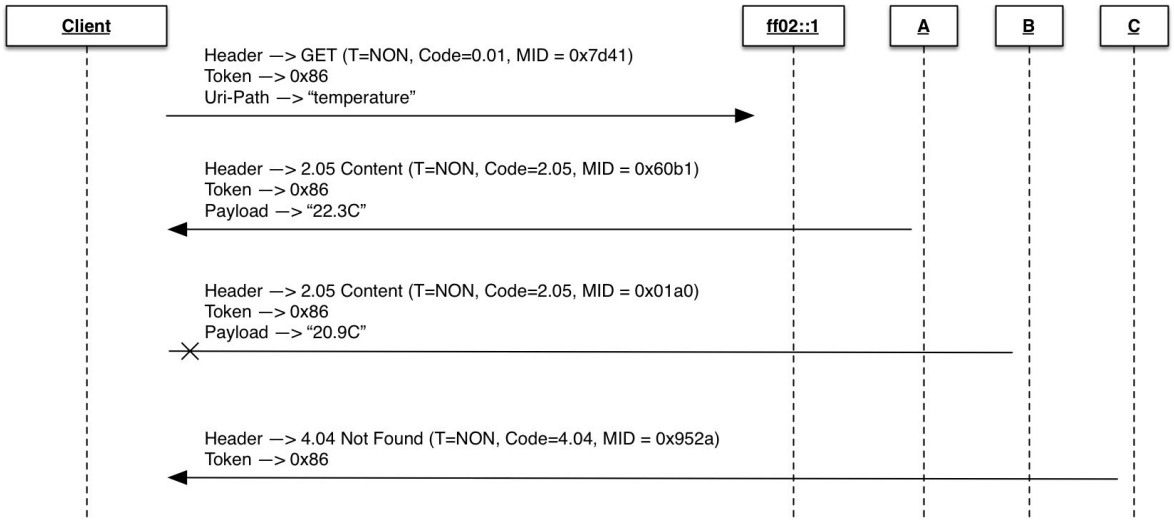
Il client dopo un intervallo di tempo ACK_TIMEOUT re-invia la richiesta.

3.1.7.3 GET con crash del client dopo l’invio del messaggio CON



In questo caso il client prima perde il suo stato (magari per un crash o per un riavvio del sistema) dopo l’invio di un messaggio CON. La risposta dal server separata arriva successivamente. In questo caso il client rifiuta la risposta del messaggio CON con un messaggio RST.

3.1.7.4 GET con invio messaggio NON in multicast



Tutti i nodi sono in uno scope link-local. Su quel determinato link ci sono tre nodi (A, B e C).

I server A e B hanno una risorsa che fa match e quindi inviano un messaggio NON 2.05 Content. La risposta inviata da B viene persa. C non ha una risorsa che fa match e quindi invia un messaggio NON 4.04 (Not Found).

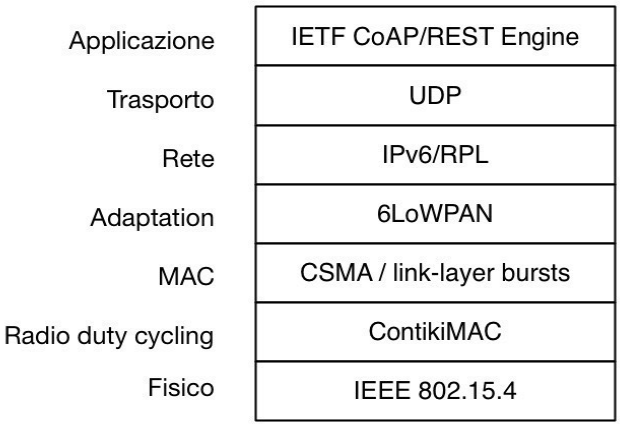
3.1.8. Implementazioni disponibili

3.1.8.1 Erbium. Un'implementazione CoAP low-power per Contiki

Contiki è sistema operativo open-source, progettato per IoT, che permette di connettere micro-controllori a basso costo ad Internet.
Le caratteristiche principali sono:

- Allocazione di memoria ottimizzata per dispositivi con pochi kB disponibili
- Gestione completa dello stack di rete IP
- Progettato per lavorare a basso consumo di potenza
- Ha un enorme supporto di protocolli di rete progettati per le WSN
- Caricamento dinamico dei moduli
- Contiene un simulatore di rete chiamato Cooja
- Supporto per sleepy router
- Supporto ai protothread (misto di meccanismi event-driven e multi-thread)
- File system leggero chiamato Coffee
- Supporto a stack leggero per operazioni semplici chiamato Rime

Quando ci troviamo in un ambiente wireless nella IoT abbiamo bisogno di protocolli power-efficient e i protocolli esistenti sono stati progettati senza considerare questo aspetto. Il componente che consuma di più in un sistema wireless è il transceiver radio. L'obiettivo di Erbium è quello di avere un tempo di occupazione radio di ogni dispositivo minimo che si traduce nel mantenere la componente radio il più possibile spenta.
Lo stack protocollare dell'implementazione di CoAP per Contiki è questo:



3.1.8.2 Libcoap Su TinyOS

Libcoap è una libreria open-source sviluppata in C, sotto licenza GPL, che implementa il protocollo CoAP nella sua forma base descritta nell'RFC 7252.
TinyOS invece è un sistema operativo open-source, sotto licenza BSD, progettato per dispositivi wireless a bassa potenza e con capacità limitate. Nella sua ultima versione è stato aggiunto anche uno stack completo IPv6 per 6LowPAN/RPL.
L'implementazione del protocollo CoAP su TinyOS utilizza libcoap come base e TinyOS blip-rpl come stack per la comunicazione UDP. Implementa solo parte di altri draft disponibili per ampliare le funzionalità di CoAP.
Ad esempio il trasferimento a blocchi e la gestione degli eventi non sono stati ancora implementati del tutto e addirittura alcuni metodi base non sono supportati (POST e DELETE, ma solo GET e PUT).

3.1.8.3 CoAPy

CoAPy l'implementazione del protocollo CoAP in Python, e l'obiettivo è quello di permettere la comunicazione tra client e server. È stato sviluppato da People Power Co ed è stato rilasciato sotto licenza BSD.
Le opzioni attualmente supportate per quanto riguarda l'header sono:

- *Content-Type*
- *Max-Age*
- *Uri-Scheme*
- *ETag*

- *Uri-Authority*
- *Location*
- *Uri-Path*
- *Block* che però è soltanto in fase sperimentale per il momento

3.1.8.4 jCoAP

jCoAP è invece l'implementazione di CoAP nel mondo Java. È compatibile sia con lo standard Java SE che con Android. È un progetto sponsorizzato dalla Siemens e le caratteristiche implementate sono:

- Messaggistica affidabile e non affidabile
- Risposta sia separata che piggy-backed
- Proxy CoAP-CoAP, CoAP-HTTP e HTTP-CoAP

3.1.8.5 nCoAP

nCoAP è un'altra implementazione del protocollo CoAP in Java.
Il supporto della specifica è molto ristretto. Infatti si limita a implementare il protocollo principale senza la parte relativa alla sicurezza. Supporta interamente l'RFC 6690 (CoRE Link Format) e il draft 14 dell'estensione Observe.
Il progetto è suddiviso in moduli maven che sono:

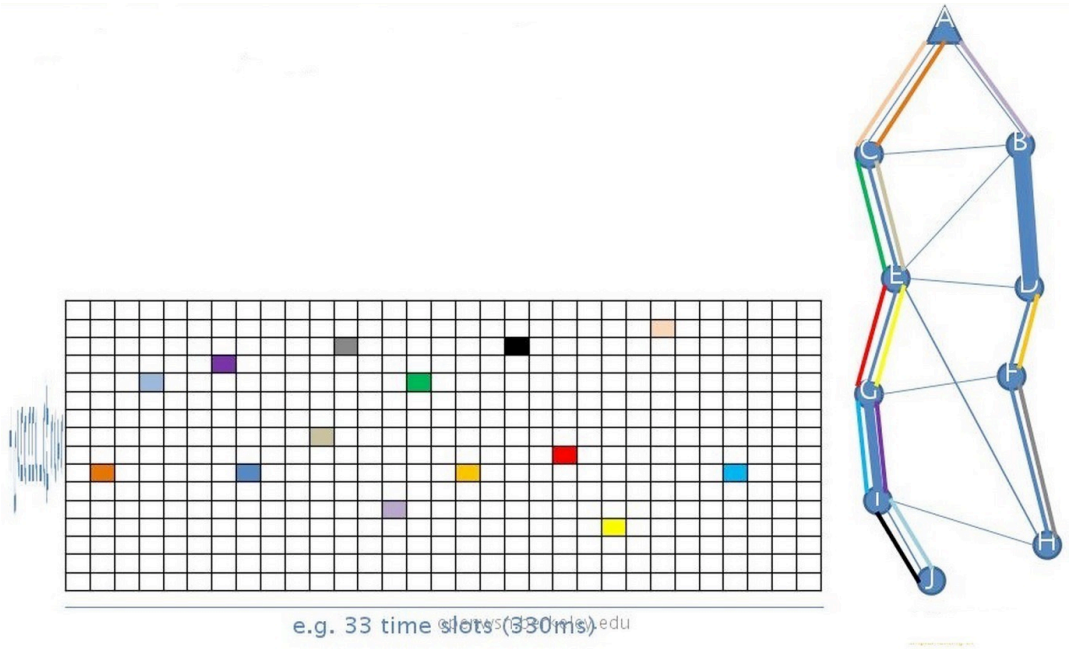
1. *Ncoap-core* : modulo principale.
2. *Ncoap-simple-client* : fornisce applicazioni per la parte client.
3. *Ncoap-simple-server* : che invece riguarda la parte servitore.

3.1.8.6 OpenWSN

OpenWSN è un'implementazione open-source di un protocollo stack completo basato su IoT e che gira su un insieme di piattaforme hardware e software molto diverse tra di loro.
È stato sviluppato grazie al supporto dell'università di Berkley e supporta diversi protocolli.
Lo stack protocollare è il seguente:

Application	CoAP, HTTP
Transport	UDP, TCP
IP-routing	IETF RPL
Adaptation	IETF 6LowPAN
Medium Access	IEEE 802.15.4e
Phy	IEEE 802.15.4-2006

In particolare *IEEE 802.15.4e* è il tentativo di migliorare il protocollo IEEE 802.15.4-2006 aggiungendo funzionalità a livello MAC per supportare molti dispositivi a livello industriale. La novità principale riguarda una funzionalità chiamata Time Synchronized Channel Hopping in cui i nodi sono sincronizzati dal punto di vista temporale e successivamente inviano pacchetti su frequenze differenti utilizzando un pattern di hopping pseudo-random.



Abbiamo poi una struttura organizzata in slot dove sono presenti superframe composti da una matrice di celle (il cui numero è modificabile in base all'applicazione da sviluppare) e ogni cella può essere assegnata ad una coppia di nodi in qualsiasi direzione si voglia comunicare. *IETF RPL* (Routing Over Low power and Lossy networks) è un protocollo di routing progettato per le reti mesh. Implementa un gradient routing in cui un nodo acquisisce un voto basato sulla distanza da nodi che collezionano questi punteggi (chiamati anche LPR ovvero Low power and lossy network border Router). Successivamente i messaggi seguiranno il gradiente calcolato per raggiungere la destinazione.

All'interno di OpenWSN è presente una libreria CoAP sviluppata in Python.
L'implementazione non supporta tutti i draft disponibili per le funzionalità aggiuntive specificate oltre a quelle di base. Infatti ci sono diversi aspetti, anche importanti dal punto di vista delle WSN, che ancora non sono stati implementati.
Iniziamo con l'elenco delle funzionalità disponibili: server e client CoAP, messaggi CON e NON, supporto al timeout e al retry e richieste concorrenti.
Invece non sono disponibili: caching, proxying, DTLS e multicast.
Per quanto riguarda invece le opzioni disponibili abbiamo: *Uri-Path* e *Content-Format*.
Altre opzioni importanti, descritte in questo documento, invece non sono minimamente supportate *If-Match, Uri-Host, ETag, If-None-Match, Uri-Port, Location-Path, Max-Age, Uri-Query, Accept, Location-Query, Proxy-Uri, Proxy-Scheme, Size1*.

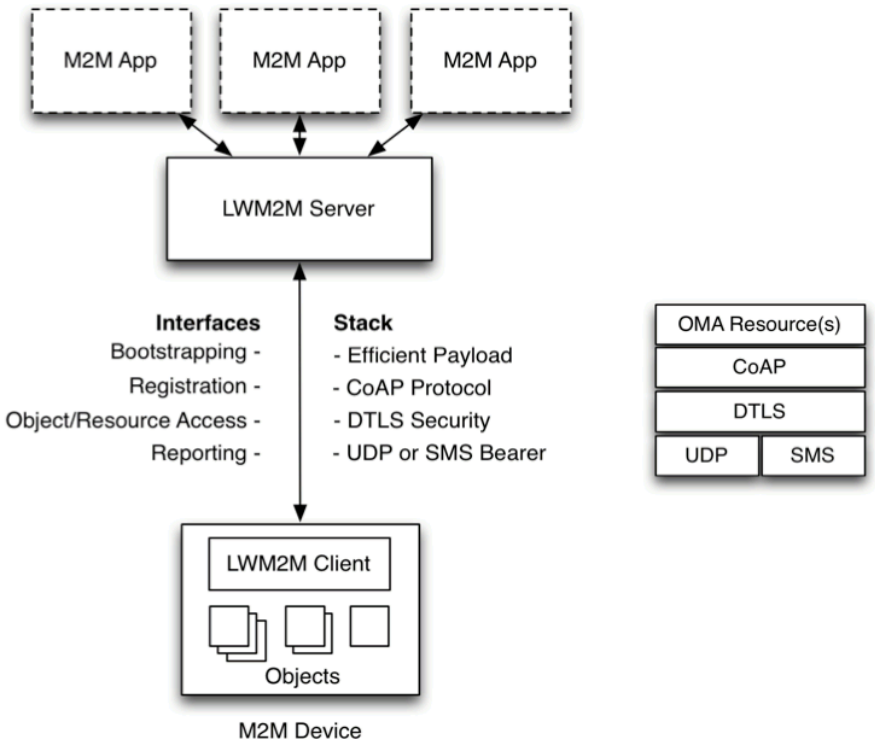
3.1.8.7 Sensinode

È un sistema di comunicazione M2M ottimizzato per dispositivi a capacità limitata che sfrutta il protocollo standard OMA Lightweight M2M. Le caratteristiche di LWM2M sono:

- Protocollo semplice ed efficiente con interfacce e payload standardizzati
- Classi di sicurezza basate su DTLS
- Modello molto potente per la gestione di oggetti e risorse
- Applicabile a reti cellulari, 6LowPAN, WiFi e ZigBee

3.1.8.7.1 Architettura

L'architettura è composta in questo modo (client-server-app):



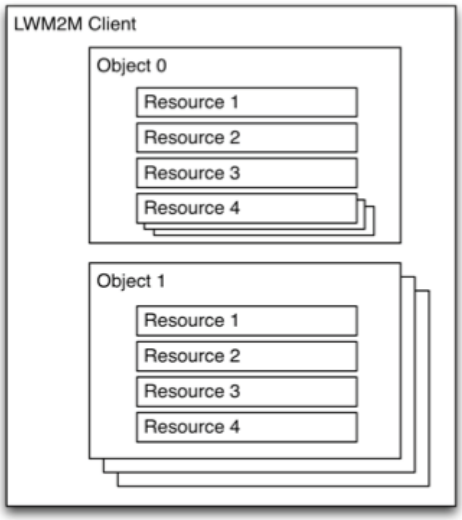
Sensinode utilizza CoAP e DTLS con il binding o su UDP o su SMS.
Le interfacce riguardano:

- *Bootstrapping* : per operazioni di provisioning o inizializzazione di client e server
- *Registration* : per operazioni di registrazione di un client e dei suoi object
- *Management & Service* : per l'accesso del server agli oggetti e alle risorse
- *Information Reporting* : per le notifiche di nuovi valori per le risorse

I payload possono essere sia nel formato testo che nel formato binario JSON.

3.1.8.7.2 Object Model

Un client ha una o più istanze di oggetti. Ogni oggetto a sua volta è una collezione di risorse. Una risorsa è un'entità atomica che può essere letta, scritta o eseguita. Le risorse possono avere istanze multiple.



Gli oggetti e le risorse sono identificati da un intero a 16 bit, mentre le stanze da un intero a che 8 bit.
La sintassi per l'accesso agli oggetti e alle risorse è la seguente:

```
/{Object ID}/{Object Instance}/{Resource ID}
```

Le specifiche tecniche definiscono 6 tipi di oggetti diversi:

Object Name	ID	Multiple Instances?	Description
LWM2M Server	1	Yes	This LWM2M objects provides the data related to a LWM2M server, the initial access rights, and security related data.
Access Control	2	Yes	Access Control Object is used to check whether the LWM2M Server has access right for performing an operation.
Device	3	No	This LWM2M Object provides a range of device related information which can be queried by the LWM2M Server, and a device reboot and factory reset function.
Connectivity Monitoring	4	No	This LWM2M objects enables monitoring of parameters related to network connectivity.
Firmware	5	No	This Object includes installing firmware package, updating firmware, and performing actions after updating firmware.
Location	6	No	The GPS location of the device.

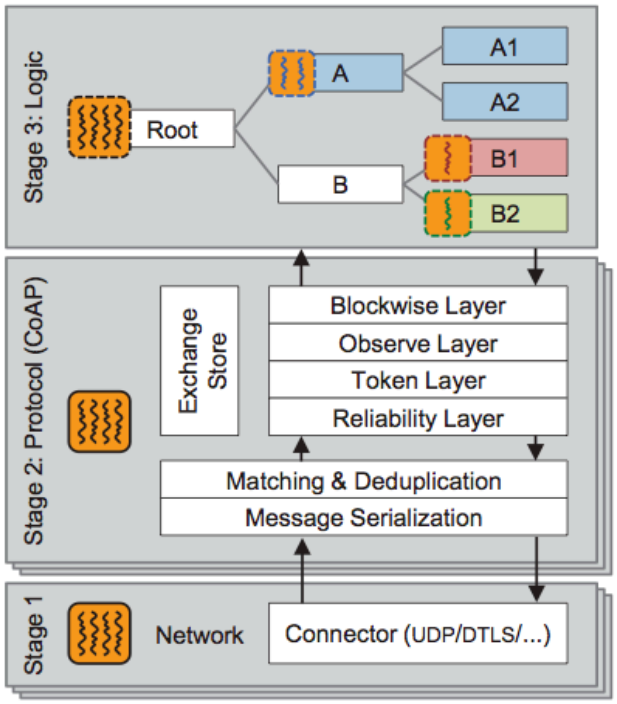
3.1.8.8 Californium

Californium è un framework CoAP molto potente che fornisce API per Web Service RESTful e implementa quasi tutte le funzionalità di CoAP. Le caratteristiche supportate sono:

- CoAP base (RFC 7252)
- Observe (draft 14)
- Trasferimento a blocchi (draft 14)
- Resource directory (draft 00)
- DTLS 1.2 (RFC 6347)
- Proxy CoAP-HTTP

3.1.8.8.1 Architettura

Californium è basato su SEDA (Staged Event-Driven Architecture) e su PIPELINED.
SEDA non fa altro che dividere il processo di gestione dei messaggi in stadi multipli. Ogni stadio consiste di una coda di eventi in arrivo, un pool di thread e un gestore di eventi che seguono la logica dello stadio. I thread non fanno altro che prendere gli eventi dalla coda e invocare l'handler apposito che può effettuare il dispatch di nuovi eventi allo stadio successivo attraverso le loro code. Ogni stadio è basato su un controller che può cambiare dinamicamente la configurazione in base a politiche configurabili. Inoltre il server SEDA può auto-configurarsi per ottimizzare il numero di thread.
PIPELINED può essere considerato una variante di SEDA dove ogni stadio ha solo un thread. Dato che i thread appartengono allo stesso pool anche le operazioni di caching sono facilitate. Normalmente un thread è associato ad un singolo core.
L'architettura è rappresentata in questo schema:



1. *Network stage* : responsabile per la ricezione e l'invio di array di byte sulla rete. Quindi non fa altro che astrarre il protocollo di trasporto.
2. *Protocol stage* : esegue il protocollo CoAP. È prevista la separazione tra book-keeping degli scambi ed elaborazione. Un oggetto Exchange contiene tutti i dati e i timer necessari per lo scambio di richieste e di risposte. Quando arriva un messaggio viene fatta un'operazione di match sull'Exchange store per recuperare lo stato necessario oppure per creare un nuovo scambio. Successivamente si ha il passaggio dello scambio con il messaggio, strato per strato e una volta che lo scambio è stato completato il sistema rimuove il suo riferimento e il garbage collector automaticamente recupera la memoria.
3. *Business logic stage* : E il livello in cui i server fanno l'host di risorse Web che sono strutturate in un albero logico e i client possono effettuare richieste sincrone e asincrone. Le chiamate sincrone inviano la richiesta al protocol stage e si bloccano fino a che non è stata restituita una risposta dallo strato sottostante. Le chiamate asincrone fanno il return immediato e riceveranno la risposta successivamente. Sarà presente un handler, registrato appositamente per questo scambio, che andrà a gestire il risultato.

3.1.9. Approfondimento

3.1.9.1 CoRE Link-Format

CoRE (Constrained RESTful Environments) permette di realizzare l'architettura REST in una forma gestibile da nodi e reti a capacità limitata.
La scoperta di risorse fornite da un server Web HTTP è chiamata solitamente Web Discovery e la descrizione della relazione tra le risorse è chiamata Web Linking. Lo scopo finale quindi è quello di fornire URI per le risorse gestite dal server complementate da attributi su quelle risorse e possibili relazioni e collegamenti con ulteriori risorse.
Il CoRE Link-Format è inserito come payload e gli viene assegnato un Internet Media Type.
Abbiamo un URI predefinita che è "/*.well-known/core*" che ci permette di avere un entry point predefinito per la richiesta della lista dei collegamenti sulle risorse gestite da un server (e quindi effettuare la discovery di risorse CoRE).

3.1.9.1.1 Discovery

Per fare discovery delle risorse su un server non bisogna far altro che inviare un messaggio GET "/*.well-known/core*" e il server restituirà un payload nel formato Core Link-Format.
Il client poi successivamente andrà a fare un'operazione di match con il Resource Type appropriato, la descrizione dell'interfaccia e i tipi possibili di Media Type (RFC2045) per la sua applicazione.
È anche possibile utilizzare directory per la memorizzazione dei collegamenti alle varie risorse. Sono infatti presenti delle vere proprie entità che hanno al loro interno una gerarchia di altre entità e memorizzano i collegamenti alle risorse presenti sugli altri server.

3.1.9.1.2 Format

Il CoRE Link Format è in formato UTF-8. Ogni link descrive un URI all'interno di parentesi angolari (< … >).
Se è presente un parametro *hosts* indica che l'URI target è una risorsa gestita dal server.
Il parametro *anchor* invece è utilizzato per descrivere le relazioni tra due risorse.
Altri attributi utilizzati dal formato CoRE Link sono:

- *rt* (resource type) : è una stringa opaca utilizzata per assegnare un tipo di semantica

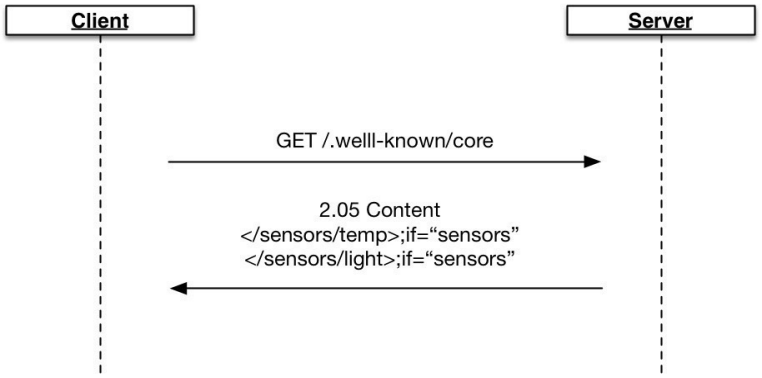
dipendente dall'applicazione a una risorsa.

- *if* (interface description) : è una stringa opaca utilizzata per fornire un nome o un URI che indica un'interfaccia specifica utilizzata per interagire con la risorsa target e può essere riutilizzata da diversi *rt*.
- *sz* (maximum size estimate) : fornisce un'indicazione della dimensione massima della rappresentazione della risorsa restituita da un'operazione di GET su un URI target. Non dovrebbe essere utilizzato per risorse molto piccole che possono rientrare in una singola MTU e comunque non è definito un limite superiore.

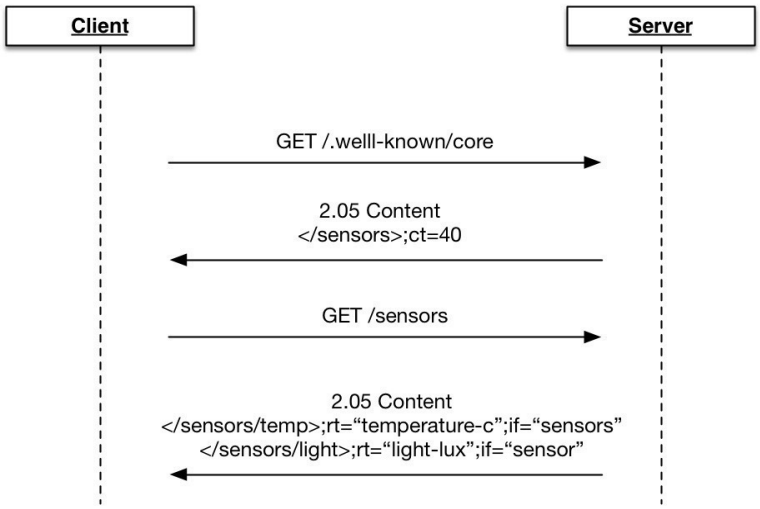
Per quanto riguarda invece le risorse disponibile al path /*.well-known*/, dipende tutto dall'applicazione l'organizzazione dei collegamenti all'interno di questo percorso. È possibile comunque effettuare operazioni di filtering per quanto riguarda i vari attributi.
Nel caso di nodi dalla capacità limitata possono anche non supportarlo e devono semplicemente ignorare la stringa di filtering e restituire l'intera risorsa.

3.1.9.1.3 Esempi

Questo esempio collega due differenti sensori che condividono la stessa descrizione dell'interfaccia (*if=“sensors”*):



Dispone la descrizione dei collegamenti in modo gerarchico e include un collegamento ad una sottorisorsa che contiene i collegamenti sui sensori.
Un esempio invece che riguarda i filtri potrebbe essere questo:



3.1.9.1.4 Considerazioni sulla sicurezza

La risorsa /*.well-known/core* potrebbe essere protetta utilizzando DTLS quando viene fatto l'hosting su un server. Si potrebbe fornire il discovery delle risorse su differenti livelli. Alcuni nodi potrebbero effettuare la scrittura di un particolare stato sulla risorsa mentre tutti gli altri hanno solo il permesso di lettura.
Le richieste in multicast sulle risorse *well-known* possono essere utilizzate per effettuare attacchi di tipo DoS su dispositivi di capacità limitata. Per ovviare a questo problema una richiesta multicast deve essere sufficientemente autenticata e messa in sicurezza (utilizzando ad esempio IPSec).

3.2. MQTT

MQTT è l'acronimo di MQ Telemetry Transport ed è un protocollo di messaging leggero, basato sull'utilizzo di broker secondo il modello publish-subscribe.
Queste caratteristiche lo rendono adatto all'utilizzo in ambienti constrained dove ad esempio abbiamo una gestione della rete molto costosa, banda limitata, la comunicazione non è affidabile oppure ci sono dispositivi con processori e risorse di memoria limitate.
Queste invece sono le caratteristiche principali:

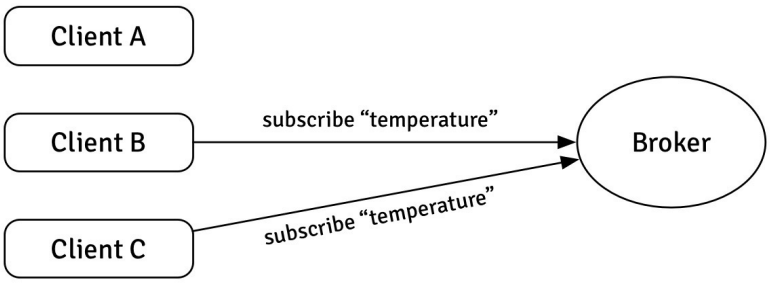
- Pattern di messaggistica publish-subscribe che permette la distribuzione uno a molti dei messaggi e il disaccoppiamento delle applicazioni.
- Il livello di trasporto per il messaging non considera il contenuto del payload.
- L'utilizzo di TCP-IP permette di ottenere connettività di rete di base.
- Abbiamo tre tipi di QoS per la delivery dei messaggi:
    1. *At most once* : i messaggi sono consegnati secondo la modalità best effort della rete TCP-IP sottostante. Può esserci sia duplicazione che perdita di messaggi. Questa modalità può essere usata con sensori di ambiente dove non importa se una lettura venga persa dato che poi ne verrà pubblicata una subito dopo.
    2. *At least once* : si ha la consegna garantita del messaggio ma si potrebbe avere un'eventuale duplicazione.
    3. *Exactly once* : si ha la garanzia che i messaggi arrivino esattamente una volta sola. Questa modalità può essere utilizzata in quei casi in cui dobbiamo effettuare operazioni di billing ed è importante che non ci siano ne duplicati e ne perdite di messaggi.
- Un overhead a livello di trasporto molto basso con un header di soli 2 byte e scambi di protocollo minimizzati per ridurre il traffico di rete.
- Un meccanismo per notificare le parti interessate di una disconnessione anormale di un client, utilizzando la caratteristica Last Will and Testament.

Nell'ultima versione disponibile (3.1) sono state introdotte alcune funzionalità e ottimizzazioni come la possibilità di inviare username e password all'interno di un pacchetto CONNECT, miglioramenti per la sicurezza con codici restituiti dall'invio di pacchetti CONNACK e il supporto per UTF-8 al posto del sottoinsieme US-ASCII.

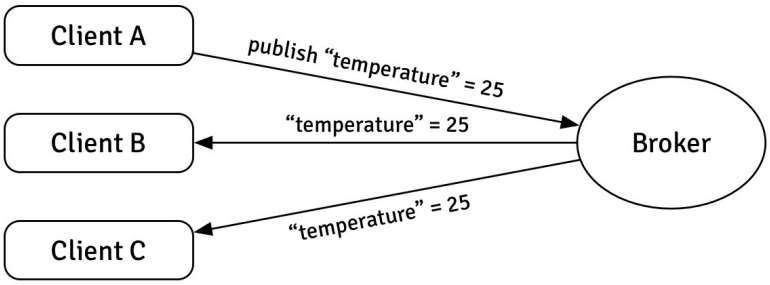
3.2.1. Architettura

MQTT è basato sul modello publish-subscribe. È message oriented e ogni messaggio è pubblicato su un indirizzo (il topic). I client possono comunque sottoscriversi a topic multipli e successivamente ognuno riceverà tutti i messaggi diretti verso il topic scelto.

Ad esempio consideriamo il caso in cui abbiamo 3 client (A, B e C) che hanno connessioni TCP aperte con il broker MQTT. B e C si sottoscrivono al topic “temperature”:



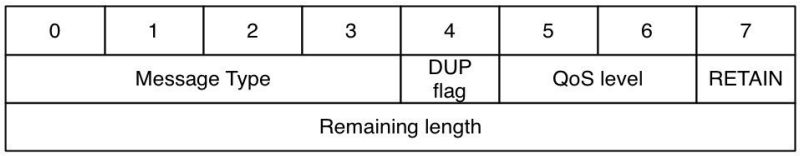
Successivamente A pubblica il valore 25 sul topic. Il broker allora effettua il forwarding dei messaggi ai client sottoscritti.



3.2.2. Messaggi

3.2.2.1 Header fisso

L’header dei messaggi è fisso e lo schema è il seguente:



I campi sono:

- *Message Type* : 4 bit
  - *Reserved* (0) : riservato
  - *CONNECT* (1) : un client richiede la connessione ad un server
  - *CONNACK* (2) : ACK sulla CONNECT
  - *PUBLISH* (3) : pubblica un messaggio
  - *PUBACK* (4) : ACK sulla PUBLISH del messaggio
  - *PUBREC* (5) : messaggio PUBLISH ricevuto
  - *PUBREL* (6) : messaggio PUBLISH rilasciato
  - *PUBCOMP* (7) : messaggio di PUBLISH completato
  - *SUBSCRIBE* (8) : richiesta di SUBSCRIBE dal client
  - *SUBACK* (9) : ACK della richiesta di SUBSCRIBE
  - *UNSUBSCRIBE* (10) : richiesta di UNSUBSCRIBE dal client
  - *UNSUBACK* (11) : ACK della richiesta di UNSUBSCRIBE
  - *PINGREQ* (12) : richiesta di PING
  - *PINGRESP* (13) : risposta al PING
  - *DISCONNECT* (14) : indica che il client si vuole disconnettere
  - *RESERVED* (15) : riservato
- *DUP* : flag utilizzato quando il client o il server vuole ritrasmettere un messaggio PUBLISH, PUBREL, SUBSCRIBE o UNSUBSCRIBE. Si applica quando abbiamo un valore di QoS level maggiore di zero e quando è impostato ad 1 allora nell'header variabile abbiamo un Message ID.
- *QoS level* : indica il livello di qualità nella delivery dei messaggi
  - 0 : At Most Once : Fire and Forget
  - 1 : At Least Once : Acknowledge delivery
  - 2 : Exactly once : Assured delivery
  - 3 : riservato
- *RETAIN* : flag utilizzato solo per i messaggi PUBLISH. Quando un client invia un messaggio PUBLISH e RETAIN è impostato ad 1 allora il server dovrebbe mantenere

in memoria i messaggi che sono stati inviati ai subscriber correnti.

- Quando c'è una nuova sottoscrizione ad un topic si dovrebbe inviare l'ultimo messaggio retained al subscriber. Si permette, così, di ricevere l'ultimo valore conosciuto di una variabile.
- Quando un server invia una PUBLISH ad un client come risultato della sottoscrizione che esisteva già quando la PUBLISH originale è arrivata non si deve impostare il flag RETAIN a uno per permette al client di sapere qual'è il valore che era stato mantenuto in memoria e qual'è il valore che invece è stato inviato al momento.
- I messaggi retained devono sopravvivere al riavvio del server.

- *Remaining length* : rappresenta il numero di byte che rimangono all'interno del messaggio corrente (riguardo l'header variabile e il payload).
  - Messaggi più piccoli di 127 byte : si utilizza il byte di Remaining length
  - Messaggi più grandi di 127 byte : si utilizzano 7 bit di un byte per indicare il remaining length e l'ottavo indica se ci sono byte successivi da tenere conto. Il numero di byte successivi massimo è di 4. Si riesce ad ottenere così un messaggio grande al massimo 256 MB.

3.2.2.2 Variable header

È poi possibile avere un header variabile con i seguenti campi (in ordine di apparizione):

- *Protocol name* : nome del protocollo di un messaggio MQTT CONNECT codificato in UTF.
- *Protocol version* (8 bit) : versione del protocollo di un messaggio MQTT CONNECT

0	1	2	3	4	5	6	7
Protocol version							

- *Flag per messaggi CONNECT*

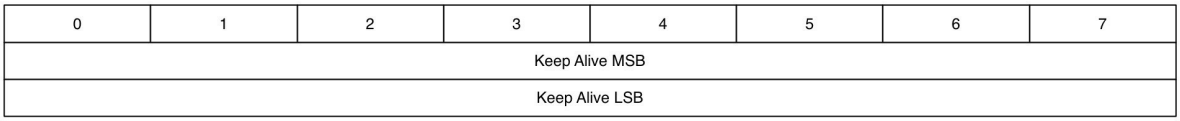
0	1	2	3	4	5	6	7
User Name flag	Password flag	Will retain	Will QoS		Will flag	Clean session	Reserved

  - *Clean session* : se non è impostato allora il server deve conservare le sottoscrizioni dei client dopo che essi si disconnettono. Il server deve anche

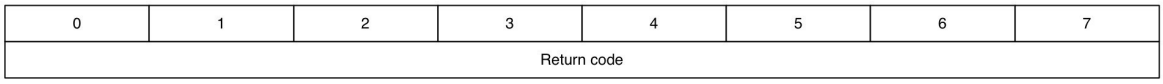
memorizzare i messaggi dal momento della disconnessione. Se invece è impostato ad uno allora il server deve cancellare ogni informazione memorizzata precedentemente.

- *Will* : definisce che un messaggio è pubblicato per conto di un client dal server quando o si incontra un errore di I-O dal server durante la comunicazione con il client oppure il client fallisce la comunicazione all'interno dell'intervallo di tempo *Keep Alive*. Se è impostato ad uno allora devono essere presenti i campi *Will QoS*, *Will Retain* e in aggiunta *Will Topic* e *Will Message* nel payload.
- *Will QoS* : livello di QoS per un messaggio Will che è inviato nel caso in cui il client si disconnetta involontariamente.
- *Will retain* : indica se il server deve memorizzare il messaggio Will pubblicato dal server per conto del client che si è disconnesso in modo imprevisto.
- *User Name e Password* : indica che nel payload sono presenti username e password per un messaggio CONNECT. Non è possibile inserire una password senza che ci sia l'username.
- *Reserved* : riservato

- *Keep-Alive timer* (16 bit) : si trova nell'header variabile di un messaggio CONNECT



  - Si misura in secondi e il valore massimo è di 18 ore.
  - Definisce l'intervallo massimo tra i messaggi ricevuti da un client.
  - Dopo questo intervallo di tempo il client viene riconosciuto come disconnesso dalla rete senza dover attendere il timeout TCP-IP.
  - Per rimanere in linea il client deve inviare un messaggio PINGREQ a cui segue una risposta PINGRESP da parte del server.
  - Se il server non riceve un messaggio dal client dopo 1 intervallo e mezzo di Keep Alive allora disconnette il client come se avesse inviato un messaggio DISCONNECT.
- *Connect return code* : codice che viene restituito nel variable header di un messaggio CONNACK.



- 0 : connessione accettata
- 1 : connessione rifiutata per versione di protocollo non accettata
- 2 : connessione rifiutata per identificatore respinto
- 3 : connessione rifiutata per server non disponibile
- 4 : connessione rifiutata per nome utente e password errati
- 5 : connessione rifiutata per mancanza di autorizzazione
- 6-255 : codici riservati e non ancora in uso

• *Topic name* : codice che si trova nel variable header di un messaggio PUBLISH
  - È la chiave che identifica il canale di informazione dove deve essere pubblicato il payload del messaggio (duale per i subscribers).
  - Codificato in UTF.

3.2.2.3 Payload

I messaggi che fanno uso del payload sono:

1. *CONNECT* : il payload contiene una o più stringhe UTF-8. Viene specificato un identificatore unico per il client, un Will Topic, un Will Message e nome utente e password da usare.
2. *SUBSCRIBE* : il payload contiene una lista di nomi di Topic dove il client può sottoscriversi e il livello di QoS.
3. *SUBACK* : il payload contiene una lista di livelli QoS. Questi livelli sono quelli che gli amministratori del server consentono sulle loro macchine.
4. *PUBLISH* : il payload contiene solo dati application-specific.

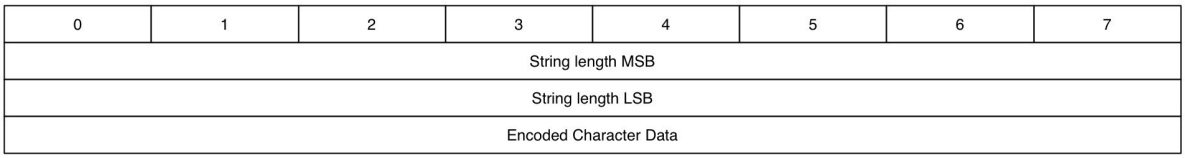
3.2.2.4 Identificatori di messaggio

Sono presenti in tutti i tipi di messaggio che possono essere inviati attraverso il protocollo MQTT.
Il *Message ID* (16 bit) è presente solo nei messaggi che presentano un livello QoS uguale ad 1 o 2 nel fixed header. Deve essere unico in riferimento ai messaggi in una particolare direzione

di comunicazione. Infatti un client deve mantenere due insiemi di Message ID e può inviare un messaggio con un determinato identificatore e riceverne uno con lo stesso ID e li considera come due messaggi totalmente differenti.

3.2.2.5 UTF-8

È un encoding efficiente di stringhe che ottimizza la rappresentazione di caratteri ASCII per il supporto alle comunicazioni text-based. Lo schema di rappresentazione è:



- *String length* : numero di byte dei caratteri codificati e non il numero di caratteri.
- *Encoded Character Data* : dati da codificare.

I bit non utilizzati devono essere invece posti a zero.

3.2.3. Topic matching

In MQTT i topic sono gerarchici. Sono permessi i wildcard (caratteri che permettono di selezionare più entità alla volta) nel momento in cui ci si registra ad un topic in modo tale da poter fare l'Observe di intere gerarchie.
Il wildcard + ci permette di fare il matching con una singola directory.
Il wildcard # invece permette di selezionare un numero di directory indefinite.
Ad esempio se ci iscriviamo a *"cucina/+/temperatura"* ci iscriviamo a *"cucina/piano_cottura/temperatura"* ma non a *"cucina/piano_cottura/forno/temperatura"*.

3.2.4. Application Level QoS

3.2.4.1 Livelli di QoS

MQTT permette di inviare messaggi secondo un livello di QoS (Quality of Service):

1. *Livello 0* : consegna At Most Once

- Il messaggio viene consegnato secondo la strategia best effort di TCP-IP
- La risposta arriva al server oppure può anche non arrivare
- Utilizzato nelle PUBLISH

2. *Livello 1* : consegna At Least Once
   - Viene effettuato l'ACK di un messaggio tramite un PUBACK
   - Se l'ACK non arriva a destinazione dopo un determinato periodo di tempo allora il mittente re-invia il messaggio con impostato il flag DUP.
   - Livello di QoS utilizzato da PUBLISH, SUBSCRIBE e UNSUBSCRIBE. La semantica della PUBLISH è: memorizza il messaggio, invia il messaggio PUBLISH ai subscribers e poi cancella il messaggio.
   - Un messaggio di questo tipo ha un Message ID nell'header.
   - Quando il server riceve un duplicato reinvia il messaggio al subscriber e invia un altro messaggio PUBACK.
3. *Livello 2* : consegna Exactly Once
   - Assicura che non ci siano duplicati inviati all'applicazione ricevente o al subscriber.
   - È presente un aumento del traffico di rete ma è di solito accettabile per l'importanza del contenuto.
   - Livello di QoS utilizzato da PUBLISH e PUBREL. La semantica di PUBLISH è memorizza il messaggio, il Message ID e pubblica il messaggio ai subscriber. La semantica di PUBREL è: pubblica il messaggio ai subscriber e cancella il messaggio oppure cancella solo il Message ID.
   - Un messaggio di questo tipo ha un Message ID nell'header.
   - In caso di failure verrà re-inviato l'ultimo messaggio non ACKed.

3.2.4.2 Ordine dei messaggi

L'ordine dei messaggi può essere affetto da un numero elevato di fattori incluso il numero di flussi verso un determinato client oppure se il client è single o multi thread.
Per avere garanzia che ci sia un qualche ordine nei messaggi bisogna che i flussi di consegna dei messaggi siano completati secondo l'ordine di partenza. Il numero di messaggi simultanei ha comunque effetto sul tipo di garanzia che possiamo avere:

1. Quando abbiamo una finestra di simultaneità uguale a uno ogni flusso deve essere

completato prima che inizi il successivo. Questo garantisce che i messaggi siano consegnati nell'ordine in cui sono stati inviati.

2. Quando invece abbiamo una finestra maggiore di uno allora l'ordine dei messaggi può essere garantito solo a livello di QoS.

3.2.5. Last Will And Testament

In MQTT abbiamo la possibilità di registrare sul server un messaggio chiamato "Last Will and Testament" che viene inviato dal broker nel momento in cui si disconnette. Questi messaggi possono essere usati per segnalare ai subscriber che un dispositivo si è disconnesso (in modo anomalo).

3.2.6. Persistence

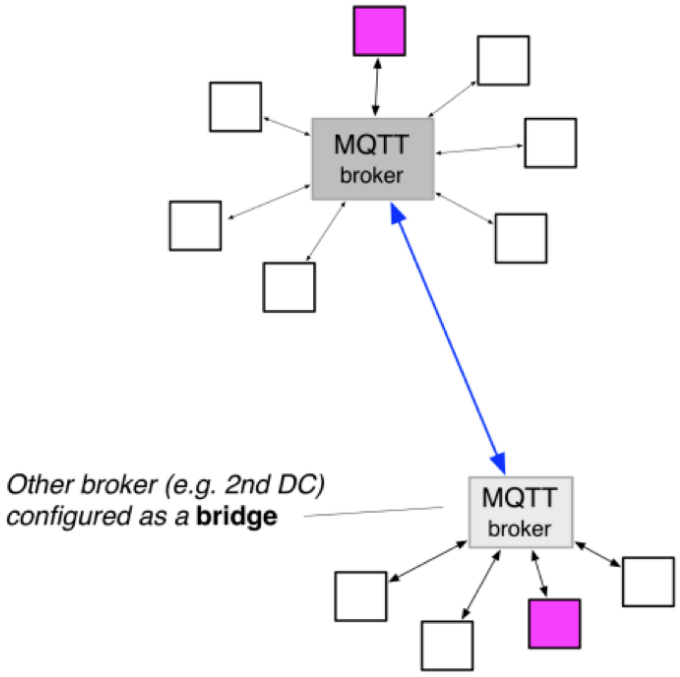
MQTT ha il supporto per i messaggi persistenti che vengono memorizzati sul broker. Nel momento in cui vengono pubblicati i messaggi, i client possono richiedere al broker di fare la persist (solo l'ultimo messaggio viene memorizzato). Quando un client si sottoscrive ad un topic, allora riceverà l'ultimo messaggio che ha ricevuto il broker.
Rispetto alle normali code i broker MQTT non permettono la persistenza di messaggi all'interno del server.

3.2.7. Broker MQTT

3.2.7.1 Mosquitto

Mosquitto è un broker MQTT open-source, che supporta TLS e permette l'autenticazione tramite nome utente e password, chiavi precondivise o certificati TLS. Mosquitto ha una semplice ACL che l'amministratore può configurare per specificare quali client possono accedere ai topic.
I client possono opzionalmente impostare un messaggio Will che viene poi pubblicato dal broker quando un client si disconnette in modo inaspettato. In altre parole se un client invia una richiesta Will prima di connettersi, e ad un certo punto cade, il broker pubblicherà il

payload per conto del client.
Mosquitto può anche essere configurato come bridge. Un bridge può collegarsi ad un altro broker come qualsiasi altro client e sottoscriversi a topic che sono poi importati all'interno del bridge. I client che successivamente si collegano questo bridge possono essere notificati di specifiche sottoscrizioni dal broker principale. Lo schema è questo:



Un'altra particolarità di Mosquitto è che può pubblicare statistiche a cui i client possono sottoscriversi ad esempio per ragioni di monitoring.

3.2.8. MQTT-SN

Anche se MQTT-SN è stato progettato per essere leggero, ha due limiti forti nel caso venga utilizzato su dispositivi a capacità limitata.
Ogni client MQTT deve supportare TCP e deve mantenere una connessione aperta verso il broker per tutto il tempo necessario. Negli ambienti citati sopra dove c'è una grande probabilità di perdita di pacchetti questo è un problema.
I nomi dei topic sono spesso stringhe lunghe che non possono essere utilizzate su 802.15.4. MQTT-SN invece non presenta questi problemi in quanto definisce un mapping UDP verso MQTT e aggiunge il supporto ai broker per i Topic Name indicizzati.

3.2.8.1 Introduzione

Una rete composta da sensori ed attuatori molte volte va incontro a situazioni in cui i collegamenti wireless possono venire meno oppure i dispositivi a loro volta possono danneggiarsi e non funzionare più. In questi casi far riferimento agli indirizzi dei nodi per comunicare rende la gestione della rete molto difficile.
Quindi bisognerebbe utilizzare un altro approccio alla comunicazione centrato sui dati, nel quale l'informazione è inviata ai ricevitori non in base al loro indirizzo di rete ma come una funzione del contenuto e dell'interesse verso l'argomento di quel dato. Un esempio è quello del pattern publish-subscribe che viene anche utilizzato da MQTT. Lo svantaggio principale di MQTT è che ha bisogno di TCP che è troppo complesso per dispositivi molto semplici e a basso costo come i sensori.
Per risolvere questo problema è stato creato MQTT-SN che è una versione di MQTT adattata per questo tipo di reti. Le WSN basate su IEEE 802.15.4 hanno infatti le seguenti caratteristiche principali:

- Forniscono una banda massima di 250 kbit/s sulla banda da 2.4 GHz.
- I pacchetti hanno una lunghezza molto corta e infatti sono limitati (a livello fisico) a 128 byte di cui la metà è spesa per l'overhead, per il supporto al livello MAC e per la sicurezza.

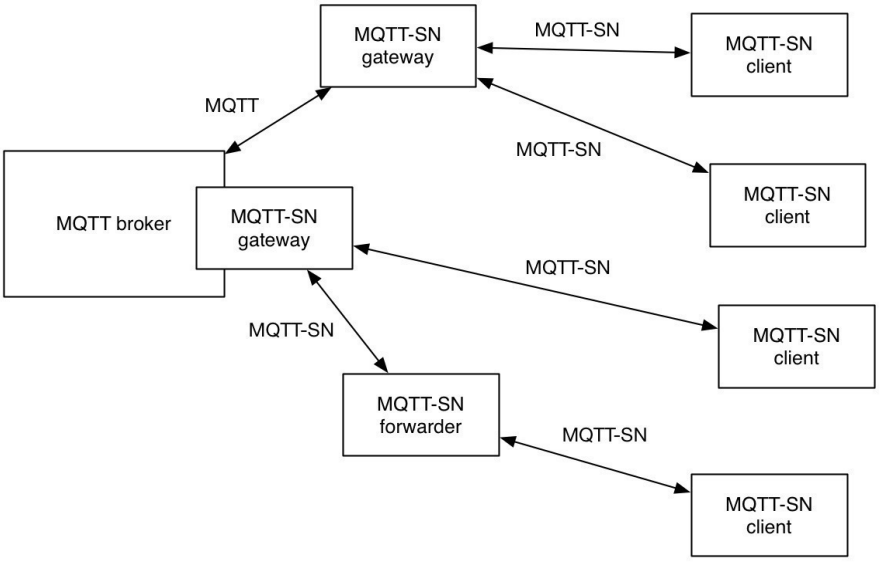
3.2.8.2 Differenze con MQTT

1. Il messaggio CONNECT è suddiviso in tre messaggi. I due addizionali sono opzionali e sono usati per trasferire il *Will Topic* e il *Will Message* al server.
2. Per gestire in modo appropriato i messaggi di 802.15.4 il nome del Topic all'interno di quelli PUBLISH è rimpiazzato da un *topic id* che è lungo solo 2 byte. Viene comunque definita una procedura che permette ai client di registrare i Topic Name e ottenere il corrispondente topic id.
3. Sono stati definiti *topic id* di default e Topic Name di lunghezza corta (2 byte) per i quali non c'è bisogno di registrazione.
4. È presente una procedura di discovery che aiuta i client senza un server o gateway pre-configurato a scoprire l'indirizzo di rete di uno di essi. Possono essere presenti gateway multipli nella stessa rete wireless e possono co-operare in una modalità load-sharing o stand-by.

5. La semantica del flag *Clean Session* è estesa al flag *Will*. Infatti non solo le sottoscrizioni dei client sono persistenti ma anche i *Will Message* e i *Will Topic*. Entrambi sono modificabili dal client durante una sessione.
6. È presente una procedura di *keep-alive* non in linea che viene utilizzata per i client sleeping. Con questa procedura, infatti, i dispositivi a batteria possono entrare in uno stato di sleep durante il quale tutti i messaggi destinati a essi sono bufferizzati nel server-gateway e sono inviati successivamente quando ri-diventano awake.

3.2.8.3 Architettura

L'architettura di MQTT-SN è rappresentata in figura:



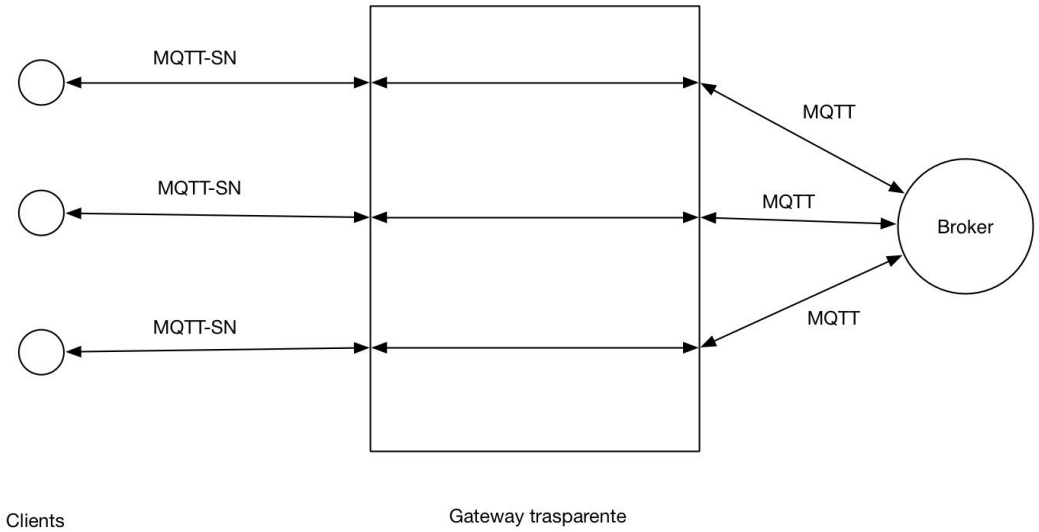
Ci sono tre tipi di componenti:

1. *MQTT-SN client* : si connettono a un server MQTT attraverso un MQTT-SN gateway utilizzando il protocollo MQTT-SN.
2. *MQTT-forwarder* : viene utilizzato quando l'MQTT-SN gateway non è direttamente attaccato alla rete del client. Non fa altro che prendere i pacchetti MQTT-SN proveniente dal client e li redirige direttamente al gateway che è collegato ad esso e viceversa.
3. *MQTT-SN gateway* : può essere o non essere integrato con un server MQTT. Nel caso non sia integrato si utilizza MQTT per la comunicazione tra le due entità. La sua funzione principale è quella di tradurre le chiamate MQTT in chiamate MQTT-SN e

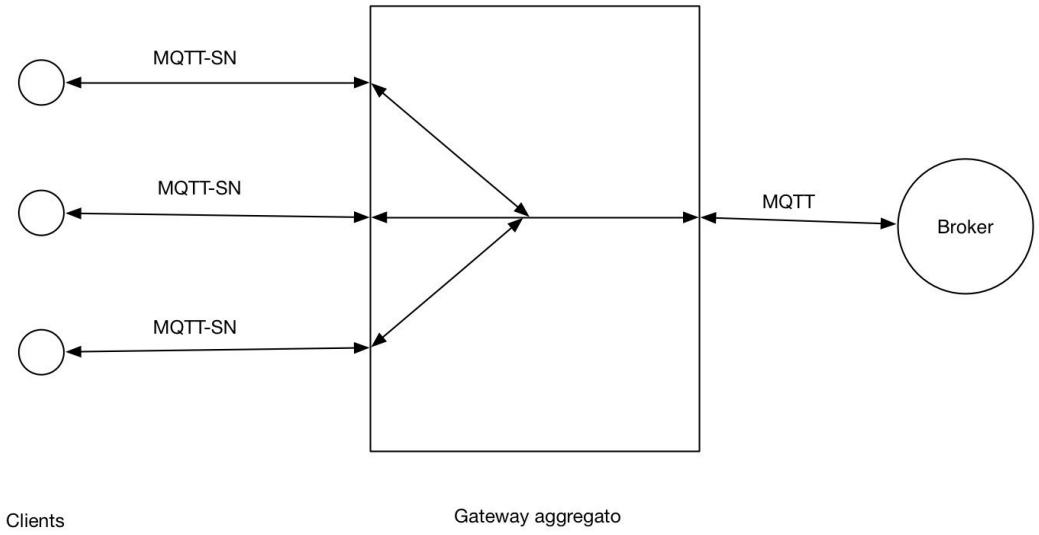
viceversa.

Ci sono 2 tipi di gateway che si differenziano per il modo in cui effettuano la traduzione del protocollo:

Il *gateway trasparente*: mantiene una connessione MQTT per ogni client connesso verso il broker. È comunque possibile limitare il numero di connessioni possibili. Effettua solo una traduzione di sintassi tra i due protocolli e tutte le funzioni che fornisce il server sono direttamente disponibili al client. Lo schema è questo:



Il *gateway aggregato*: permette di avere solo una connessione verso il server. Tutti i messaggi scambiati arrivano al gateway e sarà poi esso stesso che deciderà come dovrà essere rediretta l'informazione verso il broker. Ha un'implementazione più complessa rispetto al gateway trasparente ma può essere comunque utile nel caso di WSN molto grandi perché riduce il numero di connessioni verso il broker. Lo schema è questo:



3.2.8.4 Funzionalità

Si è cercato di progettare MQTT-SN per renderlo molto simile a MQTT. Ci sono comunque delle differenze riguardo le funzionalità disponibili:

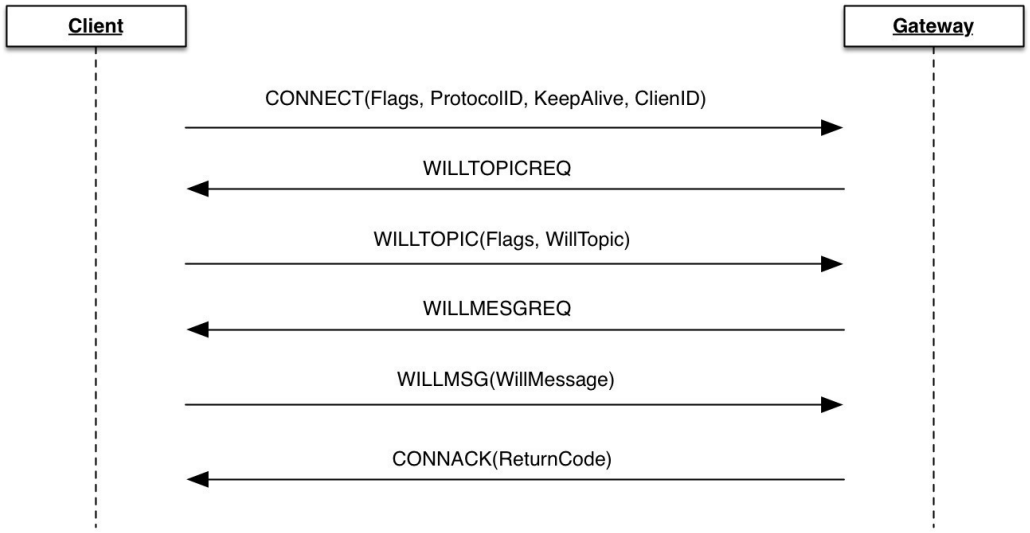
3.2.8.4.1 Gateway advertisement e discovery

Un gateway può annunciare la sua presenza (solo se connesso ad un server) effettuando il broadcast periodico di un messaggio *ADVERTISE* a tutti i componenti della rete.
Dato che possono esserci diversi gateway è il client che decide a quale connettersi. In un determinato istante comunque il client può essere connesso solo ad un gateway.
Un client mantiene una lista di gateway attivi e l'aggiorna in base al valore *TADV* presente nel messaggio *ADVERTISE* che indica quando il gateway invierà il prossimo messaggio broadcast. Se il client, passato quel periodo di tempo, non registrerà un nuovo *ADVERTISE* da parte del gateway capirà che si è disconnesso.
Dato che comunque i messaggi di *ADVERTISE* sono inviati in broadcast in una rete wireless bisogna comunque cercare di equilibrare il valore di *TADV* in modo tale da evitare congestioni di rete.
Il client può comunque effettuare il search di un gateway attraverso la *SEARCHGW*. Per evitare broadcast storm (con client multipli che cercano un gateway) il metodo viene ritardato di un valore tra 0 e *T_SEARCHGW*. Il gateway risponde successivamente con un messaggio *GWINFO* per indicare il suo ID.

In modo simile un client propaga la GWINFO ai nodi vicini in modo tale che sappiano che esiste un gateway e che quel gateway ha quel determinato ID (contenuto nella GWINFO).

3.2.8.4.2 Impostazione della connessione al client

Come in MQTT anche in MQTT-SN c'è bisogno che il client imposti una connessione ad un gateway prima di scambiare informazioni. Per crearne una viene utilizzato il messaggio CONNECT con l'impostazione del flag Will. Se è posto ad 1 allora il client si aspetta successivamente due messaggi da parte del server (WILLTOPICREQ e WILLMSGREQ) a cui risponderà con altrettanti messaggi. La procedura termina quando viene inviato il messaggio di CONNACK dal gateway.
Lo schema delle interazioni è questo:



3.2.8.4.3 Clean session

Con MQTT quando un client si disconnette le sue sottoscrizioni non vengono cancellate. Sono persistenti e valide per nuove connessioni fino a quando il client non effettua la UNSUBSCRIBE oppure si connette con il flag Clean Session impostato uguale ad 1.
In MQTT-SN si estende questo concetto anche ai Will Topic e ai Will Message. Secondo l'impostazione dei flag abbiamo:

1. *CleanSession=true* e *Will=true* : il gateway cancella sia le sottoscrizioni e sia i dati Will relativi al client e richiede nuovi Will Topic e Will Message
2. *CleanSession=true* e *Will=false* : rispetto al precedente non richiede i messaggi Will e

restituisce un messaggio CONNACK

3. *CleanSession=false* e *Will=true* : il gateway mantiene tutti i dati relativi all'utente ma richiede nuovi Will Topic e Will Message che andranno a sovrascrivere quelli precedenti.
4. *CleanSession=false* e *Will=false* : come il precedente ma non richiede nuovi messaggi WILL e restituisce una CONNACK.

3.2.8.4.4 Procedura per l'aggiornamento dei dati Will

È possibile aggiornare i dati Will (sovrascrivendoli sul gateway) attraverso un messaggio *WILLTOPICUPD* e *WILLMSGUPD*.

3.2.8.4.5 Registrazione del Topic Name

Come detto in precedenza non vengono usati i *Topic Name* ma i *topic id* per ridurre la dimensione del messaggio.
Per registrare un *Topic Name* bisogna inviare un messaggio *REGISTER* al gateway che una volta assegnato il *topic id* lo restituisce al cliente attraverso un messaggio *REGACK*. Se non è possibile assegnare un *topic id* (perché in quel momento è presente una congestione di rete) allora viene comunque restituito un messaggio *REGACK* ma con un *Return Code* che indica il problema.
Come per la SUBSCRIBE e per la UNSUBSCRIBE il cliente può avere solo un'operazione contemporaneamente.

3.2.8.4.6 Procedura di publish del client

Una volta registrato con successo un *Topic Name* con il gateway il client può inviare messaggi PUBLISH (che contiene questa volta il *topic id*) verso di esso.
Il client riceverà poi un messaggio PUBACK che può contenere:

- *ReturnCode="Rejection: invalid topic id"* : quindi deve registrare di nuovo il Topic Name.
- *ReturnCode="Rejection: congestion"* : deve smettere di inviare messaggi PUBLISH verso il gateway per un tempo T_WAIT.

3.2.8.4.7 Topic ID e Topic Name predefiniti

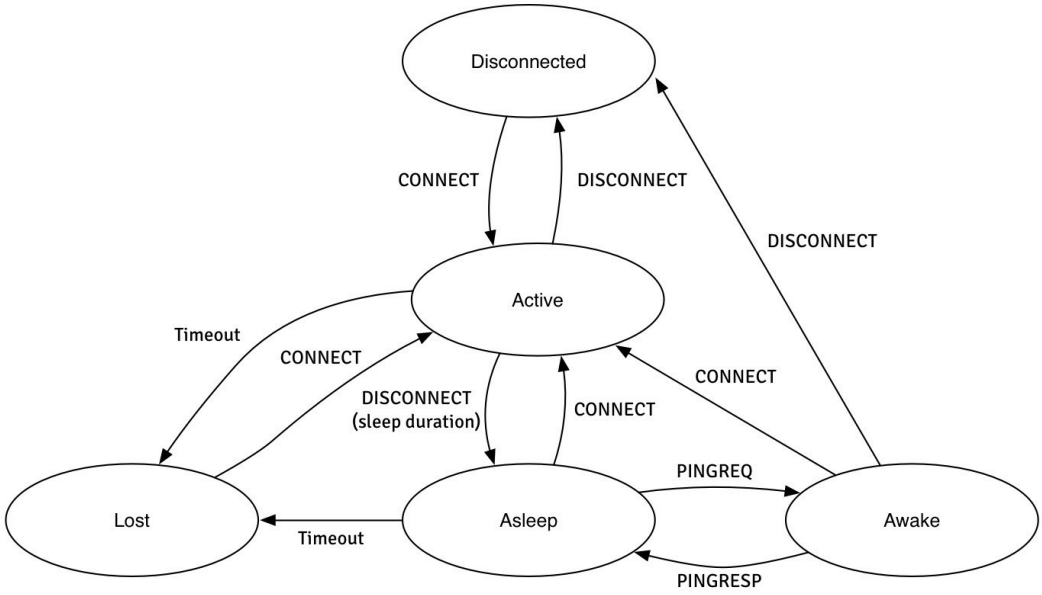
Un *Topic ID* predefinito è un *topic id* che di cui viene fatto direttamente il mapping ad un topic name in anticipo sia dall'applicazione client che dal gateway. Si può in questo modo procedere direttamente all'invio del messaggio PUBLISH senza prima utilizzare il messaggio REGISTER.
Un *Topic Name* corto è un topic name che ha una lunghezza fissa di 2 byte e può essere inserito insieme ai dati nel messaggio PUBLISH.

3.2.8.4.8 Operazione di PUBLISH con livello di QoS pari a 1

In questo tipo di PUBLISH non è presente nessun setup della connessione, registrazione o sottoscrizione. Semplicemente il client invia un messaggio PUBLISH al gateway e non si cura di verificare le condizioni per l'invio (gateway presente, indirizzo corretto o se il messaggio è arrivato a destinazione). Non bisogna far altro che impostare il flag di *QoS* a *"0b11"* e utilizzare il Topic ID predefinito o il Topic Name corto.

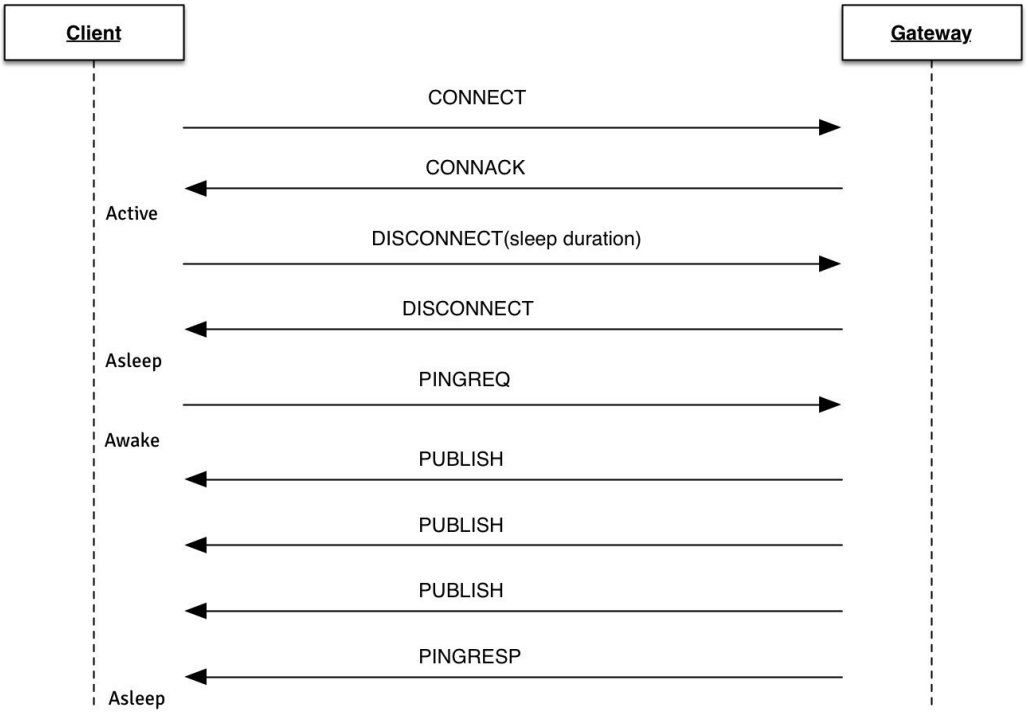
3.2.8.4.9 Supporto a client sleeping

I client sleeping non sono altro che quei dispositivi alimentati a batteria che devono risparmiare quanta più energia possibile. Il server o il gateway devono essere aware dello stato corrente del client ed effettuare il buffering dei messaggi indirizzati ad essi ed inviarli quando lo stato dei dispositivi è awake.
Questo è il diagramma dello stato dei client:



- *Active* : quando il server ha ricevuto un messaggio CONNECT dal client.
- *Lost* : quando il gateway non riceve messaggi dal client per un periodo di tempo più lungo del *Keep Alive*.
- *Disconnected* : quando il gateway riceve un messaggio DISCONNECT senza un valore per il campo di durata.
- *Asleep* : quando il gateway riceve un messaggio DISCONNECT che contiene un campo di durata dello sleep. Se poi non riceve successivamente un messaggio dal client dopo la sleep duration allora viene considerato come *Lost*

Questo è lo schema degli stati rispetto alle varie operazioni:



Se il client era in stato di sleep e invia un messaggio PINGREQ al gateway viene considerato come *Awake* e gli saranno inviati tutti i messaggi che il gateway aveva inserito nel buffer durante lo stato di sleep del client. Alla fine il gateway invierà un messaggio PINGRESP e considererà il client come *Asleep*.
Per evitare l'eccessivo scaricamento della batteria montata sul dispositivo, il client dovrebbe limitare il numero di ritrasmissioni del messaggio PINGREQ nel caso non riceva risposte dal gateway.

3.3 Confronto tra CoAP e MQTT

MQTT e CoAP sono entrambi utilissimi come protocolli all'interno di IoT e hanno delle differenze fondamentali.
MQTT è un protocollo di comunicazione molti a molti per il passaggio dei messaggi tra client multipli attraverso un broker centrale. Effettua disaccoppiamento tra produttore e consumatore dove i client richiamano la PUBLISH e lasciano che il broker decida quando effettuare il routing e la copia dei messaggi. Mentre MQTT ha qualche supporto per la persistenza, il suo punto di forza è quello di poter essere utilizzato come bus di

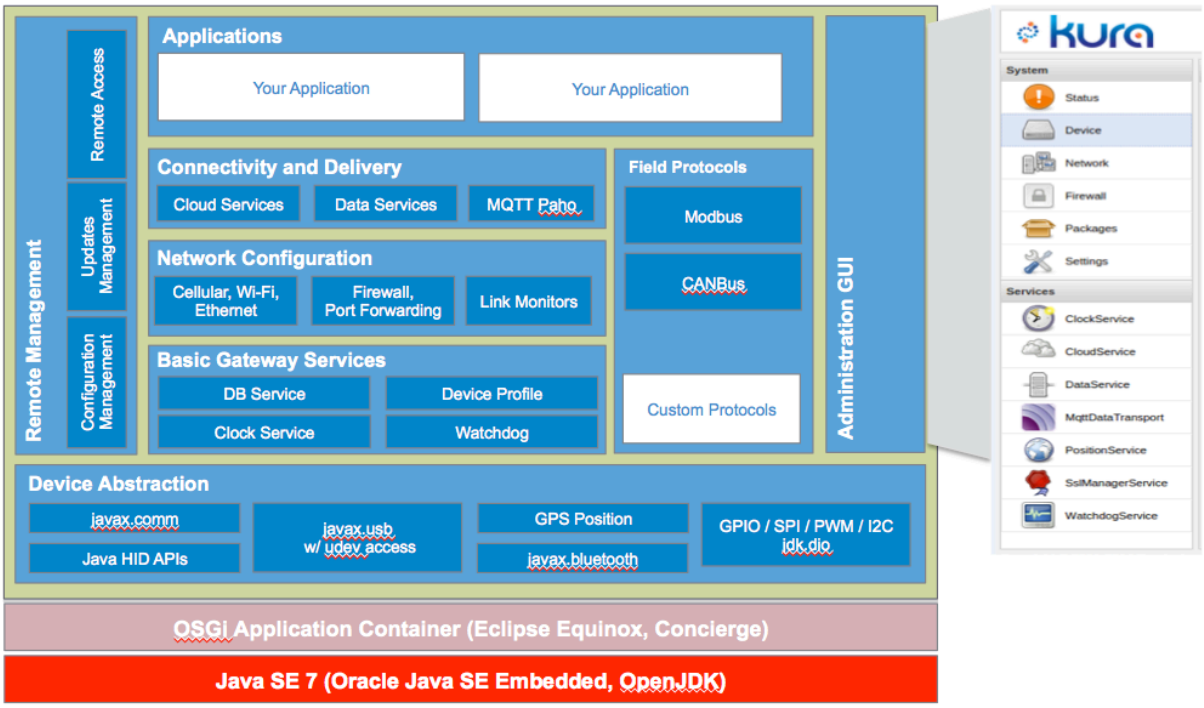
comunicazione per dati live.
CoAP è prima di tutto un protocollo per la comunicazione uno a uno per trasferire le informazioni di stato tra client e server. Il supporto per le risorse Observe non è direttamente descritto nell'RFC ufficiale ma in un draft che viene aggiornato costantemente (l'ultima versione è la numero 15 e risale a maggio 2005). CoAP rimane comunque un modello per il trasferimento di stato e non può essere ancora utilizzato in modo ottimale per la gestione degli eventi.
Mentre MQTT effettua un collegamento TCP di lunga durata verso un broker e quindi non abbiamo nessun problema per quanto riguarda i dispositivi dietro NAT, CoAP invece è strutturato in modo tale che sia client che server devono inviare e ricevere pacchetti UDP.
In un ambiente NAT, questo può essere un problema e possibili soluzioni riguardano l'utilizzo di tunnelling e port forwarding. In questo modo si permette ad un dispositivo di passare attraverso firewall che molte volte sono configurati in modo molto ristretto, soprattutto all'interno delle aziende.
MQTT non fornisce alcun supporto per l'etichettatura dei messaggi con tipi e altri metadati per aiutare i client a comprenderne il formato in modo live. I messaggi MQTT possono essere utilizzati per qualsiasi scopo, ma tutti i client devono conoscere il formato dei messaggi per permettere la comunicazione.
CoAP invece fornisce supporto integrato per la negoziazione del contenuto e la discovery in modo tale da permettere ai dispositivi di interrogarsi tra di loro e trovare un modo di scambiarsi i dati.
Entrambi protocolli hanno dei vantaggi e degli svantaggi e la scelta del migliore dipende dall'applicazione che bisogna sviluppare.
Possiamo riassumere le caratteristiche principali dei due protocolli nella seguente tabella:

	MQTT	**CoAP**
Application layer	Livello singolo	Livello singolo con 2 sottolivelli (Messaggio e Richiesta-Risposta)
Transport layer	TCP	UDP
Meccanismo di affidabilità	3 QoS	Messaggi CON, NON, ACK e ritrasmissioni

Architetture supportate	Publish-subscribe	Richiesta-risposta, Observe e Publish-subscribe

3.4. Kura

Kura è un framework open-source per IoT rilasciato da Eurotech che fornisce diverse funzionalità per quanto riguarda la pubblicazione di messaggi verso un broker MQTT e la sottoscrizione a topic specifici. Inoltre è capace di utilizzare qualsiasi protocollo per comunicare con i dispositivi di più basso livello all'interno della WSN.
L'architettura è definita dal seguente schema:



Utilizza OSGi che è un sistema a componenti dinamico di Java che permette di semplificare lo sviluppo di software riusabile. All'interno di Kura ritroviamo molte delle funzionalità fornite dal framework proprietario Eurotech.
È consigliato l'utilizzo di un formato per il payload creato appositamente da Eurotech chiamato *EDCPayload*. È composto dai seguenti campi principali:

1. *Timestamp* : è l'istante in cui il dato è stato inviato verso la piattaforma EC.
2. *EC Metrics* : è una struttura dati composta dal nome della variabile, il valore della variabile e il tipo.
3. *EC Position* : è una struttura dati che contiene al suo interno tutte le informazioni riguardo la posizione di un nodo
4. *Body* : è una parte del payload che può contenere qualsiasi tipo di informazione e che non verrà utilizzata dalla piattaforma per l'analisi statistica.

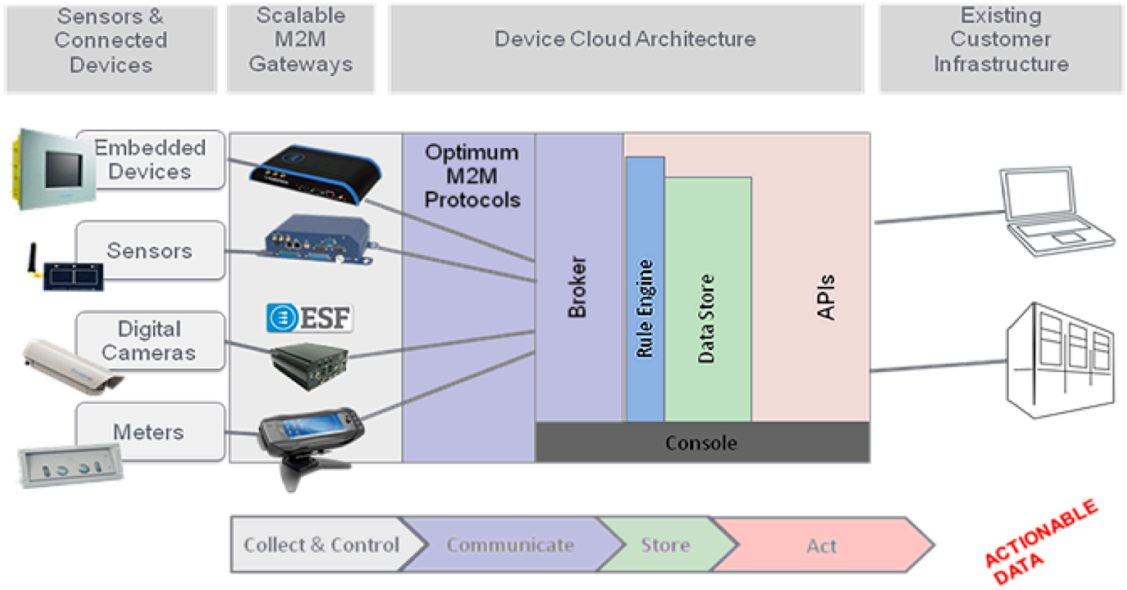
All’interno di Kura è presente praticamente la stessa struttura dati ma con un nome diverso. È possibile utilizzare questo tipo di oggetto soltanto con il *CloudService*. Questa è una delle potenzialità di questo servizio rispetto al più generico DataService che riceve in ingresso soltanto un insieme di byte.
Nell'ultima versione rilasciata fine dicembre 2014 è stato aggiunto anche il supporto al Raspberry PI B+ e sono stati corretti diversi bug relativi al client Paho di MQTT.

All'interno di Kura per comunicare con un broker MQTT si può scegliere tra tre diverse soluzioni:

1. *DataTransportService*
    - Disponibile per la messaggistica standard di MQTT.
    - Permette ai client del servizio di connettersi al broker, pubblicare messaggi per ricevere messaggi dopo la sottoscrizione ad un determinato topic.
2. *DataService*
    - Utilizza il *DataTransportService*.
    - Aggiunge delle funzionalità per la gestione della connessione ai broker, il buffering dei messaggi pubblicati e l’invio di messaggi con priorità.
3. *CloudService*
    - Estende le funzionalità del DataService.
    - Permette di ottenere comunicazioni più complesse rispetto al servizio precedente (come ad esempio comunicazione di tipo richiesta-risposta)
    - Gestisce una connessione singola verso il broker tra più applicazioni
    - Fornisce un modello dei dati per la rappresentazione del payload con la serializzazione della codifica e della decodifica.
    - Gestisce il ciclo di vita della pubblicazione dei messaggi per i dispositivi e le applicazioni.

3.5. Everyware Cloud

L'architettura di Everyware Cloud è la seguente:



Everyware Device Cloud è l'implementazione proprietaria di Eurotech che aggiunge a Kura le seguenti funzionalità:

1. API REST ottimizzate per la ricerca di dispositivi, risorse, dati interni inviati dai dispositivi, attraverso l'utilizzo di client basati su Jersey.
2. Sistema di regole avanzato basato su un motore SQL che elaborando i dati che sono stati pubblicati esegue un'azione immediata rispetto alla direttiva specificata dall'utente. Le regole di tipo statistico agiscono tenendo conto dei dati in tempo reale e le azioni possibili sono:
    1. Invio di una e-mail
    2. Invio di un messaggio SMS
    3. Invio di una notifica Twitter
    4. Pubblicazione di un evento MQTT
    5. Esecuzione di chiamate sulle API REST
3. L'implementazione ottimizzata per l'utilizzo di più broker attraverso l'interazione con un server MQTT basato su Mosquitto (broker MQTT open-source).
   L'implementazione di un possibile supporto CoAP per EDC sarebbe semplificato dal punto di vista della gestione dei device in quanto basterebbe andare ad implementare

la ricerca dei dispositivi all'interno della struttura interna di Eurotech attraverso l'utilizzo delle API REST fornite dall'azienda.

4. La funzionalità principale però rimane quella di poter fare affidamento ai server interni dell'azienda stessa per la memorizzazione dei dati provenienti dai dispositivi e naturalmente nel caso in cui si hanno dei problemi non bisogna far altro che contattare Eurotech. Lo stesso non si potrebbe dire nel caso in cui si implementasse una funzionalità in Kura dove siamo noi stessi i responsabili del salvataggio e del mantenimento dei dati. Eurotech utilizza un database no-SQL nella memorizzazione dei dati che sono replicati tra diverse aree geografiche per ottenere un'affidabilità massima. La parte relativa agli utenti, ai dispositivi e ai dettagli relativi all'account sono invece replicati in un database separato di tipo SQL.

Le classi principali messe a disposizione nelle API REST riguardano:

- *Assets* : visualizzazione della lista degli assets che possono essere ricondotti a dispositivi o ad entità di più alto livello.
- *Topics* : visualizzazione di tutti i messaggi pubblicati in un certo Topic.
- *Metrics* : permette di ottenere una serie di misure inviate da dispositivi direttamente collegati al framework.
- *Messages* : attraverso le classi di questa API riusciamo ad esempio a ottenere una lista di messaggi dall'account Everyware Cloud, visualizzare il numero di messaggi in un account, ricercare i messaggi per identificativo dell'asset, ricercare i messaggi per Topic e trovare i messaggi pubblicati in un certo periodo di tempo.
- *Rules* : restituisce una lista di regole dall'account Everyware Cloud e permette di mostrare i dettagli di una regola specifica.
- *Users* : visualizzazione degli utenti per un account Everyware Cloud.
- *Accounts* : ottenere la lista degli account visibili per specifici dati di login.

Tutte le API permettono di ottenere i risultati anche in formato JSON.
Possiamo inoltre creare endpoint REST attraverso l'utilizzo di annotazioni specifiche di Jersey (*@Path*, *@GET*, *@POST*). Jersey è un framework per la realizzazione di Web Services RESTful in Java. È possibile realizzare un endpoint che riceve le notifiche in callback dall'Engine delle Rules di Everyware Cloud, può memorizzarle e reinviarle in seguito a richieste REST su di esso da parte di un client.
Infine possiamo utilizzare funzionalità REST asincrone in modo tale da permettere l'utilizzo

di HTTP e REST per la sottoscrizione a Topic di interesse e invio in real-time dei dati aggiornati.

Utilizzando Google Dashboard è possibile inoltre definire codice per la rappresentazione di dati in tempo reale secondo una scala di riferimento specificata dallo sviluppatore stesso.
Ad esempio è possibile definire il look and feel dei grafici ed effettuare l'elaborazione dei dati restituiti dal *DataSource* di EDC utilizzano un formato di query chiamato *Google Chart Tools Query Language*.
Naturalmente è necessario che la query sia formattata secondo il formato specifico del componente scelto in Google Dashboard.

4. Progettazione

4.1. Introduzione

L’implementazione di un supporto scalabile CoAP per Kura deve essere concepita come Service all’interno del framework Kura.
Si tratta di un bundle OSGi che può essere caricato in 2 modi diversi all’interno del broker:

1. Attraverso il deploy tramite interfaccia.
2. Attraverso la modalità di installazione remota (disponibile solo quando si fa girare il framework in modo nativo e quindi senza l’utilizzo dell’emulatore).

Il Bundle è composto da un server CoAP che inizialmente verrà concepito con supporto al protocollo di trasposto UDP e non DTLS per avere lo sfruttamento totale di tutte le funzionalità fornite da Californium.

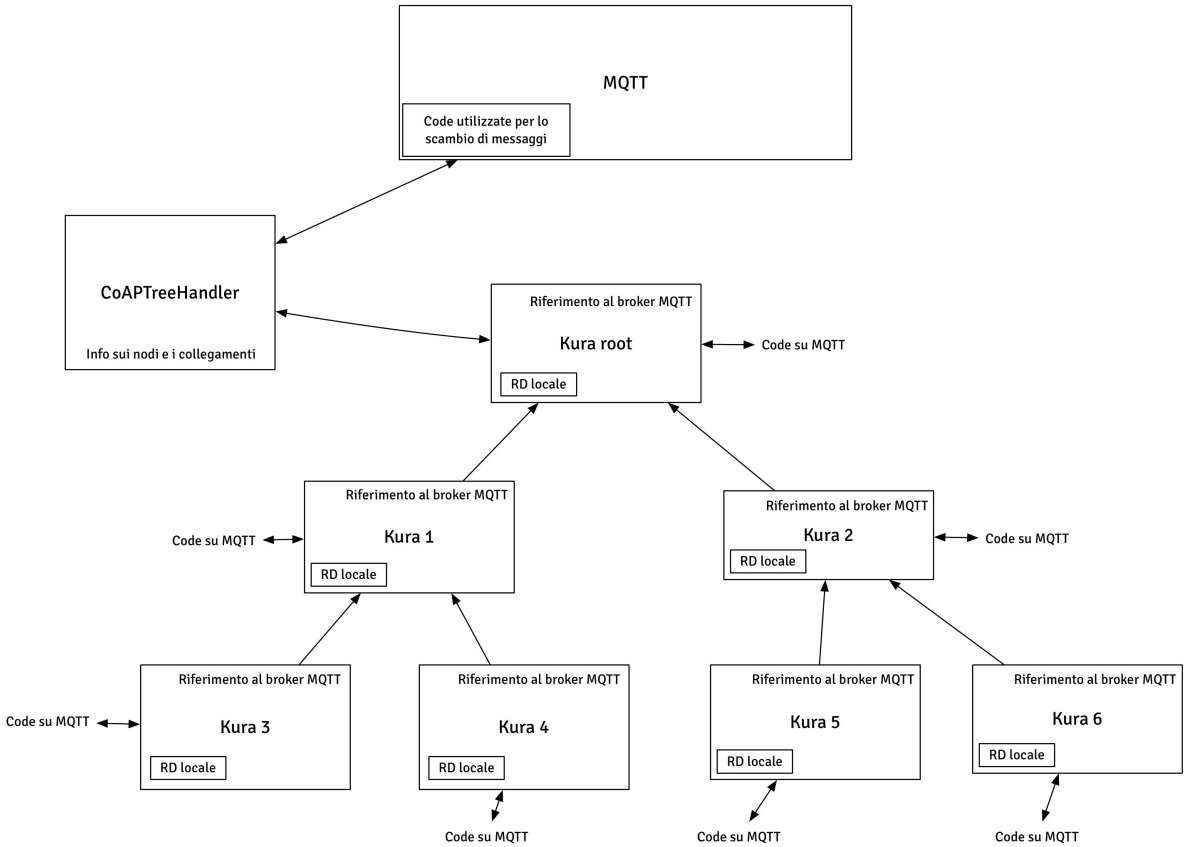
L'obiettivo del supporto non è solo quello di avere un server locale all'interno del framework, ma anche quello di poter effettuare richieste CoAP su risorse legate a dispositivi non fisicamente connessi al broker Kura da cui parte la richiesta REST.

Senza l'utilizzo di tutte le funzionalità fornite da Everyware Cloud, quindi realizzando un supporto specifico per Kura, c'è bisogno di implementare tutta la parte di sincronizzazione relativa ai dispositivi e alle risorse in maniera tale che l’aggiunta o la rimozione di una qualsiasi entità sia gestita in modo tale da avere:

1. Dati sempre aggiornati da qualsiasi punto ci troviamo.
2. Tempi di mancata sincronizzazione ridotti al minimo.

Partendo dal presupposto che una soluzione perfetta sono state definite tre entità principali che possono aiutare il supporto a gestire al meglio la caduta di uno qualsiasi dei nodi della gerarchia.
Dato che il punto forte di Kura è proprio l’integrazione con il broker MQTT per ricevere messaggi di controllo o messaggi inviati da dispositivi connessi al framework.
Per sfruttare le API (poche rispetto ad Everyware Cloud) fornite, è stato scelto di utilizzarlo

per lo scambio di messaggi di sincronizzazione tra i vari broker CoAP.
Il motivo è che il protocollo MQTT segue il modello publish-subscribe e quindi è molto più indicato utilizzarlo per lo scambio di messaggi tra più entità, soprattutto dal punto di vista della qualità di servizio che in MQTT è definita secondo tre livelli.
Tutto questo non è presente in CoAP essendo un protocollo client-server ottimizzato per lo scambio di messaggi con un basso overhead finale.
Uno schema ad alto livello dell'architettura proposta è il seguente:



È presente un componente chiamato *CoAPTreeHandler*, il gestore di tutta la gerarchia.
Il suo compito principale è quello di fornire a qualsiasi nodo che lo richiede le informazioni riguardanti la situazione gerarchica attuale.
Inoltre consente di far fronte ad eventuali problemi di rete. Infatti nel momento in cui non è più disponibile un nodo deve decidere come gestire i nodi figli in modo tale che continuino a lavorare senza problemi.
Ogni nodo ha al suo interno una struttura dati chiamata Resource Directory.
I vari livelli della gerarchia sono definiti nel modo seguente:

- *Livello 0* : nodo radice
- *Livello 1* : broker di un dominio specifico senza sottogruppi
- *Livello 2* : broker che hanno al loro interno sottogruppi di un particolare dominio

In assenza di tutte le funzionalità fornite da Eurotech nella sua piattaforma cloud, bisogna implementare da zero la parte di gestione delle informazioni su tutti i nodi e dei collegamenti fra di essi.

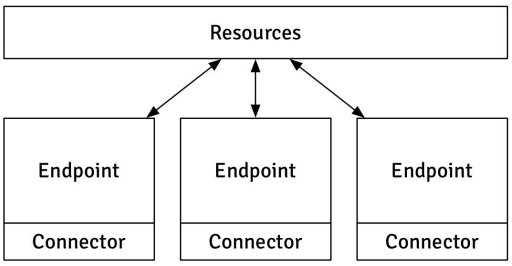
4.2. Requisiti

Affinché tutto funzioni al meglio devono essere rispettati i seguenti requisiti:

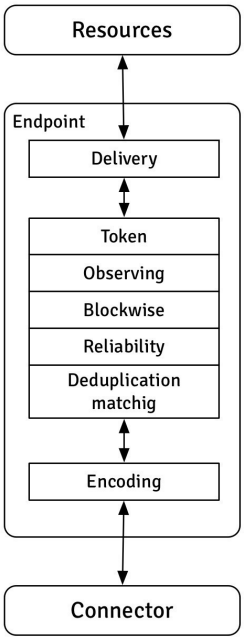
1. Bisogna stabilire una connessione verso il server MQTT. A questo proposito serve specificare una struttura ben precisa per le code presenti all'interno del broker MQTT. Il pattern scelto è il seguente: `IP_MQTT/coap/dominio/gruppo[/sottogruppo]/dispositivo/`. Sono presenti comunque delle code che vengono utilizzate dal gestore della gerarchia per mantenere il più possibile aggiornati i nodi riguardo l'identificativo attuale della radice e la replicazione della stessa in modo tale da garantire il corretto funzionamento del supporto anche in caso di improvvisa interruzione del servizio.
2. Bisogna specificare il dominio prima della connessione alla gerarchia. L'indirizzo del broker con *CoAPTreeHandler* attivo viene spedito dal broker che lo contiene dopo che è arrivata la richiesta della root.
3. Si deve poter effettuare una comunicazione inter-dominio tramite il nodo radice.
4. Si devono sfruttare le API Californium per il protocollo CoAP per realizzare risorse esterne senza dover prima importare i file *.jar*.
5. Il *CoAPTreeHandler* può trovarsi in qualsiasi posizione e interagisce con i nodi attraverso MQTT.

4.3. Considerazioni su Californium

All'interno di Californium sono presenti tre componenti molto importanti che permettono di mantenere una certa flessibilità nella composizione delle applicazioni basate su CoAP:
Lo schema di connessione di *Resource*, *Endpoint* e *Connector* è il seguente:



Resources : ogni server mantiene una struttura ad albero dove ogni nodo è una risorsa che viene identificata da un indirizzo. Quando una richiesta arriva al server, esso ricerca la risorsa all'interno della struttura ad albero e se la trova elabora la richiesta e risponde con un codice di risposta adeguato (con relativo payload).
Endpoint : un endpoint integra tutta l'implementazione del protocollo CoAP. Si occupa della decodifica del datagramma in un oggetto *Request* ed effettua il forwarding verso la *Resource*.
Lo schema principale è il seguente:



I componenti sono:

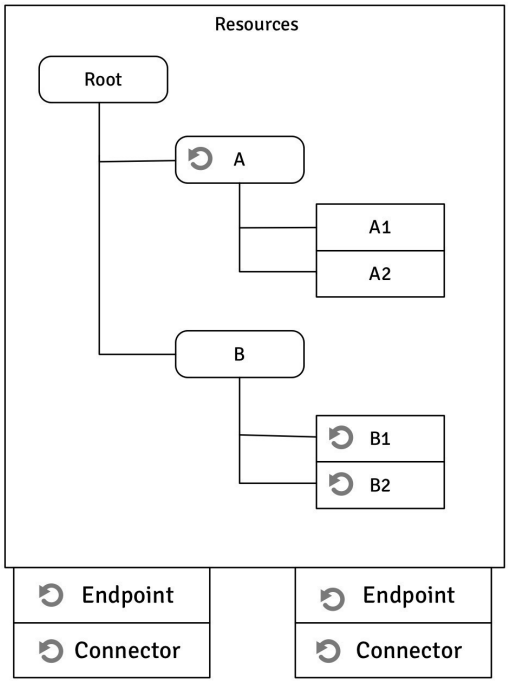
1. *Token* : genera un ID per ogni nuova richiesta.
2. *Observing* : effettua la relazione tra client e Resource per la notifica delle modifiche.
3. *Blockwise* : permette di suddividere un messaggio di grandi dimensioni in pacchetti più piccoli.
4. *Reliability* : permette la ritrasmissione di messaggi CON (messaggi che richiedono un ACK) che sono stati persi durante il primo invio.

5. *Deduplication matching* : permette il riconoscimento di messaggi duplicati.

Abbiamo poi il *Connector* che si occupa di come il server invia e riceve i messaggi e gestisce il protocollo di trasporto ma non è a conoscenza del formato dei messaggi CoAP.
Inoltre non è una socket in quanto un endpoint non effettua una query verso il Connector per la ricezione di un altro messaggio ma è il Connector che richiama l'endpoint associato e gli passa i nuovi messaggi.
I metodi dei connettori sono non bloccanti e ogni Connector ha la propria politica di concorrenza.

Sono presenti, all'interno di Californium, diversi aspetti che possiamo considerare riguardo all'implementazione vera e propria:

- Utilizzando un singolo thread per tutte le elaborazioni potremmo liberarci di tutti problemi relativi alla concorrenza ma le prestazioni ne risentirebbero.
- Utilizzando un pool di thread per l'esecuzione concorrente, ad esempio di richieste indipendenti, con un numero variabile che può essere scelto in fase di elaborazione, su una macchina quad-core si è riusciti ad ottenere (con 4 thread) quasi il doppio delle prestazioni rispetto all'utilizzo di un singolo thread.
- Nell'implementazione propria ogni *Connector*, ogni *Endpoint* e ogni *Resource* ha il proprio modello di concorrenza. Questo significa che è stato scelto un modello stage-based in cui ogni messaggio in arrivo e in uscita viene elaborato in serie dai tre componenti. Questa operazione potrebbe avvenire in parallelo ma non sarebbe efficiente in quanto verrebbero introdotti troppi context switch. In genere alcune risorse possono elaborare le richieste in modo concorrente mentre altre presentano delle sezioni critiche:



Se una risorsa non definisce un pool di thread viene ereditato il pool dal parent e se non è presente una risorsa che definisce un proprio pool di thread sul percorso associato allora verrà utilizzato lo stesso thread che elabora il protocollo CoAP.
È possibile semplicemente aggiungere la keyword "synchronized" alla dichiarazione del metodo per specificare una sezione critica da eseguire in single-thread.
Mentre nel caso di HTTP possiamo ottenere l'affidabilità grazie a TCP, in CoAP questo non è possibile e dobbiamo implementarla a livello applicazione.
Un endpoint deve essere preparato a ricevere lo stesso messaggio più volte e se un messaggio arriva una seconda volta il server deve rispondere con lo stesso ACK, RST o risposta in genere, senza eseguire la richiesta una seconda volta. È essenziale comunque che la rilevazione dei duplicati sia un'operazione atomica, in modo tale da eseguire soltanto una richiesta alla volta se arrivano contemporaneamente.

Californium memorizza tutti gli scambi in un oggetto di tipo *ConcurrentHashMap*, una struttura dati molto potente e altamente ottimizzata che supporta l'esecuzione concorrente. Infatti viene utilizzato il metodo *putIfAbsent()* per ricercare se un messaggio è già presente e inserirlo nella mappa se non lo è.
Per evitare che la struttura dati diventi troppo grande, un thread controlla periodicamente l'età di tutte le entry e se sono scadute le rimuove dalla HashMap. Questo modo di gestire gli intervalli di tempo è molto più efficiente rispetto a quello che utilizza un timer per controllare

la scadenza.
Californium permette di effettuare il wrapping del protocollo di trasporto all'interno di un Connector che richiama la propria strategia multi-threading. Si può cambiare in modo dinamico il numero di thread in un Connector, in modo tale da adattarlo ai differenti broker.

Un altro aspetto interessante è quello relativo ai Garbage Collector utilizzati da Java. Infatti quando Californium esegue molte operazioni contemporaneamente, effettua allocazioni e deallocazioni di grandi quantità di memoria.
Scegliere la strategia di garbage colletion appropriata può migliorare di molto le prestazioni e il fattore più importante è quello relativo al tempo di sospensione dei thread: quando Californium ha a disposizione uno spazio di memoria più grande può elaborare più richieste prima che ci sia un'altra iterazione da parte del Garbage Collector. Ad un certo punto il GC sospende tutti i thread per avere accesso esclusivo alla memoria e questo si traduce in una maggiore latenza tra le operazioni vere e proprie del protocollo.
Per ottimizzare le prestazioni si può effettuare il tuning di alcuni parametri del GC. Si può scegliere infatti di utilizzare un algoritmo che opera in modo concorrente rispetto agli altri thread riducendo il tempo di sospensione di quelli di Californium.

4.4. Resource Directory

Le funzionalità del supporto si basano sull'utilizzo di un'estensione del protocollo CoAP chiamata Resource Directory. Il documento RFC attuale (costantemente aggiornato anche a distanza di poche settimane) è: *draft-ietf-core-resource-directory-02*.
La RD consiste in una struttura dati che consente di memorizzare endpoint e risorse appartenenti a diversi domini e gruppi (con eventuali sottogruppi) e consente di effettuare operazioni dinamiche come:

1. Aggiunta-Rimozione-Aggiornamento-Lookup di endpoint: una funzionalità molto importante dal punto di vista della gestione della RD stessa è quella che riguarda la rimozione automatica degli endpoint presenti. In questo modo non dobbiamo gestire la disconnessione improvvisa degli endpoint perché se essi non effettuano un update entro un certo periodo di tempo verso la RD verranno eliminati dalla lista.
2. Aggiunta-Rimozione-Aggiornamento-Lookup delle risorse: questa funzionalità è adatta alla ricerca generale tra tutti i broker collegati secondo un certo criterio per poter eseguire determinate richieste REST anche se ci si trova in un nodo non

direttamente collegato alla risorsa interessata.

Tutti i risultati vengono restituiti nel payload nel formato CoRE Link Format.

Il problema principale da affrontare è quello dell'adattamento della Resource Directory allo scambio di messaggi MQTT per avere disaccoppiamento tra i broker.

Tutti i nodi possono in genere integrare il bundle del CoAPTreeHandler e quello del server CoAP per le risorse locali. Nel secondo caso l'integrazione della Resource Directory avviene in una radice di risorse separata in modo tale da renderla indipendente dalle altre funzionalità.

Nel file RFC relativo all'estensione Resource Directory di CoAP sono presenti le seguenti possibili operazioni:

- *Registrazione delle risorse sulla Resource Directory* : richiesta POST da un end-point con la lista delle risorse da aggiungere alla RD.

```
Req: POST coap://rd.example.org/rd?n=node1&lt=1024
Etag: 0x3f
Payload:
</sensors/temp>;ct=41;rt="TemperatureC";if="sensor",
</sensors/light>;ct=41;rt="LightLux";if="sensor"

Res: 2.01 Created
Location: /rd/node1
```

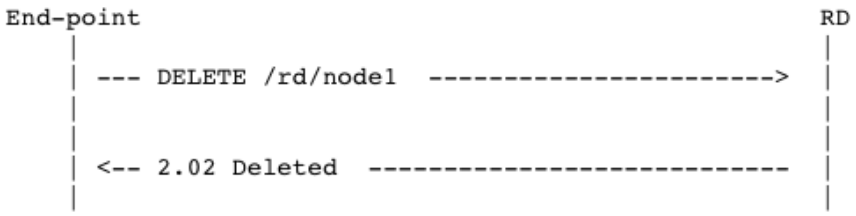
- *Aggiornamento delle risorse* : richiesta PUT con il payload in CoRE Link Format per l'aggiornamento.

```
Req: PUT /rd/node1
Etag: 0x40
Payload:
</sensors/temp/1>;ct=41;ins="Indoor";rt="TemperatureC";if="sensor",
</sensors/temp/2>;ct=41;ins="Outdoor";rt="TemperatureC";if="sensor",
</sensors/light>;ct=41;rt="LightLux";if="sensor"

Res: 2.04 Changed
```

- *Validazione delle risorse* : richiesta GET sull'interfaccia well-known con l'inclusione dell'ultimo ETag.

```
Interaction:  RD -> EP

Path:  /.well-known/core

Method:  GET

Content-Type:  application/link-format (if any)

Etag:  The Etag option MUST be included

Parameters:  None

Success:  2.03 "Valid" in case the Etag matches

Success:  2.05 "Content" in case the Etag does not match, the
   response MUST include the most recent resource representation and
   its corresponding Etag.
```

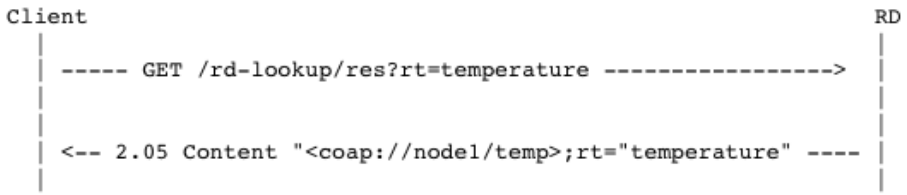
- *Rimozione delle risorse* : richiesta DELETE sull'end-point.




```
Req: DELETE /rd/node1

Res: 2.02 Deleted
```

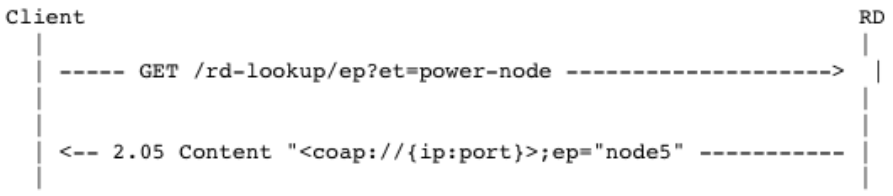
- *Lookup delle risorse* : richiesta GET dal client alla Resource Directory.

```
Interaction:  Client -> RD

Path:  /{rd-base} or e.g. /{rd-base}/{end-point name}

Method:  GET

Content-Type:  application/link-format (if any)
```




```
Req: GET /rd-lookup/res?rt=temperature

Res: 2.05 Content
<coap://{ip:port}/temp>
```

- *Lookup di endpoint* : richiesta GET su */rd-lookup/ep*.




```
Req: GET /rd-lookup/ep?et=power-node
```

4.5. CoAPTreeHandler

Come accennato in precedenza è il componente più importante di tutta l'architettura. È il perno centrale su cui si basa tutta la gestione dei nodi.
Le caratteristiche principali sono:

1. Fornisce prima di tutto un Service utilizzabile dagli altri bundle
2. I server CoAP possono interrogarlo tramite messaggi MQTT.
3. È a conoscenza della radice dell'albero e può essere utilizzato nella gestione di root multiple per la replicazione. Più in generale si possono ricavare informazioni su tutta la gerarchia.
4. È attivato da eventi implementati tramite listener sulle code per ottenere gli aggiornamenti dai broker.
5. È necessario che sia sempre disponibile. Si deve quindi cercare di replicarlo in modo

tale che nel momento in cui ci sia un failure di quello attivo venga subito messo in esecuzione il servizio di backup. Se questo non è possibile allora bisogna che comunque sia installato su una macchina che abbia un uptime elevato. Se anche questo non fosse possibile, allora si deve cercare di implementare un meccanismo di recupero delle informazioni sui nodi e forzare lo scambio delle nuove proprietà delle risorse tra di essi.

4.6. Gestione dinamica delle risorse

L'obiettivo successivo è quello di poter aggiungere risorse in modo dinamico quando aggiungiamo un servizio.
È molto importante ad esempio quando abbiamo un nuovo sensore da voler esporre esternamente e vogliamo creare velocemente una risorsa CoAP all'interno del Service implementato in precedenza.
I passi che devono essere eseguiti per poter ottenere questo risultato sono:

1. Specificare all'interno della classe appena creata i metodi che servono per l'attivazione e la disattivazione del Service CoAP.
2. Per poter modellare correttamente la nostra risorsa possiamo aver bisogno di specificare i metodi di callback relativi alle varie operazioni REST (*GET*, *POST*, *PUT* e *DELETE*). Viene presa in considerazione la classe principale di Californium relativa alle risorse (*CoapResource*). Una volta fatto l'*override* di tutti metodi che occorrono possiamo andare ad inserire all'interno dei metodi *activate* e *deactivate* le chiamate alle funzioni di aggiunta rimozione delle risorse relative al Service.

In questo modo riusciamo a mantenere aggiornata la lista delle risorse esposte dal Service in modo dinamico e senza doverlo riavviare.
Lo schema appena descritto può essere utilizzato anche in concomitanza con la Resource Directory per ottenere un supporto di CoAP scalabile, estendibile e sempre aggiornato.

La gestione delle risorse locali e quelle esterne è diversa. Infatti quelle locali vengono gestite in maniera ottimizzata sulla stessa piattaforma senza quindi andare ad effettuare delle richieste CoAP.
Quelle esterne invece dato che utilizzano la Resource Directory sono organizzate in modo diverso. L' aggiunta e la rimozione (o anche l'aggiornamento) delle risorse sulla RD avviene attraverso le normali richieste REST.

Gli endpoint non sono memorizzati permanentemente nella RD ma vengono conservati per un periodo di tempo limitato.
Come descritto all'interno della RFC si può utilizzare un timer per ogni entry. Una volta scaduto ci permette di capire che dobbiamo eliminarla senza che l'endpoint debba effettuare una richiesta di DELETE.
Dato che comunque ci troviamo in un ambiente molto dinamico questo tipo di approccio è utile soprattutto quando un nodo cade e non ha la possibilità di de-registrarsi.
L'unico aspetto negativo è quello che l'endpoint entro un certo intervallo di tempo deve inviare una richiesta di tipo PUT per azzerare il countdown.

L'implementazione principale del server utilizza una delle funzionalità più importanti dei bundle OSGi: la fornitura di servizi con l'exporting dei package.
Lo sviluppo di una risorsa personalizzata con i vari metodi REST è molto semplice in quanto non dobbiamo fare altro che importare solo i package di Californium che ci servono direttamente dal CoAP Service.

Il modo in cui i vari broker interagiscono nell'architettura è molto importante in quanto gestendo opportunamente lo scambio di messaggi e la propagazione degli aggiornamenti riusciamo ad ottimizzare le prestazioni del supporto.
Ogni nodo all'interno dell'architettura specifica il proprio dominio e il gruppo di appartenenza. Questo permette di limitare l'interesse verso le risorse esterne da memorizzare all'interno della Resource Directory. Naturalmente se non viene specificato il gruppo, il numero di messaggi che il broker dovrebbe elaborare sarebbe maggiore.

Una prima ottimizzazione per quanto riguarda il servizio *CoAPTreeHandler* è che ogni volta che riceve una richiesta per il riferimento alla radice dell'albero aggiorna il proprio stato interno sul nodo che si è appena collegato ed ha inviato la richiesta.
L'aggiornamento della Resource Directory all'interno della root invece avviene, come da protocollo, solo tramite una richiesta REST.
Considerando l'inserimento di un nodo ad un livello gerarchico inferiore bisogna capire quali informazioni propagare verso l'alto fino ad un determinato livello:

1. Una prima soluzione sarebbe quella di inviare tutti i link delle risorse al livello superiore. L'aspetto negativo è che il nodo parent potrebbe avere un numero di figli

elevato e quindi la propria RD potrebbe contenere un numero troppo elevato di risorse.

2. Un'altra soluzione sarebbe quella di inviare solo il numero totale di risorse. In questo caso però ci sarebbe il problema di non essere a conoscenza del tipo di risorse presenti nel nodo figlio e quindi potremmo comunque avere dei problemi nel caso di richieste specifiche. Inoltre per ogni nuova risorsa collegata dobbiamo inviare gli aggiornamenti ai nodi di livello superiore.
3. Una terza soluzione sarebbe quella di inviare solo le proprietà delle risorse attualmente collegate. Rispetto alla soluzione precedente minimizziamo il numero di interazioni tra i broker dato che non inviamo tutti i link ma soltanto le informazioni utili ad un'eventuale ricerca. Un comportamento simile non è descritto all'interno del documento ufficiale della Resource Directory e quindi deve essere implementato a parte come caratteristica specifica del supporto.

Se avviamo un nuovo broker che presenta le stesse proprietà gerarchiche di altro nodo già presente bisogna che prima di entrare a far parte dell'albero ci sia l'intervento del *CoAPTreeHandler*.
Prendiamo in considerazione i vari casi possibili:

1. Aggiunta di un nodo foglia con un percorso già esistente : viene comunque aggiunto alla stessa root del broker gemello.
2. Aggiunta di un nodo intermedio : viene aggiunto come discendente del nodo gemello
3. Aggiunta di un'altra root : se non specifichiamo il dominio ed eseguiamo una root request ci viene restituito l'indirizzo della radice al quale collegarci. Eventuali richieste da parte di altri nodi non verrebbero comunque redirette verso il nuovo nodo e potrebbe comunque essere utilizzato successivamente come nuova root nel caso cada quella precedente.

4.7. Qualità di servizio

Per quanto riguarda la qualità di servizio che vogliamo ottenere attraverso il supporto bisogna specificare che durante la comunicazione tra i broker utilizzando MQTT, possiamo specificare la QoS tra le tre disponibili per quanto riguarda questo tipo di protocollo:

1. At-least-once
2. At-most-once

3. Exactly-once

In relazione alla comunicazione locale attraverso CoAP, dato che comunque si tratta di un protocollo client-server molto leggero, possiamo solo affidarci all'utilizzo di ACK tra gli endpoint.

Naturalmente quando utilizziamo i diversi livelli di servizio forniti da MQTT ci possiamo trovare di fronte a duplicazione dei messaggi.
Dato che comunque sarebbe troppo pesante per il framework memorizzare tutti i messaggi per scoprire poi che uno di essi è un duplicato, bisogna gestirli a livello di supporto CoAP.

Per quanto riguarda gli endpoint l'arrivo di un messaggio duplicato:

1. Nel caso di aggiunta (o aggiornamento) deve essere prima effettuata una ricerca per stabilire se l'endpoint è già presente e poi andare ad inserirlo (o aggiornarlo) se l'esito della ricerca è negativo.
2. Nel caso di rimozione invece dato che il dispositivo è già stato rimosso l'arrivo di un altro messaggio non comporta nessun problema.

5. Implementazione

5.1. Introduzione

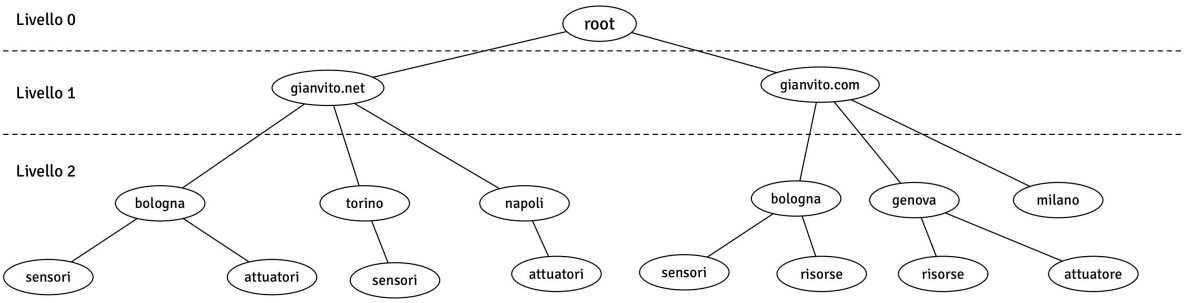
Questo capitolo è composto in modo tale che verranno prima elencate le funzionalità del service base, che contiene tutte le classi e le librerie importate esternamente e che vengono utilizzate poi dai service di livello superiore. Verranno poi descritti i bundle OSGi del CoAPTreeHandler (vero e proprio gestore della gerarchia dei broker CoAP) e del server CoAP.
Sull'endpoint a cui fa riferimento il server CoAP è possibile infine aggiungere una risorsa dedicata alle richieste di altre *CoapResource* (nome della classe in Californium) situate in broker remoti.

5.2. Bundle

5.2.1. Service base

È possibile utilizzare questo bundle senza l'installazione di quelli relativi al broker e al CoAPTreeHandler in modo tale da sviluppare applicazioni esterne sfruttando le strutture dati e le utility presenti all'interno dei vari package.

All'interno della gerarchia possono essere presenti tre tipi di broker:



- *Livello 0*
  - Comprende solo il nodo radice.
  - Il compito principale è quello di fare da tramite verso domini diversi. Ogni query che coinvolga il passaggio da un dominio ad un altro passerà per il nodo root dato che è l'unico che contiene al suo interno tutte le informazioni riguardo gli attributi dei domini sottostanti. Naturalmente queste informazioni non sono complete ma sono presenti solo dei collegamenti ai nodi figli.
- *Livello 1*
  - In questo caso i nodi che ne fanno parte sono quelli che ne rappresentano il dominio associato. Il solo nodo parent possibile è quello radice e la sua presenza necessita di aggiornamenti più rapidi rispetto ai nodi di livello 2. Quando un nodo radice viene aggiunto o rimosso, il nodo di livello 2 ne viene subito a conoscenza perché effettua la subscription sui topic relativi e quindi non deve ricevere un messaggio MQTT, su una coda privata, da parte del CTH.
- *Livello 2*
  - È associato sempre a nodi che fanno parte di un dominio e hanno un particolare gruppo di appartenenza e rispetto ai livelli precedenti ha delle proprietà meno tempestive sull'aggiornamento del parent (la ricezione del nuovo parent avverrà sulla coda privata associata). In mancanza di sincronizzazione o nel caso in cui non è mai stato associato a quel parent riceverà l'update nell'intervallo di aggiornamento successivo. Il motivo principale è che il CTH dovrebbe mantenere una lista di nodi interessati a quel broker in particolare e in un ambiente dinamico è molto complicato ed esoso dal punto di vista computazionale andare ad agire in questo modo.

5.2.1.1. Integrazione di librerie esterne

5.2.1.1.1. *Californium*

https://eclipse.org/californium/

È stata scelta Californium come implementazione principale del protocollo CoAP perché:

1. Risulta la più completa al momento. Supporta la maggior parte dei draft relativi alle estensioni di CoAP, in particolare la Resource Directory che è stata utilizzata per lo sviluppo del supporto.
2. È sviluppata in Java e quindi è facilmente integrabile con Kura. È possibile aggiungere molte altre librerie scritte in Java per velocizzare la parte di implementazione di alcune funzionalità.
3. È facilmente estendibile dato che supporta diversi tipi di utilizzo, per ottenere le stesse funzionalità. È possibile partire dagli oggetti semplici ed ottenere funzionalità complesse, oppure partire direttamente da oggetti più elaborati impostando le opzioni necessarie per raggiungere i propri obiettivi.

5.2.1.1.2. *Google Guava*

https://github.com/google/guava

Google Guava è una libreria sviluppata da Google ed utilizzata in molti dei loro progetti, che aggiunge a Java diverse funzionalità riguardo a:

- Collection
- Utility di caching
- Librerie per operazioni concorrenti
- Annotation
- Processing di oggetti di tipo String
- Utility di I/O

Uno dei motivi per cui è stata aggiunta anche questa libreria all'interno del Service è la presenza di una Collection chiamata *Multimap* che consiste in una *Map* del tipo *Map<K, List<V>>* o *Map<K, Set<V>>*.
In genere quando si sviluppa una struttura simile si può intenderla in due modi:

1. Mapping da chiavi singole a valori singoli
    - *Padre 1 —> Figlio 1*
    - *Padre 1 —> Figlio 2*
    - *Padre 1—> Figlio 3*
    - *Padre 2 —> Figlio 4*
    - *Padre 3 —> Figlio 5*
2. Mapping da chiavi uniche a più valori
    - *Padre 1 —> [Figlio 1, Figlio 2, Figlio 3]*
    - *Padre 2 —> Figlio 4*
    - *Padre 3 —> Figlio 5*

Grazie a Guava, è possibile passare da una vista ad un'altra attraverso i metodi presenti all'interno della Collection.
Ad esempio, se vogliamo considerare la seconda vista basterà eseguire il metodo *asMap()* sulla Multimap e verrà restituita una View di quel tipo.
Quando una key non ha almeno un valore associato allora viene rimossa, facilitando così la gestione della struttura dati.

Un altro motivo per cui si è scelto di utilizzare Guava è perché fornisce diversi metodi static per effettuare operazioni sui Set (come ad esempio l'intersezione tra insiemi), attraverso la classe *Sets*.

5.2.1.1.3. *Kryo*

https://github.com/EsotericSoftware/kryo

Kryo è un framework di serializzazione per Java, molto efficiente e veloce. È un progetto open-source ed è aggiornato abbastanza frequentemente.

Il motivo principale della scelta di aggiungere Kryo alle librerie del supporto è che fornisce prestazioni nettamente superiori alla normale serializzazione Java.
Sono stati effettuati diversi test (disponibili a questo indirizzo: http://java-persistence-performance.blogspot.it/2013/08/optimizing-java-serialization-java-vs.html) su un semplice oggetto di tipo *Order* con un paio di campi e i risultati sono abbastanza evidenti:

Order with 1 OrderLine

Serializer	Size (bytes)	Serialize (operations/second)	Deserialize (operations/second)	% Difference (from Java serialize)	% Difference (deserialize)
Java Serializable	636	128,634	19,180	0%	0%
Java Externalizable	435	160,549	26,678	24%	39%
EclipseLink MOXy XML	101	348,056	47,334	170%	146%
Kryo	90	359,368	346,984	179%	1709%

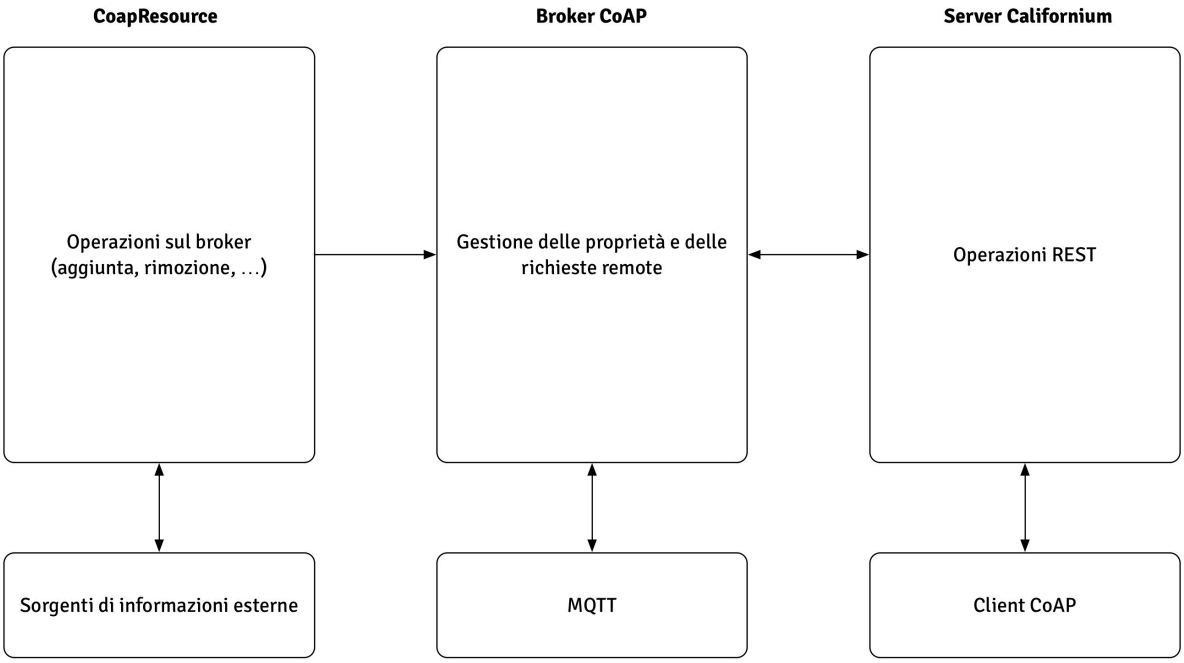
Order with 100 OrderLines

Serializer	Size (bytes)	Serialize (operations/second)	Deserialize (operations/second)	% Difference (from Java serialize)	% Difference (deserialize)
Java Serializable	2,715	16,470	10,215	0%	0%
Java Externalizable	2,811	16,206	11,483	-1%	12%
EclipseLink MOXy XML	6,628	7,304	2,731	-55%	-73%
Kryo	1216	22,862	31,499	38%	208%

L'utilizzo di questo framework migliora di molto le prestazioni quando bisogna trasformare i *CoAPMessage* in array di byte (dato che MQTT non tiene conto del formato dei messaggi) e viceversa.
Un altro vantaggio è quello della massiccia riduzione della dimensione dei messaggi scambiati, aiutando così anche a minimizzare l'uso della rete.

5.2.2. Californium Service

L'obiettivo del bundle Californium è quello di poter avviare un server CoAP non distribuito utilizzando le classi interne di Californium.
Per un maggiore disaccoppiamento dei diversi componenti si è deciso di suddividere le funzionalità in questo modo:



Come è possibile notare dalla figura abbiamo la possibilità di:

1. Avere un Service (*Resource*) dedicato alla risorsa (o alle risorse) da esporre all'esterno. Può eseguire operazioni sul broker, come ad esempio la propria aggiunta, rimozione o la modifica degli attributi.
2. Avviare un altro Service (*CoAPServer*), dedicato questa volta alla gestione di tutte le proprietà delle risorse registrate che può coordinarsi con gli altri broker ed eseguire i metodi di callback in base ai messaggi ricevuti.
3. Eseguire un Service che sfrutta Californium (*CaliforniumServer*) per lanciare un CoapServer che è indipendente dallo stato degli altri 2 servizi e può tranquillamente eseguire le operazioni REST senza essere interrotto dalla gestione dei messaggi MQTT, migliorando così le prestazioni generali del supporto. Naturalmente, implementando il protocollo CoAP, ha la possibilità di interfacciarsi con client CoAP

di ogni tipo (non solo quelli che eseguono Californium) ed è facilmente estendibile (con l'adozione poi del livello DTLS), dato che è indipendente dal broker.

5.2.3. CoAPTreeHandler

Il *CoAPTreeHandler* consiste in un bundle OSGi da installare sul framework Kura per la gestione di tutta la gerarchia di broker CoAP.
Il compito principale è quello di ricevere tutte le richieste provenienti dai nodi Kura e restituire i riferimenti, in modo tale che venga creata una struttura ad albero organizzata per domini e gruppi di appartenenza.
I nodi comunicheranno tra di loro tramite il protocollo MQTT. La sincronizzazione tra nodi e CTH avverrà sempre attraverso questo protocollo.
Il CTH può risiedere su uno qualsiasi dei nodi dato che comunque, utilizzando MQTT, c'è disaccoppiamento tra la posizione di un entità rispetto ad un'altra.

5.2.3.1. Topic e QoS

È molto importante definire le code su cui si andranno a scambiare i messaggi e il livello di Quality of Service per ogni interazione. In base all'importanza del messaggio scambiato è stata scelta una strategia di qualità di servizio differente.
Per creare una comunicazione bidirezionale tra i nodi stessi oppure tra i nodi ed il CTH sono state create code differenti, a seconda che sia il Publisher o il Subscriber ad interagire con il broker MQTT.
Iniziamo con i topic interessati dal Publisher del CTH:

- Topic relativi ai nodi radice:
  - *ROOT_ADDED_TOPIC* : topic generico che viene utilizzato per lo scambio di messaggi che riguardano l'aggiunta di nodi root alla gerarchia
  - *ROOT_NOT_AVAILABLE_TOPIC* : topic generico che viene utilizzato per lo scambio di messaggi che riguardano l'assenza di nodi root nella gerarchia
  - *ROOT_RESPONSE_TOPIC* : topic specifico per un nodo che richiede info sulla radice.
- Topic relativi ai nodi di livello maggiore di 0:
  - *NODE_ADDED_TOPIC* : quando viene aggiunto un nodo di interesse per un altro (non utilizzato al momento ma di utilità in caso di estensioni al protocollo), viene pubblicato un messaggio su questa coda.
  - *PARENT_RESPONSE_TOPIC* : topic specifico del nodo sui cui riceve

messaggi riguardanti un possibile parent da assegnare.

- *PARENT_AVAILABLE_TOPIC* : questo è un topic specifico, che può essere utilizzato nel caso in cui si estenda il protocollo e lo si renda più reattivo anche per i nodi di livello 2. Come detto in precedenza non è stata fatta questa scelta perché si introdurrebbe un overhead troppo alto per il CoAPTreeHandler, dato che dovrebbe analizzare ogni volta tutti i nodi.
- *PARENT_NOT_AVAILABLE_TOPIC* : topic specifico per il nodo che consente di conoscere se il parent attualmente impostato non è più disponibile.
- *CHILD_NOT_AVAILABLE_TOPIC* : topic specifico per un nodo (parent) che viene informato dell'assenza di uno dei suoi child. Viene utilizzato quando devono essere rimossi tutti i riferimenti agli attributi presenti sulle risorse del nodo caduto.

Per quanto riguarda invece i topic su cui il CTH effettua le subscription abbiamo:

- Topic relativi ai nodi radice:
  - *ROOT_REQUEST_TOPIC* : ricezione di richieste di informazioni sul nodo radice attualmente presente nella gerarchia.
  - *ROOT_ON_NODE_UPDATED_TOPIC* : riguarda lo scambio di informazioni, che partono dal nodo e arrivano al CTH, sull'aggiornamento del parent effettuato sul nodo (il parent è stato inviato in un messaggio precedente da parte del CTH).
- Topic relativi ad un nodo generico di livello superiore allo 0:
  - *PARENT_REQUEST_TOPIC* : topic su cui avviene la ricezione di richieste di informazioni sul possibile parent che il nodo mittente può impostare come proprietà interna e su cui può inviare gli aggiornamenti sulle proprietà delle risorse attualmente collegate.
  - *PARENT_ON_NODE_UPDATED_TOPIC* : topic su cui il CTH riceve l'info che il parent sul nodo è stato aggiornato e quindi può di conseguenza fare l'update sulle proprie strutture dati.
  - *NODE_REMOVE_TOPIC* : su questo topic il CTH riceve le info sull'attuale disconnessione di un nodo dell'albero. Deve, quindi, effettuare le azioni necessarie per ristabilire i collegamenti giusti, notificando i nodi interessati.
- Topic legati al Last Will And Testament:
  - *MQTT_LWT_TOPIC* : è il topic che viene utilizzato per conoscere quando un

nodo si disconnette in modo imprevisto.
Le proprietà QoS scelte per il CTH, data l'importanza che esso possiede nel ciclo di vita di tutto il supporto, devono per forza essere di tipo "exactly once" (o almeno "at least once").

5.2.3.2. Strutture dati

Il CoAPTreeHandler, per mantenere le informazioni sui dispositivi che fanno parte della gerarchia utilizza la classe *CTHCollection*.

Una scelta è stata quella di mantenere le informazioni sulla gerarchia e sui nodi che ne fanno parte in due Collection ottimizzate per la ricerca e la sostituzione immediata dei nodi.
La struttura dati che mantiene le informazioni sui percorsi dalla root fino alle foglie e che associa ad ognuno un nodo ben preciso è la *DevicesMap*.
Consiste nell'estensione di una Map composta in questo modo:

- Chiave : percorso sulla gerarchia ad albero
- Valore : nodo presente su quel path (con tutte le sue proprietà)

Grazie all'utilizzo della *DevicesMap,* possiamo effettuare diverse operazioni sull'albero che vanno a migliorare la ricerca nel caso di inserimenti o cancellazioni di nodi.
Possiamo, ad esempio, andare a ricercare tutti i nodi che hanno uno specifico prefisso nel percorso e utilizzare il Set di nodi trovato per gestire al meglio Resource query con attributi di ricerca del tipo "*group=nomegruppo/**".

La seconda struttura dati che fa parte della *CTHCollection* è *ParentTree*. Come accennato in precedenza utilizziamo Google Guava per importare la classe *Multimap*, utilizzata per creare l'associazione uno a molti che sussiste tra padre e figli all'interno della gerarchia.
Vengono definiti anche dei metodi di wrapping sulla Collection in modo tale da semplificare il codice e renderlo flessibile in caso di modifiche su determinate operazioni.

I metodi definiti nella *CTHCollection* vengono poi richiamati dal *CoAPTreeHandler* direttamente alla ricezione di messaggi MQTT con funzioni di:

- Aggiunta di nodi radice, con la pubblicazione del messaggio di notifica.
- Rimozione del nodo radice, con l'aggiornamento della struttura locale e l'invio di messaggi MQTT verso i nodi interessati.
- Aggiunta di nodi semplici alla gerarchia e l'invio del relativo nodo padre.

- Rimozione di nodi con livello gerarchico maggiore di 0 e pubblicazione di messaggi MQTT verso i nodi collegati (e naturalmente aggiornamento delle strutture dati *ParentTree* e *DevicesMap*).
- La ricerca di dispositivi, dato un certo indirizzo MAC (utilizzato per il Last Will and Testament di MQTT).
- Ricerca di un nuovo parent node per il nodo in ingresso scorrendo l'albero dalle foglie fino alla radice.

5.2.3.3. Gestione dell'aggiornamento dei nodi

5.2.3.3.1. *Aggiunta di nodi*

CTHCollection è la classe che si occupa principalmente di gestire la mappa dei dispositivi e l'albero gerarchico dei nodi, mentre in *CoAPTreeHandler* sono presenti tutte le operazioni di notifica.
Nel momento in cui un nodo tenta di collegarsi alla gerarchia, invia o una *rootRequest* o una *parentRequest* e il CTH richiamerà la specifica funzione sulla *CTHCollection*, grazie all'implementazione dell'interfaccia *CTHListener*.
La Collection. in base alla gerarchia del nodo che ha effettuato la richiesta decide:

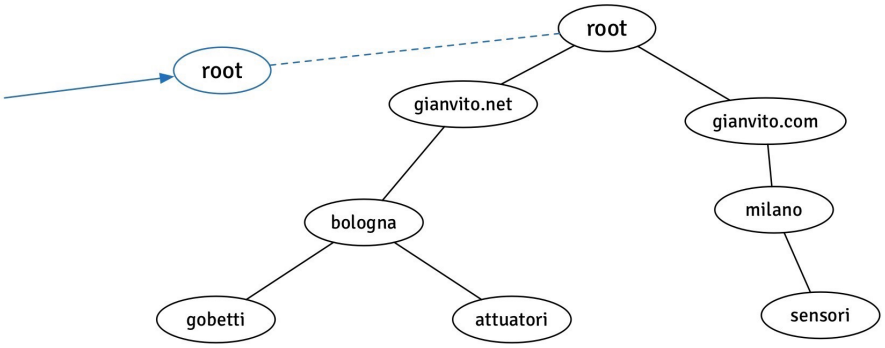
1. Se aggiungerlo come root (con dominio * e gruppo *) quando non è presente nessun nodo radice e se è già presente viene restituito l'indirizzo di quello memorizzato in modo tale da ottenere gestione dei duplicati a livello 0. In questo modo se il nodo radice ha una failure, nell'intervallo successivo, il broker di backup rieffettuerà la richiesta di root e non essendo presente più il precedente gli verrà restituito il suo stesso riferimento. Da quel momento in poi tutti i nodi Kura di primo livello invieranno ad esso gli aggiornamenti delle risorse.
2. Se aggiungere il nodo come nodo di livello superiore allo zero. Si ha un comportamento simile al precedente perché quando abbiamo un duplicato viene aggiunto solo come figlio del nodo originale (che si trova sullo stesso path). È comunque in grado di effettuare *RemoteResource* query ed è in grado di restituire riferimenti alle proprie risorse locali, quindi effettuare POST sulle RD remote. Non è invece in grado di avere nodi figli dato che non verrà mai restituito come potenziale parent in una ricerca effettuata in seguito ad una parent request.

Quando è disponibile un nuovo parent da assegnare vengono effettuate le rimozioni dal

vecchio riferimento (che può comunque restare ancora connesso). Il motivo è che il nodo poteva essere associato precedentemente con un determinato padre e solo in seguito, dopo una nuova parent request, è stato abbinato ad un broker più specifico per quanto riguarda il gruppo di appartenenza.
Infatti nella *CTHCollection.searchNewParentNode* si parte dal nodo con il gruppo più specifico e si risale la gerarchia per ottenere il parent con il path più vicino al gruppo di appartenenza del nodo che ha inviato la request.

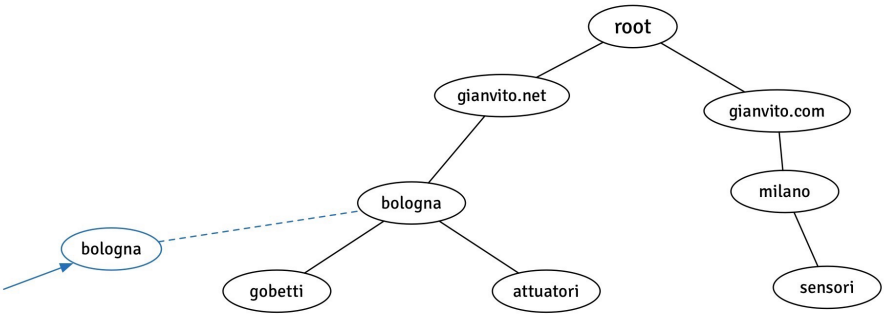
Ecco alcuni esempi di possibili casi:

- Aggiunta di un nodo radice duplicato



Viene solamente aggiunto come nodo figlio della root e non compare nella lista dei dispositivi.

- Aggiunta di un nodo di livello superiore allo zero



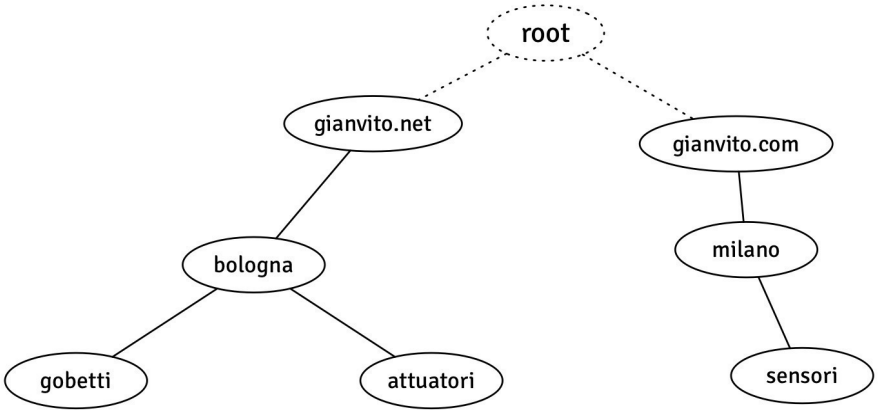
Stesso comportamento del precedente con la differenza che diventa nodo child del nodo originale.

5.2.3.3.2. *Rimozione normale di nodi*

Quando un nodo si disconnette normalmente richiama la propria funzione *disconnectFromTree* che invia un messaggio al CTH. Il CTH quindi si occupa di avvertire tutti gli altri nodi che non è più disponibile e verranno effettuati tutti gli aggiornamenti necessari in base alla gerarchia.

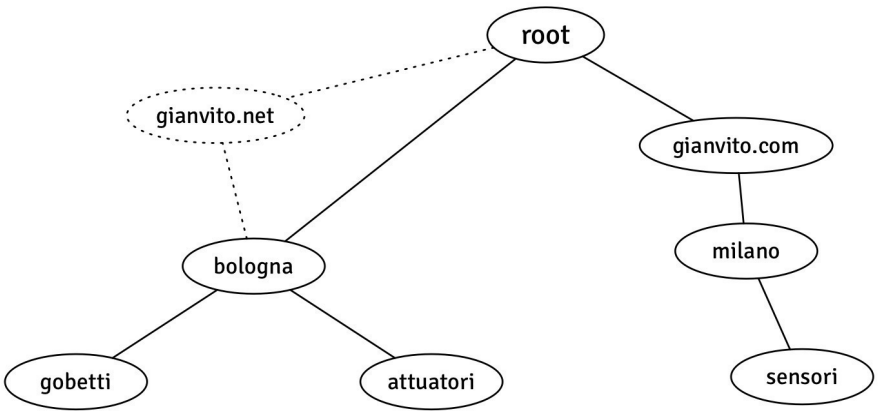
Ecco alcuni esempi di possibili casi:

- Esempio con rimozione di un nodo radice



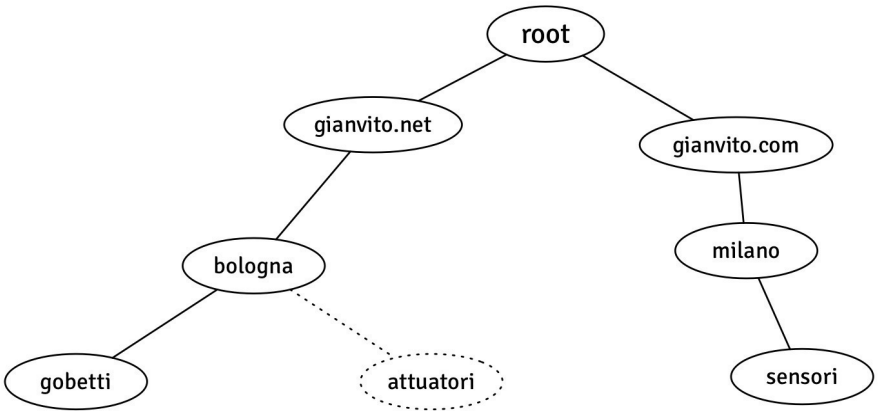
In caso di rimozione della root non è possibile più effettuare RemoteQuery verso domini diversi da quello di appartenenza. In caso di presenza di un duplicato della root, entrerà a far parte della gerarchia e tutti i nodi rimasti senza parent avranno il riferimento verso il nuovo broker a cui invieranno gli aggiornamenti delle risorse presenti.

- Esempio con rimozione di un nodo di livello 1



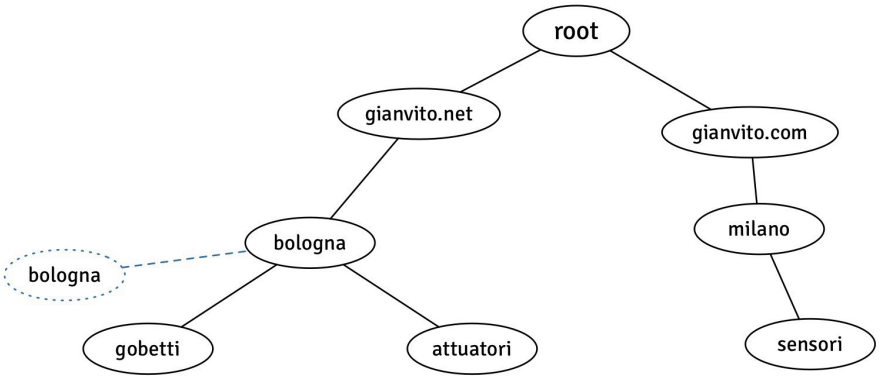
Il CTH, tramite modifiche alle Collection locali e invio di messaggi MQTT ai nodi interessati, assicura che i nodi rimasti parent-less siano associati alla root (fino all'arrivo di un nuovo nodo con le stesse proprietà di quello non più disponibile).

- Esempio con rimozione di un nodo di livello 2 (o maggiore di 2)



Non ci sono particolari precauzioni da prendere in questo caso, tranne che notificare il nodo *bologna* che il figlio non è più disponibile e che quindi deve cancellare le risorse precedentemente associate (con eventuale propagazione di aggiornamenti lungo la gerarchia).

- Esempio di rimozione di un duplicato



In questo caso non ci sono comunque problemi dato che il nodo non era presente tra i dispositivi della gerarchia (era presente solo come nodo child) e quindi devono essere rimossi solo gli eventuali attributi salvati sul nodo originale e poi eliminarlo dalla ParentTree.

5.2.3.3.3. *Rilevamento dei failure dei broker*

Nel caso in cui invece un nodo si disconnetta senza richiamare la funzione specifica verrà utilizzato LWT (Last Will and Testament) di MQTT.
Per determinare con esattezza il nodo disconnesso, si è deciso di utilizzare un pattern che permettesse il riconoscimento sia del percorso nella gerarchia che dell'ID del dispositivo (considerando il MAC address).

Le scelte possibili erano:

1. Topic specifico del dispositivo.
   - Schema:

     Topic : `#MAC/MQTT/LWT`

     Payload : nullo
   - Una soluzione simile non sarebbe fattibile in un ambiente con molti dispositivi perché ci sarebbero troppe subscribe da parte del *DataService* del CTH e si dovrebbe mantenere comunque un vettore di tutti i MAC address che sono presenti in quel momento. Nel caso il CTH si ricolleghi e non abbia quelle informazioni si troverà in difficoltà a gestire le disconnessioni.
2. Utilizzo di un topic generico con payload dinamico.
   - Schema:

     Topic : `MQTT/LWT`

     Payload : `#client-id`
   - Questo sarebbe lo schema ideale in quanto si deve effettuare un'unica subscribe da parte del DataService del CTH e si ottiene il nome del dispositivo caduto dal payload. Il problema è che non si può sviluppare una soluzione simile in Kura dato che l'LWT non è modificabile tramite API.
3. Utilizzo di un topic generico, con specializzazione del suffisso per il riconoscimento del dispositivo.
   - Schema:

     Topic : `MQTT/LWT/#client-id`

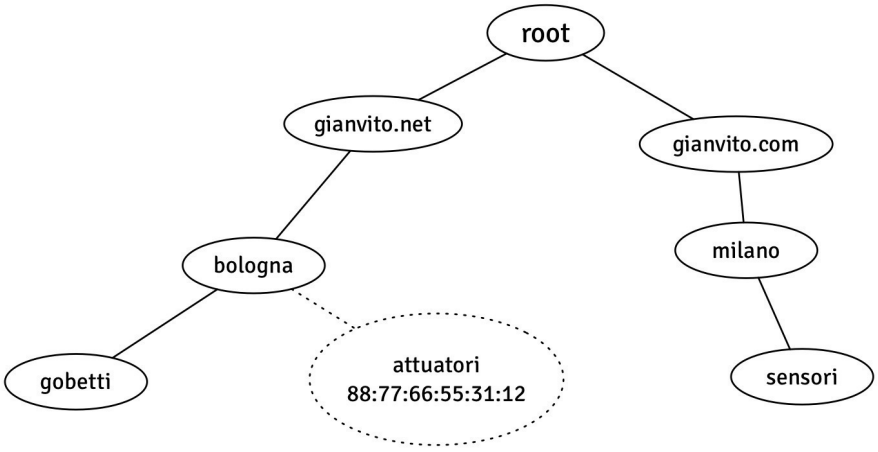
     Payload : nullo
   - Il *#client-id* sarebbe composto da due campi:
     1. Nome del percorso sulla gerarchia dei nodi

2. MAC address per l'identificazione

Prima di avviare il server dovremmo inserire solo il percorso all'interno della sezione MQTT della Web UI in quanto il MAC address viene riconosciuto e aggiunto automaticamente.
Il *DataService* del *CoAPTreeHandler* deve effettuare solo la subscribe verso `MQTT/LWT/#` (dove # è la wildcard che indica che dovranno essere presi in considerazione tutti i sottotopic).
Quando il *CoAPTreeHandler* riceve un messaggio LWT:

1. Effettua il parsing della stringa relativa al topic definito in precedenza
2. Trovare il dispositivo dato il percorso gerarchico
3. Controllare che sia effettivamente quello il nodo caduto attraverso un operazione di matching con il MAC address. Questo è un passaggio importante in quanto nell'aggiunta dei broker è stato scelto di considerare anche i duplicati che, essendo tali, hanno lo stesso percorso di almeno un altro nodo.

Ad esempio consideriamo il seguente schema:



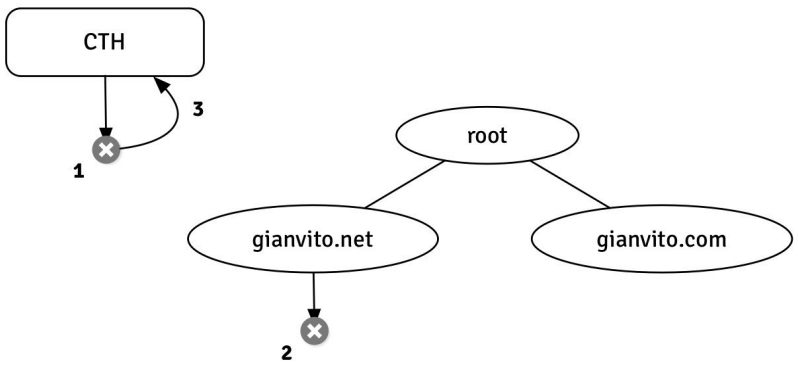
Ad un certo punto il nodo attuatori diventa non più disponibile. Allora il CoAPTreeHandler riceverà l'LWT dal broker MQTT e prima di tutto deve effettuare il parsing del topic del messaggio ricevuto:

1. Viene estratto il percorso *gianvito.net/bologna/attuatori*
2. Viene estratto il MAC address *88:77:66:55:31:12*

Una volta ottenute tutte le informazioni sul nodo, viene ricercato all'interno dell'albero e in caso di coincidenza di tutte le proprietà viene rimosso e vengono notificati gli altri broker interessati.

5.2.3.4. Failure del CTH

Come già accennato in precedenza, il CoAPTreeHandler è essenziale per gestire tutta la gerarchia di broker. Nel momento in cui non è più disponibile e avviene la caduta di altri nodi, è impossibile riuscire a ricavare quest'informazione successivamente.
Il problema non risiede nel fatto che il CTH non conoscerà tutti i dispositivi attualmente connessi in quanto, dopo ogni intervallo di tempo ogni broker invierà una root o parent request. Viene invece intaccata l'integrità delle informazioni presenti sugli attributi delle risorse presenti sui nodi figli.
Consideriamo un semplice esempio:



Nel momento in cui si disconnette CTH (**1**), se cade anche un nodo della gerarchia (**2**), al nodo radice non arriverà il messaggio di rimozione delle risorse del broker *gianvito.net*, dato che è il CTH a gestire questi scambi.
Nel momento in cui si riconnette (**3**), dato che le proprie Collection sono vuote, non ha l'informazione riguardo la disconnessione di *gianvito.net* e quindi non aggiornerà la root.

Ci sono diversi modi per poter gestire questi imprevisti:

1. Fare in modo che il CTH memorizzi tutta la struttura in memoria persistente e quando si riattiva, dopo essersi disconnesso, va a controllare tutti i nodi figli per verificare che siano realmente attivi. Il problema principale di questo approccio è che se un nodo non ha ancora effettuato la ri-connessione verso il CTH e viene rimosso dai linked nodes di un broker, non effettuerà poi l'aggiornamento verso il nodo parent fino a che non avvengono modifiche di proprietà alle risorse collegate in locale. Inoltre dato che dobbiamo memorizzare informazioni in memoria persistente, in un ambiente dinamico

è facile che diventino molto complicate da gestire. Naturalmente questo va ad impattare sulle prestazioni in quanto, nel caso peggiore, dovremmo scrivere su disco ogni volta che avviene un aggiornamento alla gerarchia.

2. Una seconda alternativa è quella in cui ogni nodo debba inviare una ResourceUpdate verso il nodo parent. Se trascorre un certo periodo di tempo e quel determinato nodo non ha ancora inviato l'aggiornamento, viene rimosso dai linked nodes. Il problema di questo approccio è che dobbiamo mantenere un timer per ogni collezione di attributi, vengono aggiunti altri attributi da memorizzare e si aumenta il numero di messaggi MQTT. Uno degli obiettivi del protocollo è quello di evitare scambi inutili nella gerarchia in modo tale da affidare ad ogni broker la gestione di un insieme di risorse e dato che la caduta del CTH non avviene così spesso, è inutile andare a prendere in considerazione azioni di contenimento dei danni molto complesse e difficili da gestire.
3. Il terzo approccio (implementato) è quello di avere un metodo di emergenza, avviato nel momento in cui il CTH si riconnette.

I passaggi sono:

1. Il CTH invia un messaggio di *CTH_ONLINE* su MQTT.
2. I nodi che attualmente sono connessi lo ricevono, bloccano il timer di update del parent attualmente presente e avviano l'esecuzione di un metodo specifico dopo un certo periodo di tempo, (impostabile dalla WebUI) in cui viene effettuato un soft reboot.

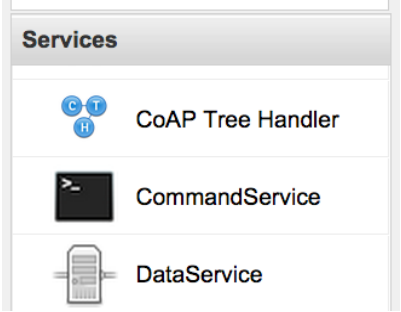
La procedura di soft-reboot consiste nella:

1. Rimozione di tutte le proprietà dei linked nodes collegati in precedenza.
2. Impostazione del parent corrente a null in modo tale che dopo una successiva richiesta, anche se viene restituito lo stesso riferimento, viene comunque propagata la Collection di attributi al nodo padre.

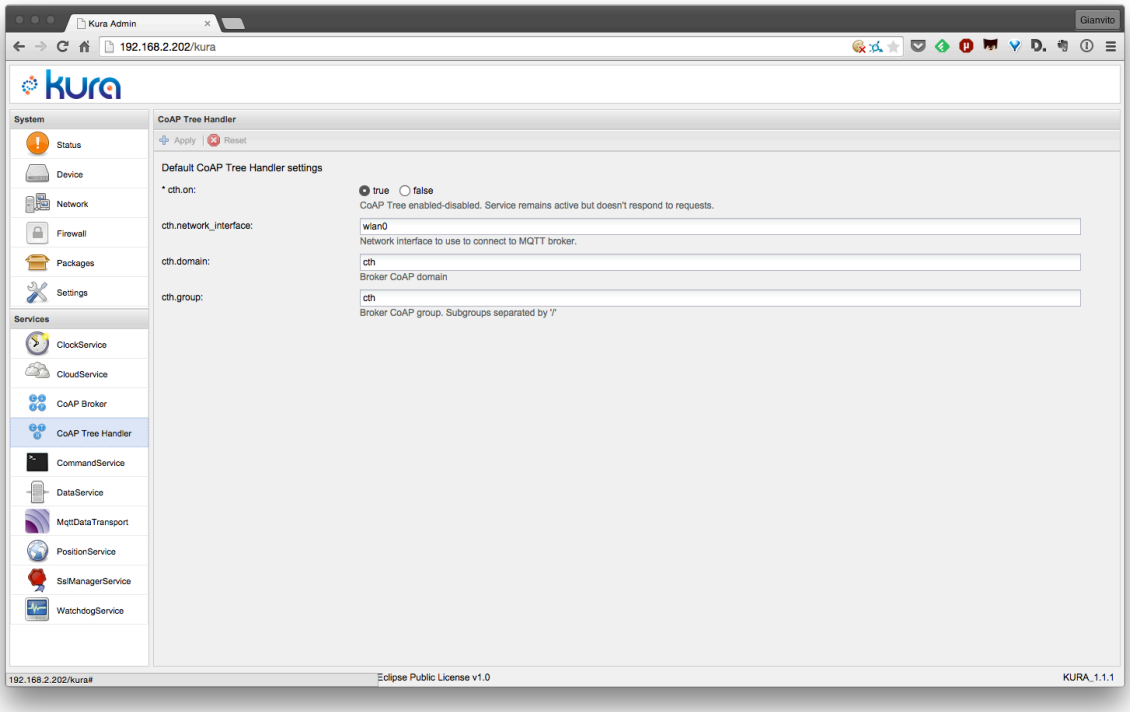
In questo modo otteniamo una gestione veloce, semplice e stateless di tutti i broker anche nel caso di caduta del CTH in quanto non abbiamo dati di gestione da memorizzare su disco e non ci sono vincoli temporali da dover seguire.
Permette inoltre di recuperare il funzionamento ottimale del protocollo nel caso in cui ci siano stati altri problemi che abbiano modificato lo stato degli attributi memorizzati.
Come detto in precedenza, il CTH deve essere il più stabile dei broker e quindi sarebbe inutile andare a rendere più complesso lo scambio di messaggi per gestire l'eventualità che questo nodo abbia una failure.

5.2.3.5. Configurazione nella Web UI

Per il CoAPTreeHandler è stato aggiunto l'elemento *metatype*, che permette di avere la possibilità di impostare proprietà a runtime attraverso l'interfaccia Web di Kura.
All'interno della lista dei Service è presente la seguente voce:



La schermata di configurazione invece è la seguente:



È possibile impostare:

- Se attivare automaticamente il CTH appena installato (utile nel caso in cui vogliamo avere solo la funzione di broker).

- L’interfaccia di rete da utilizzare.
- Il dominio e il gruppo del CTH.

5.2.4. Broker CoAP

Lo scopo principale di questo bundle è quello di implementare un supporto ad alta scalabilità per CoAP all'interno del framework Kura.

5.2.4.1. Utilizzo della Resource Directory

In un specifico momento più client possono richiedere risorse remote e quindi bisogna trovare il modo di conservare tutti i risultati.
È per questo motivo che è stata scelta la Resource Directory. Al suo interno possono essere memorizzate tutte le risorse CoAP che riceviamo dagli altri broker attraverso operazioni POST e dato che dopo un certo periodo di tempo scadono, verranno rimosse automaticamente.
La conseguenza positiva di questo comportamento è che se le risorse non vengono utilizzate non c'è neanche bisogno di effettuare operazioni di manutenzione.
La RD offre inoltre funzioni di ricerca con gli stessi attributi della Remote Query. Se più client effettuano diverse ricerche contemporaneamente, possono effettivamente trovare i risultati successivamente cercandoli nella Resource Directory con la stessa request usata in rd-remote.

5.2.4.2. Strutture dati

5.2.4.2.1. PropertyCollection

Per il corretto funzionamento del supporto, c'è bisogno di implementare una struttura dati che permetta di associare gli attributi di una CoAPResource con l'insieme delle risorse che effettivamente li possiede.
All'interno della *PropertyCollection* è presente una struttura di tipo:

Map<LinkAttribute, Set<GeneralProperty>>

La chiave scelta è di tipo *LinkAttribute*. Quest'oggetto permette di creare una coppia nome-valore di un attributo. È presente nell'implementazione della Resource Directory e contiene al proprio interno tutti i metodi che permettono di effettuare il parsing dell'URL di richiesta

sulla RD e in genere di CoAP. Non possiede però la capacità di gestire domini e gruppi che sono stati inseriti nel supporto e sono funzionalità che sono state inserite ad-hoc.
La parte invece relativa alle *GeneralProperty* permette di associare ad un determinato attributo sia collegamenti a risorse locali che a nodi esterni.
Questa caratteristica è utilizzata quando riceviamo update su risorse presenti sui nodi figli e dobbiamo aggiornare la Collection locale (questo avviene comunque solo se ci sono attributi che non sono già presenti nella struttura dati locale).
La *GeneralProperty* viene estesa poi in:

1. *ResourceProperty* : collegamento effettivo alla risorsa locale (presente come oggetto interno).
2. *NodeProperty* : collegamento al nodo Kura che possiede almeno una risorsa con quell'attributo (presente come oggetto KuraNode all'interno).

Sono poi presenti tutti i metodi necessari per la propria gestione e anche per la corretta serializzazione in caso di strumenti di debug esterni.

5.2.4.2.2. LinkedNodeCollection

Per gestire correttamente l'arrivo di aggiornamenti sulle risorse presenti sui nodi figli abbiamo bisogno di una struttura che memorizzi sia il broker che la lista di attributi posseduti dalle risorse presenti su di esso.
La struttura è composta in questo modo:

Map<KuraNode, AttributeCollection>

Ogni volta che arriva un update dal basso lo memorizziamo nella *LinkedNodeCollection* controllando prima però il numero di versione dell'*AttributeCollection* inviata. Questo per assicurarci che localmente siano sempre presenti gli attributi aggiornati dei nodi figli.

5.2.4.2.3. *AttributeCollection*

Questa struttura è molto semplice e permette la memorizzazione di un insieme di *LinkAttribute*. È stata aggiunta in modo tale da disaccoppiare l'idea stessa dell'insieme di attributi dalla sua implementazione.
È composta in questo modo:

Set<LinkAttribute>

Ha al suo interno un numero di versione che viene utilizzato poi dalla *LinkedNodeCollection* per capire se vale la pena o meno effettuare un update.

5.2.4.2.4. *BrokerCollection*

Questa struttura è il vero e proprio cuore di tutto lo scambio di risorse con richieste e risposte. Permette di gestire in maniera appropriata le due Collection definite in precedenza (*PropertyCollection* e *LinkedNodeCollection)*.
Contiene al suo interno tutti le funzioni richiamate dai metodi di callback, alla ricezione dei messaggi MQTT, tra cui:

- *handleResourceUpdate* : permette di effettuare tutta una serie di operazioni (documentate all'interno del codice) per gestire l'arrivo di nuove *AttributeCollection* per i nodi figli.
- *handleResourceRequest* : come la precedente funzione ma permette di capire quali sono le risorse che fanno match con gli attributi di ricerca. Quando sono presenti *domain* o *group* come keyword, vengono prese delle precauzioni particolari per poterle gestire correttamente.

5.2.4.3. Topic e QoS

A differenze del CoAPTreeHandler, un server CoAP può anche ricevere richieste riguardanti le risorse remote.
I topic interessati dal Publisher del server CoAP sono:

- Relativi ai nodi radice:
  - *ROOT_REQUEST_TOPIC* : è il topic in cui viene richiesto il riferimento alla root da impostare poi successivamente come parent locale.
  - *ROOT_ON_NODE_UPDATED_TOPIC* : su questo topic vengono pubblicati messaggi relativi all'aggiornamento del parent effettuato sul nodo (in questo caso una root, inviata precedentemente dal CTH).
- Relativi ai nodi di livello superiore a 0:
  - *PARENT_REQUEST_TOPIC* : topic di richiesta dell'invio del parent per un nodo qualsiasi della gerarchia

    - *PARENT_ON_NODE_UPDATED_TOPIC* : topic che ha la stessa funzione del *ROOT_ON_NODE_UPDATED* ma associato a nodi qualsiasi. Nel caso in cui manchi un livello e quindi ci si trovi in una situazione di emergenza possono anche essere inviati riferimenti al nodo radice su questo topic.
    - *NODE_REMOVE_TOPIC* : topic utilizzato dal CoAP Server per indicare al CTH che sta per disconnettersi. Viene utilizzato al posto del LWT per i casi in cui si ha una rimozione di tipo graceful.
- Legati allo scambio di risorse:
    - *RESOURCE_UPDATE_TOPIC* : topic specifico di un nodo, utilizzato da altri broker (che evidentemente sono stati associati come figli) per inviare gli attributi delle risorse (con i relativi collegamenti).
    - *RESOURCE_QUERY_REQUEST_TOPIC* : topic utilizzato per inviare richieste di risorse remote.
    - *RESOURCE_QUERY_RESPONSE_TOPIC*: quando invece viene ricevuta una remote resource query viene controllata la *PropertyCollection* locale, se ci sono dei nodi da restituire vengono aggregati in un insieme e si invia un messaggio MQTT su questo topic con la struttura appena creata.

I topic interessati dal Subscriber del server CoAP invece sono:

- Relativi ai nodi radice:
    - *ROOT_ADDED_TOPIC* : utilizzato dai nodi di livello 1 per conoscere se è stato o meno aggiunto un nodo radice.
    - *ROOT_NOT_AVAILABLE_TOPIC* : in questo caso il topic viene utilizzato per sapere se è stato rimosso o meno il nodo radice.
    - *ROOT_RESPONSE_TOPIC* : questo è il topic privato su cui il nodo riceve messaggi (con il nuovo parent all'interno) dal CTH riguardo ad una precedente *root_request*.
- Relativi ai nodi di livello superiore allo 0:
    - *NODE_ADDED_TOPIC* : topic privato su cui si riceve un messaggio di aggiunta di un nodo di interesse. Non è utilizzato al momento ma può esserlo in futuro, in caso di estensioni al protocollo.
    - *PARENT_RESPONSE_TOPIC* : topic su cui il nodo riceve la risposta riguardo la disponibilità di un parent. Può essere restituito anche il parent già impostato in precedenza.

- *PARENT_AVAILABLE_TOPIC* : non è utilizzato al momento ma può essere utile nel caso si estendesse il protocollo e si rendessero anche i nodi aware della presenza di altri broker.
- *PARENT_NOT_AVAILABLE_TOPIC* : topic specifico del nodo che permette di informarlo che il parent impostato in precedenza non è più disponibile. Se è effettivamente quello correntemente impostato allora verrà rimosso.
- *CHILD_NOT_AVAILABLE_TOPIC* : topic specifico del nodo, che permette di informarlo che uno dei figli non è più disponibile e che quindi deve rimuoverne le risorse o comunque tutte le informazioni che ha memorizzato su di esso.

• Legati allo scambio di risorse:
- *RESOURCE_UPDATE_TOPIC* Su questo topic in maniera simile a quello specificato in precedenza vengono ricevuti tutti gli aggiornamenti sugli attributi dei nodi figli e in base al numero di versione dell'*AttributeCollection* si decide se aggiornare o meno il riferimento locale.
- *RESOURCE_QUERY_REQUEST_TOPIC* : su questo topic si ricevono le Resource query provenienti da broker esterni.
- *RESOURCE_QUERY_RESPONSE_TOPIC* : grazie a questo topic riceviamo le risposte a Resource query effettuate in precedenza da questo nodo. Si tratta di una lista di altri nodi da contattare. Quindi non resta che procedere nell'invio di altre richieste verso i broker che fanno parte dell'insieme ricevuto.

Rispetto al caso del CoAPTreeHandler vengono definiti due tipi di QoS da utilizzare:

1. Scambio di resource query tra nodi: "at least once" (quindi Acknowledge delivery). Il motivo principale di questa scelta è quello di ottenere prestazioni soddisfacenti senza andare a congestionare molto la rete con un livello di qualità di servizio maggiore. Anche se vengono ricevuti messaggi duplicati non c'è nessun problema dal punto di vista dell'aggiornamento delle risorse perché si controlla il numero di versione prima di procedere. Nel caso invece di Resource query simili non abbiamo la modifica della semantica delle operazioni in quanto viene semplicemente inviata di nuovo la query. Se sono state già effettuate operazioni di POST sulla RD remota, non avremo duplicati perché è in grado di gestirli in modo nativo.
2. Scambio di messaggi tra nodi e CTH: "exactly once" (quindi Assured delivery). Caso ben diverso è quello che avviene sul CTH e dato che comunque è molto importante

mantenere aggiornate le informazioni sulla gerarchia vale la pena utilizzare il livello di qualità di servizio massima che offre il protocollo MQTT.

5.2.4.4. Connessione all'albero

5.2.4.4.1. *Aggiornamenti dei nodi parent e connessione all'albero*

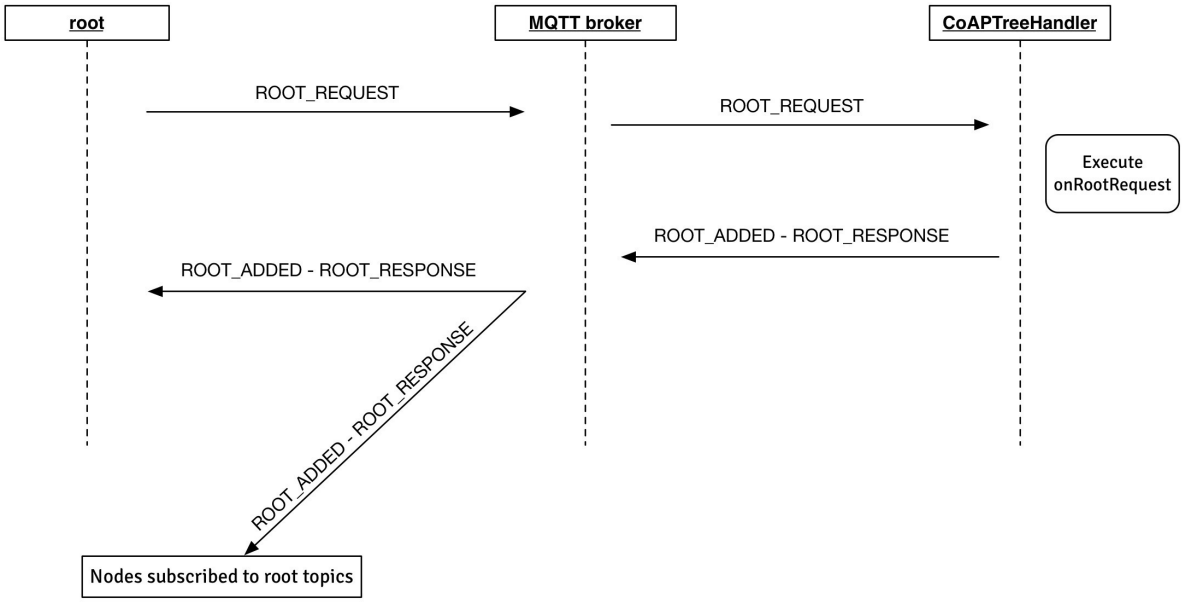
Come già accennato in precedenza i nodi parent vengono modificati in base all'appartenenza ad uno dei livelli della gerarchia.
Innanzitutto quando viene effettuata la richiesta (sia root che parent):

- Se ci si trova a *livello 0* ed è presente già un nodo root gli viene restituito semplicemente il riferimento al nodo radice attuale. Se invece non è presente un nodo root viene aggiunto il nodo che ha effettuato la richiesta come radice e vengono notificati tutti i broker interessati.
- Se ci si trova a *livello 1* ed è già presente un nodo root gli viene restituito il riferimento al nodo radice. Se invece non è presente un nodo root riceve un messaggio MQTT di *NotAvailableRoot*. Il motivo è che se ha effettuato una richiesta successiva (e il nodo lo aveva già impostato come parent), significa che quella radice non è più presente e quindi deve essere rimossa dalle proprietà del broker.
- Se ci si trova a *livello 2* si cerca prima di scoprire se il nodo è stato già inserito precedentemente. Se è un duplicato ed è presente un potenziale nodo parent e non è uguale a quello che è correntemente impostato bisogna inviarlo sulla coda specifica del nodo (evitando così di congestionare la rete in caso contrario). Se invece non sono presenti nodi parent compatibili con le proprietà gerarchiche del nodo non viene eseguita nessuna azione. Se invece non è un duplicato bisogna aggiungerlo alla collection e cercare un nuovo nuovo parent prima di inviarglielo.

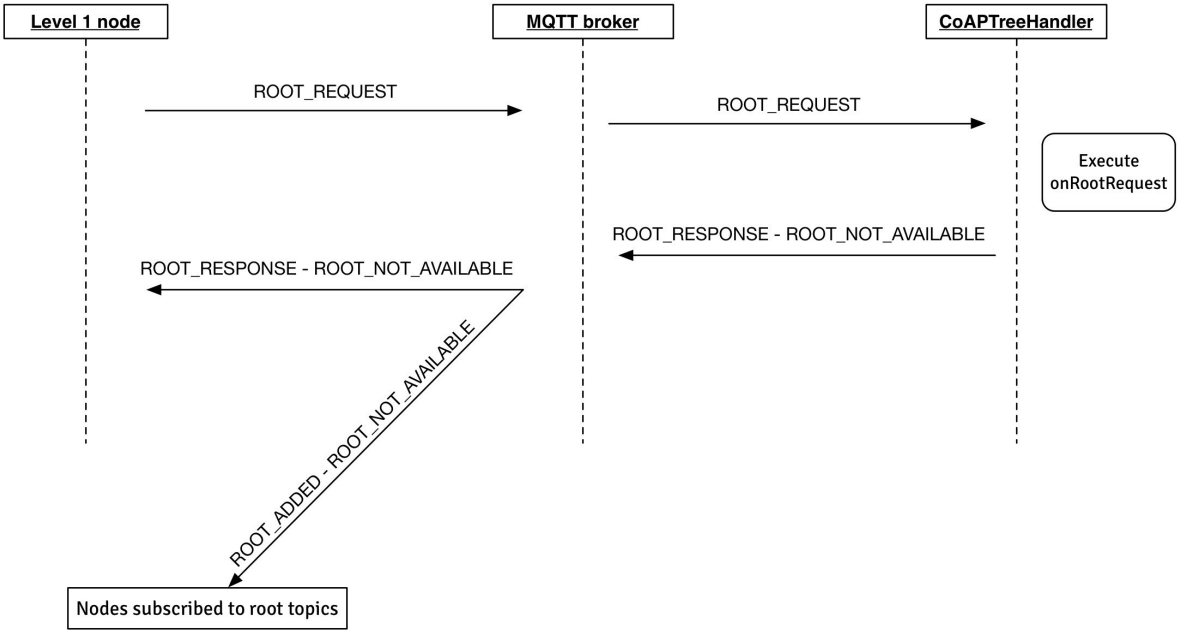
Prima che comunque il CTH aggiorni la propria struttura dati, deve ricevere un messaggio MQTT dal nodo notificato in precedenza. Questo messaggio può essere di tipo *RootUpdateOnNode* o *ParentUpdateOnNode*. Come si può intuire dal nome sono entrambi messaggi di ACK per l'operazione di impostazione del parent avvenuta correttamente.
La *connectToTree* presente nel server CoAP riconosce il livello del nodo dopo la lettura delle informazioni dalla WebUI e se è di *livello 0* o di *livello 1* viene inviata un *rootRequest* mentre se è di *livello 2* viene inviata una *parentRequest*.

Queste sono le interazioni che avvengono per l'inserimento di un nodo all'interno della gerarchia:

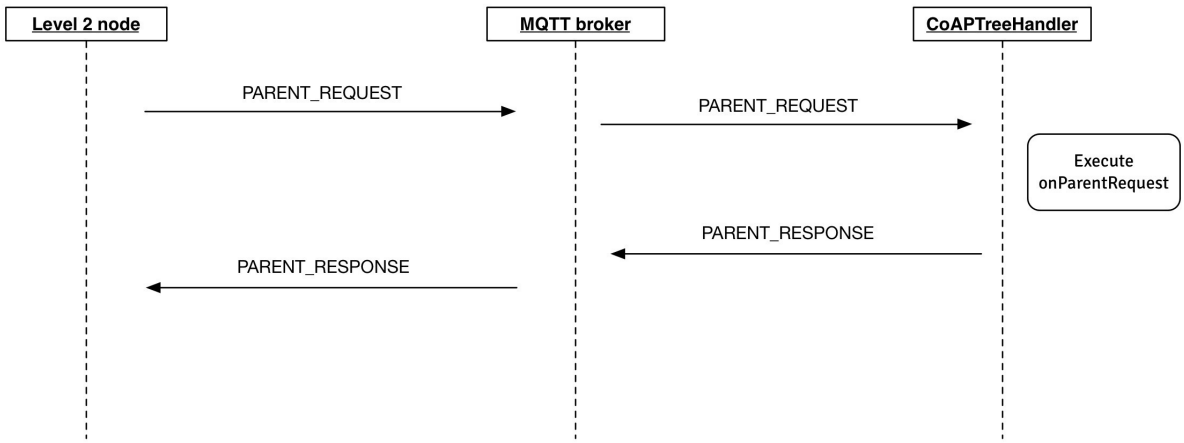
- Root request da un potenziale nodo root



- Root request da un nodo di livello 1



- Parent request da un nodo con livello maggiore di 1



Come si può notare dalla figura, la risposta ad una *PARENT_REQUEST* è specifica del nodo e quindi verrà inviata su una coda privata.

5.2.4.4.2. *Disconnessione dall'albero*

Una volta connesso all'albero, un nodo può uscire dalla gerarchia in due modi:

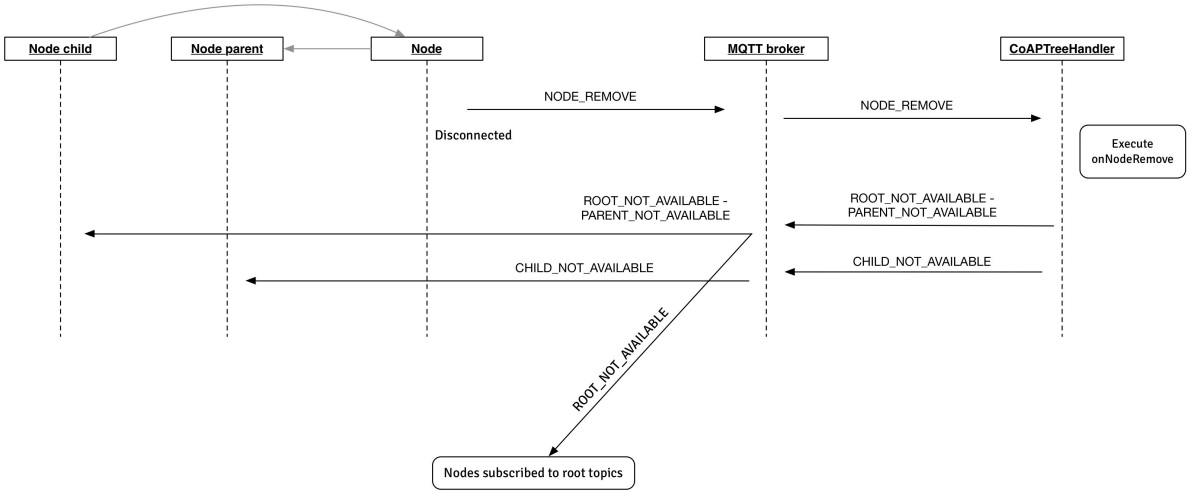
1. In modo graceful, utilizzando il metodo *disconnectFromTree* che è richiamato automaticamente quando si va ad eseguire la *deactivate* del Service o da terminale oppure da WebUI. Viene pubblicato un messaggio di rimozione del nodo sul CTH che poi eseguirà le operazioni specifiche sulle Collection locali.
2. In modo brusco, in cui viene chiamata in causa la funzionalità di MQTT chiamata Last Will and Testament. Viene pubblicato automaticamente un messaggio MQTT su una coda ben precisa definita seguendo il pattern specificato nella sezione CTH.

Quando il CoAPTreeHandler riceve il messaggio LWT:

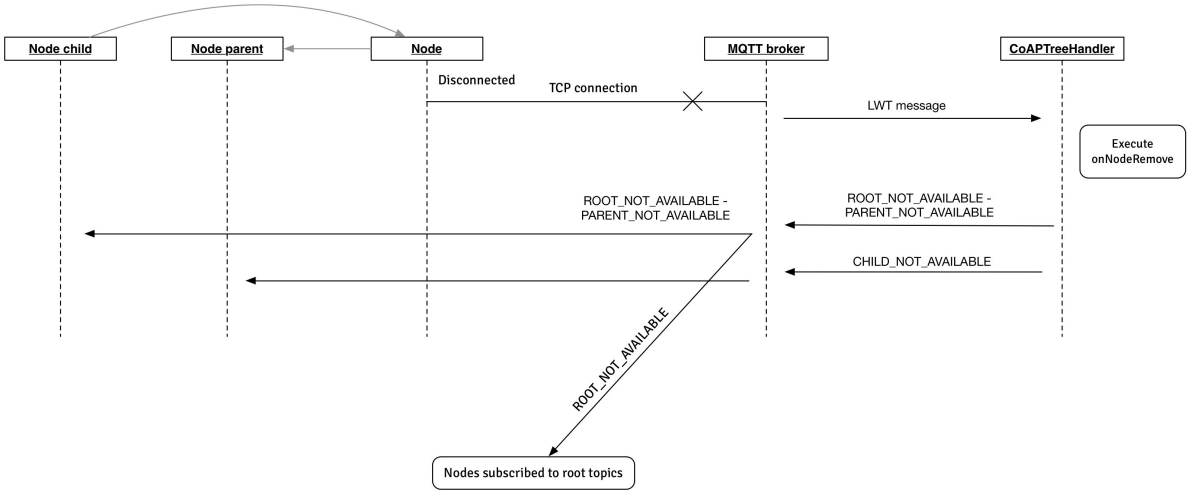
1. Ne effettua il parsing.
2. Riconosce sia l'indirizzo di rete che l'indirizzo MAC.
3. Effettua una ricerca all'interno della propria Collection e rimuove il nodo trovato.
4. Avverte i nodi interessati che il broker caduto non è più disponibile.

Queste sono le interazioni che avvengono per la rimozione di un nodo dalla gerarchia:

- Disconnessione normale



- Disconnessione con utilizzo di LWT



Stesso comportamento rispetto al caso precedente, con la differenza che il CoAPTreeHandler riceve il messaggio LWT sui cui effettuare il parsing per il riconoscimento del nodo caduto.

5.2.4.5. Risorse CoAP integrate con il broker

Il primo tipo di risorsa utilizzabile si chiama *BrokerResource*. Affinché si possa ottenere un'integrazione con le proprietà interne del server CoAP è possibile estendere la classe principale di Californium chiamata *CoapResource*. A questa classe sono stati aggiunti gli attributi:

1. Dominio
2. Gruppo

3. Data di creazione

Inoltre è presente un metodo richiamabile in fase di attivazione del Service implementato, attraverso il quale vengono aggiornati automaticamente dominio e gruppo.
Si tratta comunque di una classe minimale. Vengono mantenute tutte le caratteristiche della *CoapResource*, con il vantaggio dell'integrazione automatica con le proprietà del broker.

È presente poi la *ConcurrentBrokerResource*. È l'equivalente della precedente, ma estende la *ConcurrentCoapResource* in modo tale da non considerare il thread del parent per l'esecuzione delle operazioni, ma si può definire il proprio modello di concorrenza
Una *ConcurrentCoAPResource* definisce il proprio Executor e quando arriva una richiesta (anche verso risorse child che non definiscono il proprio thread) verrà eseguita all'interno di esso.

La *ExtendedResource* estende la *BrokerResource* e aggiunge un valore privato su cui effettuare le operazioni REST.

La *TimedCoAPResource* estende la *ConcurrentBrokerResource* e permette di evitare l'esecuzione di operazioni dopo un certo periodo di tempo.
Utilizza l'attributo *creationDate* per sapere se il metodo è eseguibile o meno (senza l'utilizzo di timer, on-demand).

Esempio di implementazione di una risorsa

```
/**
 * Example of a temperature sensor with ExtendedResource.
 *
 * @author Gianvito Morena
 */
public class TemperatureResource extends ExtendedResource{
     private static final Logger logger =
                     LoggerFactory.getLogger(TemperatureResource.class);
     private static String resourceName = "temperature";
     private String resourceType = "temp";
     private transient CoAPServer cs;

     // Observe test
     private int value = 10;
```

```
private transient Timer timer = new Timer();

public TemperatureResource(){
     super(resourceName);
     this.setResourceType(resourceType);
     this.addAttribute("position", "bologna");
     this.addAttribute("p1", "4000");
     this.addAttribute("p2", "10000");
}

protected void activate(ComponentContext componentContext){
     // If we want to add domain and group attributes
     this.updateDomainAndGroup(cs);
     cs.addResource(this, ExtendedResource.class);
     timer.scheduleAtFixedRate(new ValueTimer(), 5000, 5000);
}

protected void deactivate(ComponentContext componentContext){
     cs.removeResource(this, ExtendedResource.class);
}

@Override
public void handleGET(CoapExchange exchange){
     exchange.respond(value);
}

protected void setCoAPServer(CoAPServer cs){
     this.cs = cs;
}
protected void unsetCoAPServer(CoAPServer cs){
     cs = null;
}

class ValueTimer extends TimerTask{
     public ValueTimer(){
          super();
     }

     @Override
     public void run() {
          if(value == 10){
               value = 15;
               changed();
```

```
                }
                else{
                      value = 10;
                      changed();
                }
          }
     }
}
```

I passaggi principali sono quelli evidenziati in grassetto:

1. Prima di tutto bisogna creare un nuovo Plug-in Project (come per la realizzazione di qualsiasi Service in Kura).
2. Aggiungere nel file *Component Definition* l'utilizzo del CoAP Service con l'impostazione tramite i metodi *setCoAPServer* e *unsetCoAPServer.*
3. Richiamare in fase di attivazione il metodo *updateDomainAndGroup* che imposta automaticamente gli attributi domain and group dal CoAP Server e il metodo *addResource* che aggiunge la risorsa al server, rendendola così disponibile alle chiamate sia locali che remote.
4. Si possono implementare poi i metodi per la gestione delle operazioni REST.

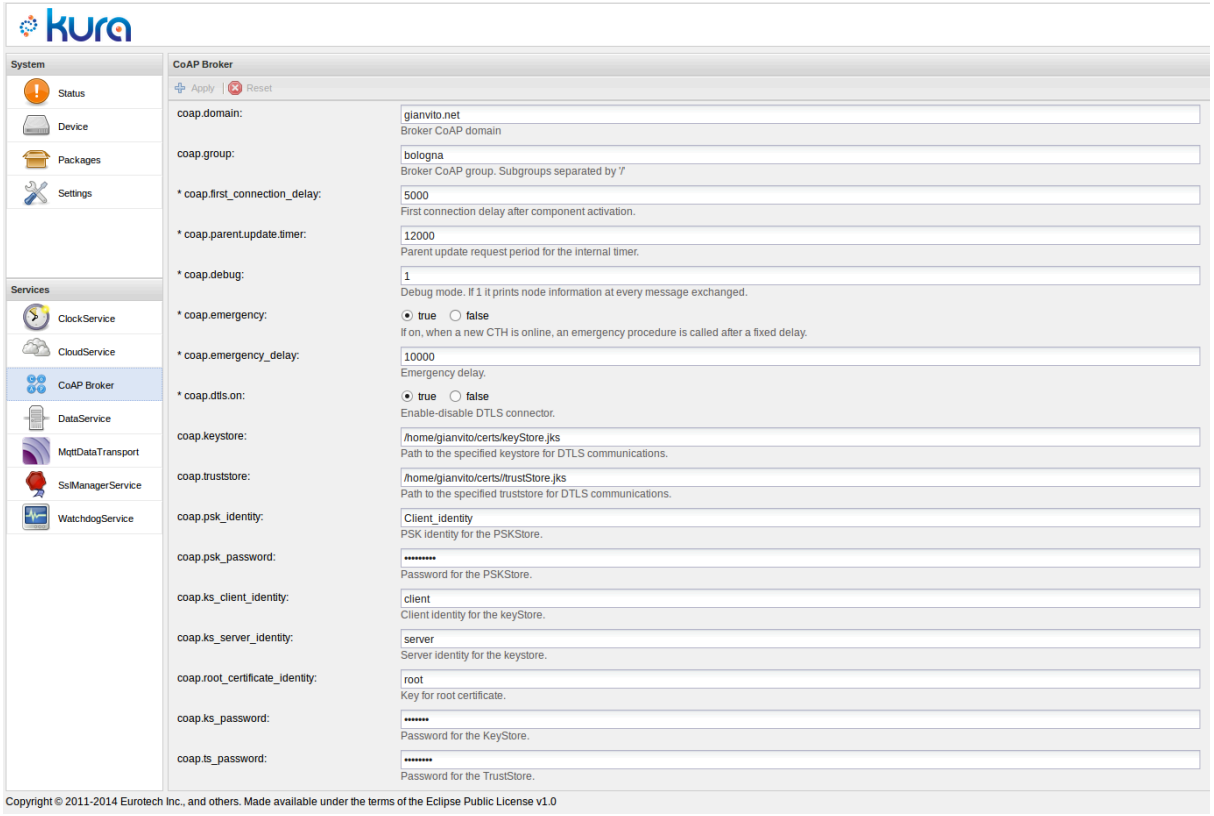
5.2.4.6. Aggiunta del supporto DTLS

Se il supporto va a considerare informazioni di tipo pubblico e quindi non ci sono problemi di sicurezza per quanto riguarda l'accesso diretto ai valori dei sensori.
Se il supporto viene utilizzato in ambito privato bisogna comunque proteggere le informazioni, soprattutto nel caso ci siano attuatori su cui possono essere eseguite operazioni di POST, PUT o DELETE.
Il problema è facilmente risolvibile aggiungendo il supporto DTLS per consentire l'esecuzione delle operazioni REST solo a chi ha le credenziali di accesso.

Il supporto DTLS per il progetto Californium è contenuto nel subproject Scandium:

Link: https://github.com/eclipse/californium.scandium

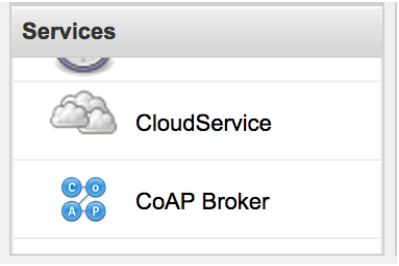
Per avere un funzionamento dinamico del server CoAP (con e senza DTLS) sono state aggiunte diverse voci all'interfaccia di configurazione in modo tale da consentire l'utilizzo di diversi keystore e truststore cambiando semplicemente il path e le password associate.
Il risultato è il seguente:



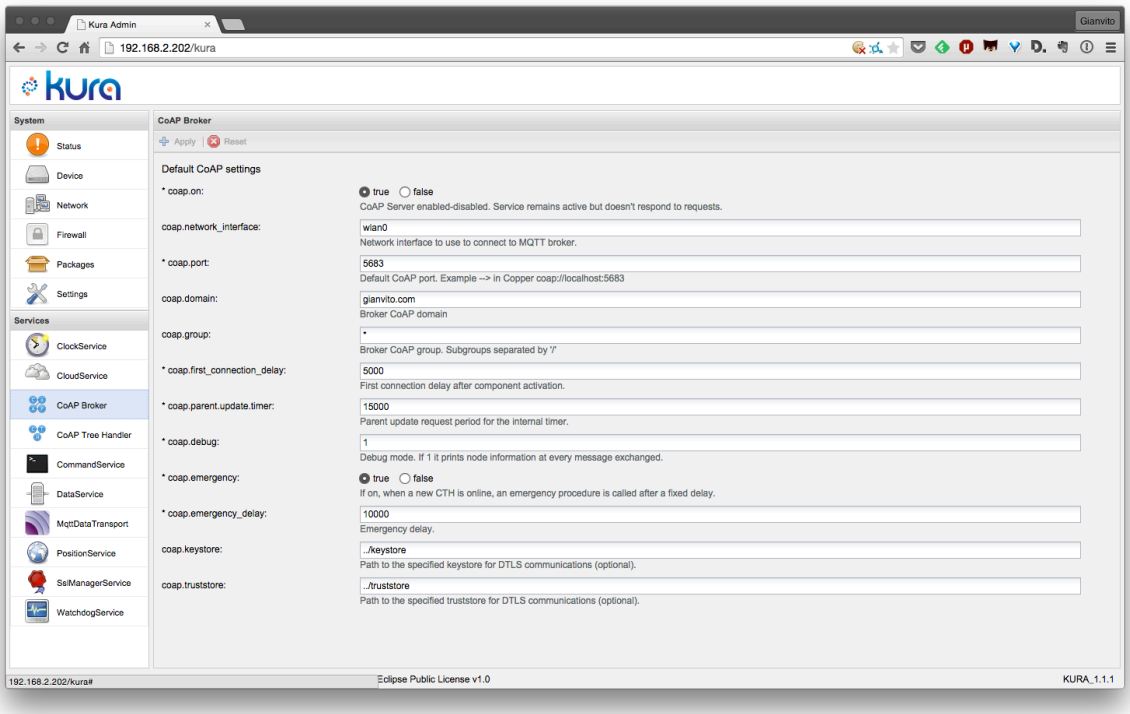
Dato che Scandium utilizza un tipo di logger differente rispetto a quello di Kura, per avere una visione generale di quello che succede, almeno in fase di implementazione, è necessario andare a sostituire le chiamate con quelle del logger utilizzate da Kura.
In questo modo sono emersi diversi requisiti riguardo l'utilizzo di *javax.crypto* (da importare opportunamente attraverso il file di MANIFEST).
Una volta risolti tutti i problemi iniziali si è proceduto all'organizzazione del supporto interno per avere flessibilità massima di configurazione: a questo proposito sono state aggiunte le classi *CoAPDTLSOptions* e *CoAPDTLSBase* che permettono di racchiudere tutte le opzioni e le informazioni riguardo il Connector DTLS da utilizzare in concomitanza con il server CoAP.

5.2.4.7. Configurazione nella Web UI

È stato aggiunto l'elemento metatype, in modo tale da poter impostare proprietà a runtime attraverso l'interfaccia Web di Kura.
All'interno della lista dei Service è presente la seguente voce:



Cliccandoci ci troviamo di fronte la schermata di configurazione:



Possiamo impostare:

- Se attivare automaticamente il broker appena installato (utile nel caso in cui vogliamo avere solo la funzione di CTH).
- L'interfaccia di rete da utilizzare.
- Il dominio e il gruppo.
- L'intervallo di tempo dopo il quale il broker tenterà di connettersi alla gerarchia.
- L'intervallo di tempo dopo il quale si richiederà un nuovo parent al CTH.
- Il livello di debug nella stampa delle informazioni di esecuzione.
- Se attivare o meno il protocollo di emergenza nel caso di caduta del CTH.
- L'intervallo di tempo dopo il quale eseguire il metodo di emergenza.
- Il path per il keystore ed il truststore nel caso di aggiunta della parte DTLS.

5.2.5. Remote Query Resource

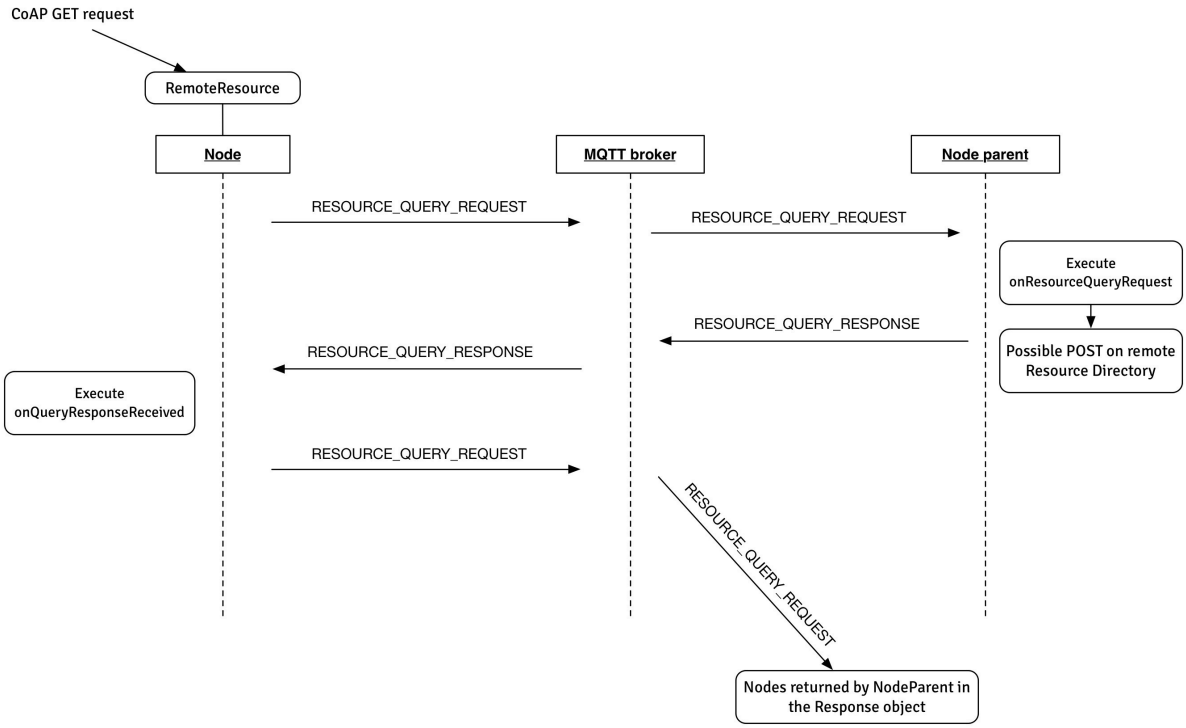
Per effettuare le query riguardanti le risorse presenti sui broker esterni, abbiamo bisogno di un punto d'appoggio che i client CoAP possono richiamare per popolare la Resource Directory. Per questo motivo si deve avere un collegamento tra la parte CoAP del server, su cui possono accedere quindi solo i client CoAP, e la parte invece che gestisce tutto lo scambio di messaggi MQTT.
È stato creato un path "*rd-remote*" su cui si va ad interagire con richieste di tipo REST. La risorsa in questione andrà poi ad utilizzare il publisher del server CoAP per effettuare resource query sui nodi direttamente accessibili.

Il motivo principale per cui non si è scelto di integrare questo Service all'interno di quello del server CoAP è che il disaccoppiamento fornisce flessibilità maggiore per quei casi in cui non si vuole permettere ai client di effettuare query remote. In questi casi basta non avviare il Service o non installarlo.
Il broker rimane comunque capace di agire sulle RD remote, nel caso in cui siano presenti risorse locali con gli attributi ricercati da client di altri nodi.

5.2.5.1. Funzionamento

Per poter effettuare query tra diversi broker bisogna creare un sistema di richieste e risposte che possa adattarsi anche in caso di più operazioni contemporanee.
L'obiettivo è quello di implementare un Service Kura che una volta avviato ci consenta di richiamare gli altri broker, seguendo la gerarchia, e popolare la Resource Directory locale, attraverso operazioni POST, con le risorse presenti sui server CoAP remoti.

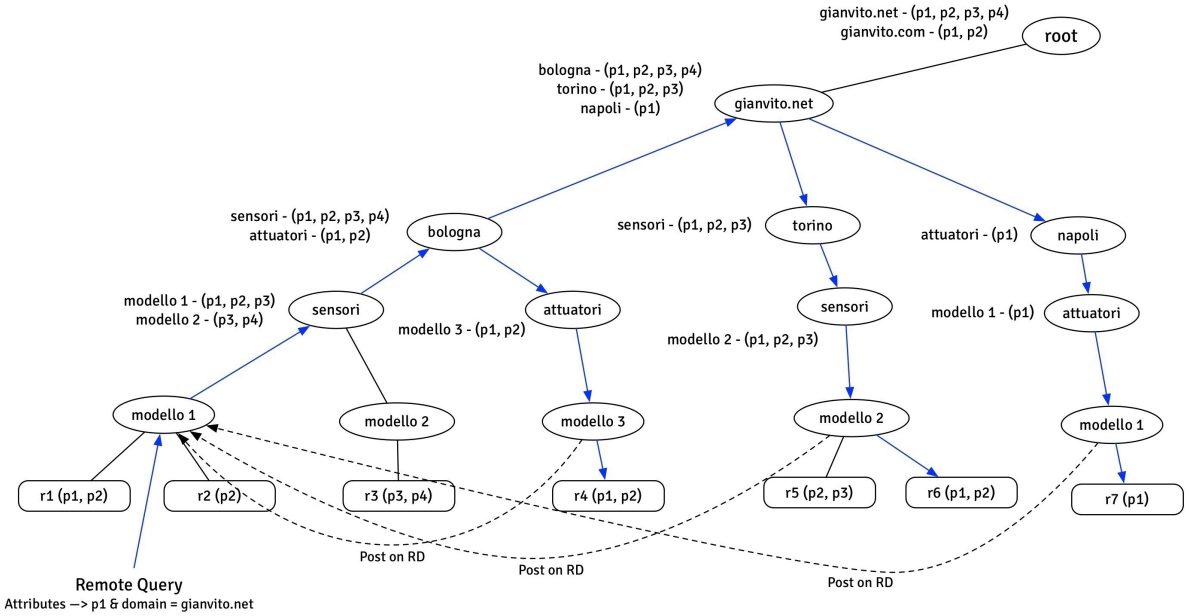
Queste sono le interazioni che avvengono per una semplice richiesta di una risorsa remota:



Come possiamo notare dallo schema temporale sul nodo che effettua la richiesta:

1. È stato prima di tutto avviato il Service *RemoteResource*.
2. Viene effettuata una richiesta da un client CoAP sul Service.
3. Il *CoAPPublisher* del nodo invia un messaggio MQTT di tipo *RESOURCE_QUERY_REQUEST* al nodo parent.
4. Seguendo lo schema, si arriva nel momento in cui dopo aver eseguito *onQueryResponseReceived* si devono inviare altri messaggi MQTT verso i broker restituiti precedentemente dal nodo parent. Si effettua quindi un'operazione di publish verso di essi impostando lo stesso tipo utilizzato per la prima richiesta (*RESOURCE_QUERY_REQUEST*).

Possiamo considerare il seguente esempio:



Abbiamo che:

1. Un client utilizza la *RequestRemoteResource* sul broker "*modello 1*" e la query contiene gli attributi *p1* (che può essere "*position = bedroom"* ad esempio) e *domain = gianvito.net*
2. *modello 1* controlla tra le risorse locali e aggiunge *r1* all'interno della RD interrogando (attraverso messaggi MQTT specifici) il proprio parent "*sensori*"
3. *sensori* non ha localmente risorse con gli attributi di ricerca (tranne quelli in *modello 1* che naturalmente non vengono restituiti dato che *modello 1* è il richiedente).
4. Si risale ancora la gerarchia fino a quando in *bologna* troviamo che in *attuatori* (o in uno dei suoi nodi figli) c'è almeno una risorsa con le proprietà ricercate (dovrebbe essere restituita anche sensori ma dato che è il parent di *modello 1* non viene considerata, nell'algoritmo si tiene conto del fatto che sensori si trova sullo stesso percorso di *modello 1*, quindi è già stato attraversato).
5. *modello 1* va ad effettuare la query su *attuatori* che restituisce il link di *modello 3* e una volta richiamato *modello 3*, il componente "*RemoteClient*" di quel nodo farà la POST delle risorse di interesse sulla RD di *modello 1*. *modello 3* non restituisce *attuatori* (come in precedenza attuatori non ha restituito *bologna*) perché, dato che non si trova sullo stesso path che porta a *modello 1*, *modello 3* non sarebbe mai stato contattato da *modello 1* se non fosse stato già interrogato *attuatori* (o *bologna* in precedenza).

6. Si prosegue così nella restituzione di nodi e POST sulla RD fino a quando non terminano le risorse da inserire in *modello 1*.
7. Viene richiamato *gianvito.net* (restituito precedentemente da *bologna*) che a sua volta restituisce *torino* e *napoli*. Non restituisce *bologna* per la regola definita in precedenza e cioè che evita di restituire nodi che siano più generici del mittente se si trovano lungo il path che congiunge il sender al nodo attuale. Infatti in questo caso bologna si trova tra *modello 1* (il sender) e *gianvito.net* (il nodo che sta analizzando quelli da restituire) e non saremmo mai potuti arrivare a *gianvito.net* se non fossimo passati per *bologna*. È inutile, quindi, restituirlo.
8. Si prosegue così a interrogare *torino* e *napoli* che restituiscono, rispettivamente, *sensori* ed *attuatori*.
9. *modello 1* effettua la query su *…/torino/sensori/modello 2* e *…/napoli/attuatori/modello 1* che a loro volta effettuano la POST sulla RD di *modello 1*.

6. Valutazioni sperimentali

In questo capitolo verranno effettuati vari test sul supporto in relazione ai requisiti definiti nella parte introduttiva del documento.
Inoltre verranno specificate le varie ottimizzazioni applicate alle librerie e verranno confrontate con le implementazioni attuali in modo tale da evidenziarne il miglioramento.

6.1. Indicatori di performance

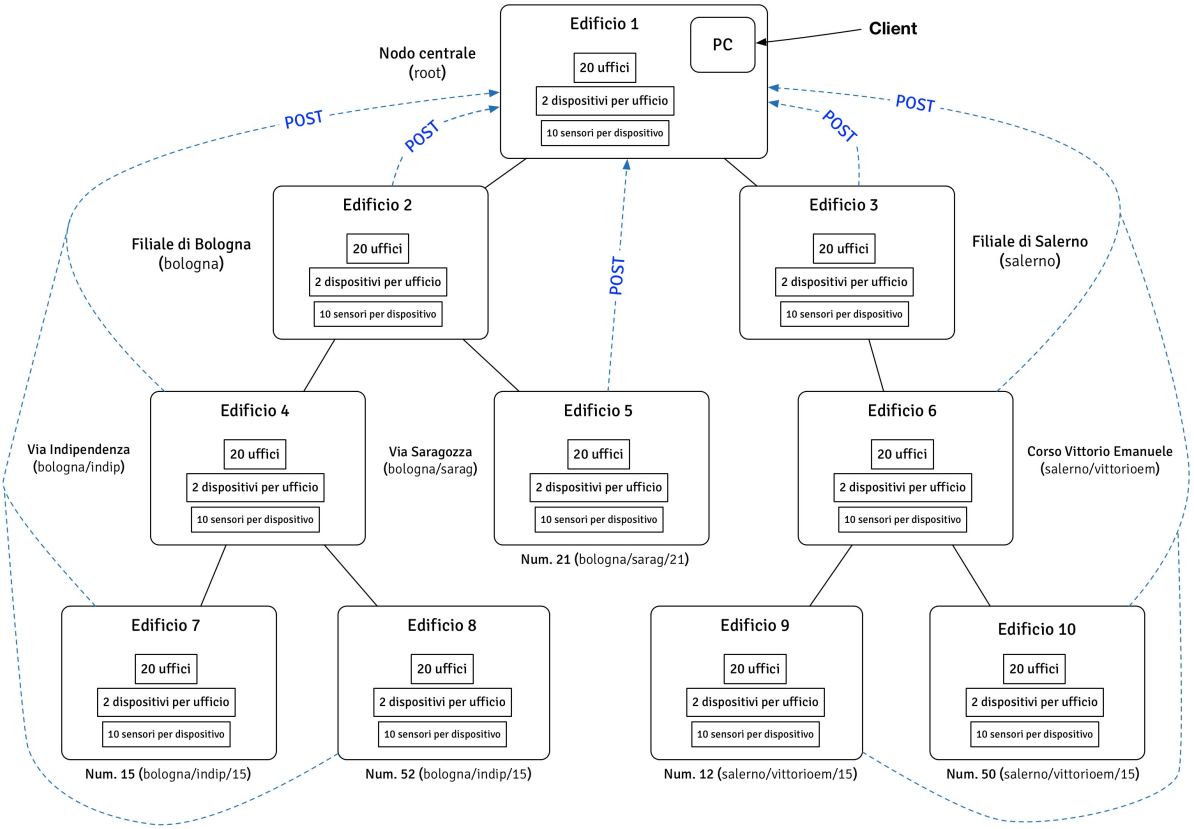
Prima di effettuare qualsiasi test bisogna fare delle considerazioni sui vari aspetti che incidono sul comportamento del supporto durante l'utilizzo.
Gli indicatori su cui si basano le considerazioni sulle prestazioni sono:

1. *Numero di richieste di risorse (intese come GET su rd-remote)* : è medio, dipende dall'applicazione ma non è molto alto in genere. Nel caso sia molto elevato è possibile riorganizzare la gerarchia in maniera tale da diminuire il numero di richieste.
2. *Numero di richieste sulle risorse (intese come operazioni GET, POST, …)* : di solito è molto alto. Con l'utilizzo di CoAP non c'è bisogno di implementare altre ottimizzazioni dato che è stato progettato proprio per queste circostanze.
3. *Numero di livelli dell'albero*: è medio e non raggiunge comunque un valore molto grande. Nel caso sia molto elevato basta effettuare una riorganizzazione della gerarchia.
4. *Numero di richieste sul CTH* : è medio-basso ma dipende dal numero di livelli dell'albero. È invece molto elevato in caso di numerose failure dei nodi.
5. *Scambio di risorse (proprietà)*: è medio-basso. Dipende dal tipo di richiesta e dal tipo di risorse (e cioè da quante proprietà hanno al loro interno).
6. *Numero di nodi*: è medio-alto. Sarà presente comunque un numero elevato di richieste iniziali se più nodi cercano di collegarsi direttamente.
7. *Carico sulla root*: è basso se i domini e i gruppi dei nodi sono ben distribuiti.
8. *Sicurezza tra broker e MQTT* : è assicurata dati che è possibile utilizzare SSL, già presente in Kura.
9. *Sicurezza tra broker* : è assicurata dato che utilizziamo MQTT per disaccoppiare la comunicazione e vengono fornite all'esterno solo le informazioni accessibili

pubblicamente (è presente solo il collegamento con il nodo parent).

10. *Sicurezza per accessi esterni sulle risorse* : il supporto è stato realizzato in modo tale che sia possibile aggiungere il connector DTLS al server CoAP se i dati sono privati.

Consideriamo un esempio tipo dove sono presenti una serie di edifici:



I dati sono:

- Numero di edifici: 10
- Numero di uffici per edificio: 20
- Numero di apparecchi per ufficio: 2
- Numero di sensori per apparecchio: 10

Il client effettua una resource query sul supporto installato in uno degli edifici (ad esempio nella root). Si vogliono analizzare i valori dei sensori da una determinata zona e magari raccoglierli per intervallo di tempo. Non si conoscono però i vari indirizzi e sarebbe troppo oneroso andare a configurarli in maniera statica. Quindi si vuole avere capacità di query limitate solo agli apparecchi realmente connessi.

L'obiettivo è conoscere l'indirizzo di tutti i dispositivi (esposti tramite risorse CoAP) di una particolare categoria. Nel caso medio sono presenti:

- 8 sensori di interesse in ogni apparecchio.
- Gli apparecchi sono installati in tutti gli uffici.
- In ogni ufficio sono presenti 2 apparecchi.

Quindi verrebbero effettuate:

- 2 operazioni di POST per ufficio.
- 2 * 20 = 40 operazioni di POST per edificio.
- 40 * 10 = 400 operazioni di POST in totale.

Il client da cui parte la chiamata conosce l'indirizzo web di un broker ad esso vicino e vi accede tramite Copper (estensione di Firefox per CoAP). Quindi tutte le POST verranno effettuate sulla RD di quel dispositivo.
Successivamente le richieste saranno effettuate tramite GET e POST sulla risorsa stessa e dato che CoAP è ottimizzato proprio per questo, non ci saranno problemi nell'esecuzione.

Il collo di bottiglia principale in questo caso è il *numero di operazioni POST su Resource Directory* che un nodo può ricevere ed elaborare in un tempo è abbastanza ridotto.
Non sono operazioni che verranno effettuate spessissimo ma è comunque importante, per l'usabilità del supporto, avere dei tempi ragionevoli di esecuzione.
Il numero di messaggi MQTT inviati ed elaborati incide in maniera proporzionale al numero di livelli dell'albero e al numero di nodi interessati. È comunque ridotto rispetto al numero di POST effettuati.

Considerando questo dominio di applicazione emergono quindi due operazioni principali che, se eseguite in maniera lenta, possono incidere fortemente sulle prestazioni:

- È importante ottenere un numero di POST su RD in tempi ridotti.
- È utile avere una veloce ricezione ed elaborazione dei messaggi MQTT.

Di seguito vengono elencate le varie prove per entrambe le situazioni, con riferimento ai miglioramenti ottenuti.

6.2. Configurazione di prova

La configurazione utilizzata per le prove consiste in:

- Macbook Pro Retina Mid 2014 con le seguenti caratteristiche:
  - Processore: *2,2 GHz Intel Core i7*
  - Memoria: *16 GB 1600 MHz DDR3*
  - Grafica: *Intel Iris Pro 1536 MB*
  - Scheda di rete: *AirPort Extreme con Broadcom BCM43xx*
  - Sistema operativo: *OSX Yosemite 10.10.3*
- Macchine virtuali *Parallels* versione 10.1.3
  - Sistema operativo *Xubuntu* versione 14.10
  - Per l'emulazione su PC è stato utilizzato il bundle *org.eclipse.kura.emulator* versione 1.1.0.
- Broker MQTT: *Mosquitto* versione 0.15 (build date 2013-08-23 19:23:43+0000).
- 2 *Raspberry Pi B+*:
  - Versione di Raspbian installata : *February 2015 (2015-2-16)*
  - Versione di Kura installata : *1.1.1*
  - Interfaccia di rete utilizzata : *wlan0*
- Router: *Asus RT-N56U* con firmware custom versione 3.4.3.9-091

6.3. Test su MQTT

Sono stati eseguiti dei test per quanto riguarda gli scambi MQTT in seguito a resource query sulla prima versione del supporto, senza nessun tipo di ottimizzazione.
I risultati per 1000 richieste di risorse (senza POST CoAP) tra Raspberry Pi connessi in Ethernet sono: **[203-220 secondi]**. L'esecuzione però si blocca molto prima con problemi del tipo:

- *Store exceeded* : troppi messaggi nello store
- *KuraTooManyMessagesInFiight* : troppi messaggi non ancora acked.

Dopo il lancio delle Exception appena descritte è impossibile continuare a comunicare con il broker a meno che non si riavvii la connessione MQTT.

6.3.1. Ottimizzazione della serializzazione

Uno dei primi miglioramenti che è possibile effettuare riguarda le operazioni più semplici coinvolte nell'invio dei messaggi MQTT.
MQTT è content-less, supporta solo l'invio di array di byte come contenuto del payload. Bisogna, quindi, trasformare gli oggetti creati appositamente per il supporto in array di byte per l'invio ed effettuare l'operazione opposta in fase di ricezione.

Partiamo dal modo in cui viene composto un *CoAPMessage*:

- Per ogni tipo di operazione vengono popolati i vari campi dell'oggetto in una maniera ben precisa.
- Prima di richiamare il *DataService* (servizio presente nel framework Kura per l'invio di messaggi MQTT), viene effettuata l'operazione di serializzazione.

Nella prima versione del supporto è stata usata la serializzazione standard di Java.
Il meccanismo di serializzazione in questo caso permette di ottenere la conversione in array di byte del *CoAPMessage*, ma l'operazione risultante risulta troppo lenta per gli scopi del supporto.
Il motivo principale è che i requisiti con cui è stata realizzata sono ben diversi rispetto a quelli richiesti dal supporto. I principali sono:

- Deve essere in grado di serializzare qualsiasi oggetto che implementa *Serializable*.
- Deve essere in grado di gestire in maniera corretta anche classi con versione differente

(avendo impostato la variabile *serialVersionUID*).

- Le performance, naturalmente hanno una priorità molto bassa.

La soluzione sarebbe quella di trovare un framework capace di gestire in maniera completa la serializzazione, ma con priorità alle prestazioni. Il nome di questo framework è *Kryo*.
Si è proceduti quindi alla scrittura di metodi che sfruttino Kryo al posto della serializzazione standard.
Si è tenuto conto del fatto che una istanza di Kryo non è thread-safe e quindi c'è bisogno di una Factory di istanze che possono essere borrowed per ogni serializzazione. Una volta completata l'operazione basta rilasciare l'istanza che potrà essere riutilizzata.
Si è passati quindi alla realizzazione dei metodi, aggiungendo anche lo Stream di compressione del messaggio.

A realizzazione ultimata sono stati rieseguiti i testi effettuati all'inizio e questi sono i risultati per 1000 richieste di risorse (senza POST CoAP) tra Raspberry Pi connessi in Ethernet: **[107-112 secondi]**. L'esecuzione però, anche in questo caso, si blocca molto prima con problemi del tipo:

- *Store exceeded* : troppi messaggi nello store
- *KuraTooManyMessagesInFiight* : troppi messaggi non ancora acked.

Come si può notare è stato dimezzato il tempo di esecuzione ma rimane comunque molto elevato per gli scopi prefissati.

6.3.2. DataService e DataTransportService

Analizzando i vari passaggi che avvengono tra l'invio del messaggio di richiesta di risorse e la ricezione è stato rilevato un altro collo di bottiglia: *DataService*.

Per spiegare il motivo per cui DataService limiti le prestazioni del supporto, bisogna analizzare il suo funzionamento. Prima di tutto contiene al proprio interno diverse strutture dati sviluppate per scopi ben diversi da quelli del supporto: è presente un *DataStore* che viene utilizzato per:

1. La memorizzazione dei messaggi nel caso in cui il dispositivo non sia collegato alla rete.
2. La gestione delle priorità e cioè, quando ci sono messaggi con priorità diversa si cerca

di inviare sempre quelli con il valore più alto (non è comunque garantita sempre la consegna a priorità più alta).

È presente un metodo per l'handling della congestione messaggi in-flight consente per ogni nuova sessione di rimuovere tutti i messaggi di cui non è stato ancora eseguito l'ACK da parte del broker. Consente inoltre di disconnettere il framework dalla sessione MQTT, se è attivata la congestion handling.

Ogni volta che viene inviato il comando di publish al DataService viene:

1. Memorizzato il messaggio nel DataStore interno.
2. Eseguita una submit di pubblicazione sull'Executor del DataTransportService interno.

La prima delle due operazioni è molto costosa e nel caso di molti messaggi pubblicati contemporaneamente, rende il dispositivo inutilizzabile:

- Sia dal punto di vista del comunicazione di rete dato che una volta che viene raggiunto il limite di memorizzazione bisogna solo riavviare il Service.
- Sia dal punto di vista dell'utilizzo del dispositivo stesso, in quanto il DataStore richiede diversi cicli di CPU che potrebbero essere utilizzati per svolgere altre operazioni.

Per gli obiettivi prefissati c'è bisogno di un Service che permetta di inviare direttamente i messaggi al broker MQTT senza operazioni intermedie (e soprattutto di memorizzazione). Dato che il DataStore utilizza *DataTransportService*, e dato che tutte le operazioni richieste sono già presenti nel servizio a livello inferiore, è possibile utilizzarlo per migliorare le prestazioni.

Analizzando come è stato sviluppato *DataTransportService* è possibile notare che:

1. Al suo interno vengono richiamati tutti i metodi di Paho (implementazione del protocollo MQTT).
2. Vengono sfruttati tutti i listener del client asincrono fornito da Paho.
3. Viene realizzato il whiteboard pattern per ottenere la propagazione degli eventi (pubblicazione del messaggio, conferma del messaggio da parte del broker, perdita con connessione con il broker, …). Per realizzare questo pattern viene sfruttato il ServiceTracker di OSGi che permette di ottenere tutti i riferimenti ai Service che implementano una determinata interfaccia o che estendono una specifica classe. Una volta ottenuti tutti i riferimenti si effettua la chiamata sui metodi implementati, in

modo tale da completare la propagazione dell'evento su tutti i listener.

Dopo aver integrato opportunamente questo Service all'interno del supporto sono stati rieffettuati tutti i test e questi sono i risultati per 1000 richieste di risorse (senza POST CoAP) tra Raspberry Pi connessi in Ethernet: **[24-26 secondi]**. In questo caso non abbiamo alcun tipo di errore.
Per 10000 richieste invece l'esecuzione si blocca dopo circa 6000 operazioni compiute con una Lost Connection con il broker MQTT.

Come possiamo notare il miglioramento rispetto all'utilizzo del DataService è di circa 5 volte con DataTransportService. Inoltre, dato che non viene utilizzato più il DataStore, sono stati rimossi tutti i problemi che riguardano la memorizzazione dei messaggi. È stato rimosso anche il problema della presenza di troppi messaggi in-flight dato che non devono essere prima salvati nel DB per essere inviati.
Il motivo principale del comportamento per 10000 richieste risiede nel fatto che, comunque il Raspberry Pi è un dispositivo con prestazioni non molto elevate e l'esecuzione di moltissime operazioni di rete contemporanee utilizzando Java come linguaggio di programmazione (che quindi viene eseguito su una JVM) comporta uno sforzo elevato sulla CPU.

Per come è stato strutturato MQTT, un client connesso al broker deve inviare in ogni intervallo di tempo un hearthbeat che indica che è ancora connesso. In caso contrario viene considerato come disconnected e viene pubblicato il messaggio di Last Will and Testament. In questo caso viene la CPU viene messa sotto pressione (anche senza altre operazioni) e non riesce ad eseguire l'invio del messaggio di *keepAlive* in tempo.

Una conferma di quanto appena descritto si può ritrovare nei test condotti su emulatore su PC. I risultati ottenuti (tra emulatori installati su 2 macchine virtuali con 512MB di RAM e 1 core assegnato) sono:

- Per 1000 richieste di risorse (senza POST CoAP): **[0.9 - 1.8 secondi]** senza alcun tipo di errore.
- Per 10000 richieste: **[9 - 10 secondi] s**enza alcun tipo di errore.
- Per 60000 richieste: **[31 - 33 secondi]** senza alcun tipo di errore.

6.4. Test sulla Resource Directory

Sono stati effettuati vari test per POST su Resource Directory ed i risultati ottenuti con l'implementazione predefinita di Californium non sono ottimali per l'applicazione considerata.

Considerando 500 operazioni di tipo POST su RD con client singolo in relazione ai vari dispositivi considerati:

- Raspberry Pi sia come mittente che come ricevente : **[96-102] secondi** con la presenza di diversi errori dovuti allo scadere del timeout per l'operazione REST.
- Raspberry Pi come mittente ed emulatore su PC come ricevente : **[12-13] secondi** e non sono presenti errori di POST.
- Emulatore su PC come mittente e ricevente : **[5-7] secondi** e anche in questo caso non sono presenti errori di POST.

Il motivo principale di questo comportamento è la non ottimizzazione dell'implementazione standard per un numero molto elevato di richieste contemporanee.
Analizzando il codice si può notare che:

- Vengono utilizzate strutture dati non efficienti per il parsing del payload CoAP. Sono presenti infatti espressioni regolari per il riconoscimento delle varie parti del messaggio. Vengono utilizzati gli Scanner per sfruttare le regex ed è risaputo che sono molto lenti dato che effettuano un numero di elaborazioni elevato per riconoscere i vari pattern all'interno della String.
- Non vengono sfruttati pattern di impostazione dei messaggi, naturalmente presenti nel CoRE Link Format. Ad esempio un link CoRE segue questo pattern:

  <Link_1>; nome_attributo=valore; nome_attributo=valore,
  <Link_2>; nome_attributo=valore; nome_attributo=valore,
  <Link_2>; nome_attributo=valore; nome_attributo=valore,

Quindi vale la pena ottimizzare per bene il parsing in modo tale da ottenere prestazioni accettabili nell'utilizzo del supporto.

6.4.1. Ottimizzazione dell'analisi del CoRE Link

Il path di una risorsa, secondo lo standard, potrebbe trovarsi in qualsiasi posizione del CoRE Link Format dato che tutti gli attributi sono separati da ";".
Questo però andrebbe a complicare il parsing per un dispositivo a capacità limitate che poi è il target device su cui si basa tutta la Internet of Things.
Quindi vale la pena fare questa supposizione: il path deve trovarsi sempre in prima posizione.

Data la presenza nel package Guava di Google di librerie ottimizzate per lo splitting di String in Java e dato che è stato incluso per lo sviluppo del supporto, vale la pena utilizzarlo.
La classe static che è possibile richiamare si chiama *Splitter*. Contiene al proprio interno diversi metodi che permettono di sviluppare in maniera semplice e veloce varie funzionalità relative alla Resource Directory.

La prima ottimizzazione riguarda proprio l'eliminazione di una parte di espressioni regolari e sfruttare le supposizioni definite in precedenza.
I risultati ottenuti in questo caso sono per un numero di POST su RD pari a 500 in relazione ai dispositivi considerati:

1. Raspberry Pi sia come mittente che come ricevente : **[69-70] secondi** con la presenza di qualche errore di timeout.
2. Raspberry Pi come mittente ed emulatore su PC come ricevente : **[7-8] secondi** senza alcun tipo di errore.
3. Emulatore su PC come mittente e ricevente : **[4-5] secondi** e non sono presenti errori di POST.

La differenza è di un buon 30% ma dobbiamo ancora applicare la nuova tipologia di parsing ad altre parti del codice.
Aggiungendola ai metodi che permettono aggiungere alcuni parametri della risorsa e che impostano il timer di scadenza per un numero di POST su RD pari a 500 con:

1. Raspberry Pi sia come mittente che come ricevente : **[63-65] secondi** non abbiamo alcun tipo di errore.
2. Dispositivi considerati: Raspberry Pi come mittente ed emulatore su PC come ricevente : **[6-7] secondi** senza alcun tipo di errore.
3. Emulatore su PC come mittente e ricevente : **[3-5] secondi** e non sono presenti errori di POST.

Dato che per ogni nodo che può effettuare la POST è probabile che ci siano più risorse da dover inviare, si può cercare di ottimizzare il riconoscimento dei vari attributi considerando la composizione nel CoRE Link Format. Una riscrittura completa di questa parte parte porta ai seguenti risultati per un numero di POST su RD pari a 500 con:

1. Raspberry Pi sia come mittente che come ricevente : **[43-67] secondi** senza alcun tipo di errore.
2. Raspberry Pi come mittente ed emulatore su PC come ricevente : **[5-6] secondi** senza alcun tipo di errore.
3. Emulatore su PC come mittente e ricevente : **[3-4] secondi** e non sono presenti errori di POST.

L'aggiunta dello *Splitter* di Guava in altre parti del codice per eliminare l'utilizzo dello *Scanner* con le espressioni regolari (insieme ad altre micro-ottimizzazioni, come ad esempio l'utilizzo di *StringBuilder* per la gestione ottimale della modifica alle String), ha portato a ridurre ancora il tempo necessario per il completamento delle operazioni. Per un numero di POST su RD pari a 500 in relazione ai dispositivi considerati:

1. Raspberry Pi sia come mittente che come ricevente : **[19-20] secondi** senza alcun tipo di errore.
2. Raspberry Pi come mittente ed emulatore su PC come ricevente : **[3-5] secondi** senza alcun tipo di errore.
3. Emulatore su PC come mittente e ricevente : **[0.5-1.4] secondi** senza errori di timeout.

Per un numero di POST su RD pari a 1000 con:

1. Raspberry Pi sia come mittente che come ricevente : **[50-53] secondi** senza alcun tipo di errore.
2. Raspberry Pi come mittente ed emulatore su PC come ricevente : **[11-12] secondi** senza errori.
3. Emulatore su PC come mittente e ricevente : **[1.6-2.1] secondi** senza alcun tipo di errore di timeout.

Per un numero di POST su RD pari a 5000 con:

1. Raspberry Pi sia come mittente che come ricevente : **[240-270] secondi**. In questo caso sono presenti parecchi errori di timeout e il risultato è quello di avere un dispositivo poco reattivo dato che viene sommerso da così tante richieste che per 60

secondi smette di essere responsive anche il server Californium.

2. Raspberry Pi come mittente ed emulatore su PC come ricevente : **[52-56] secondi** senza errori. Un tempo così alto è dato comunque dalla poca capacità del RaspberryPi di inviare un numero elevato di richieste in poco tempo.
3. Emulatore su PC come mittente e ricevente : **[3.1-3.6] secondi** senza errori di timeout.

Per un numero di POST su RD pari a 10000 con:

1. Raspberry Pi sia come mittente che come ricevente : **[482-500] secondi** e dopo circa 2 minuti vengono visualizzati i primi errori in timeout.
2. Raspberry Pi come mittente ed emulatore su PC come ricevente : **[103-106] secondi** senza alcun tipo di errore.
3. Emulatore su PC come mittente e ricevente : **[5.7-8.2] secondi** senza errori.

Naturalmente tutti gli errori in POST per i Raspberry Pi sono dovuti alla scarsa capacità di elaborazione per gestire molte richieste che arrivano in un tempo molto ridotto.
Analizzando i risultati restituiti dal comando 'top' si nota come il processo java occupi il 98% della capacità della CPU, rendendo il dispositivo poco reattivo.
Il guadagno in termini di prestazioni rispetto all'implementazione standard della Resource Directory è maggiore di 5 volte per quanto riguarda i dispositivi Raspberry Pi e di 6 volte per quanto riguarda l'utilizzo su emulatore Kura.

6.5. JDK 8

Per capire più a fondo perché abbiamo determinati risultati nei test condotti è necessario effettuare un'operazione di profiling del codice.

Nel momento in cui si acquista un Raspberry Pi la versione di Java installata di default è la numero 7. I test condotti in precedenza riguardavano l'esecuzione del framework Kura su questa versione.
Il primo problema riscontrato è nell'utilizzo del profiler interno di Java: VisualVM.
Per effettuare il collegamento in remoto e riuscire a visualizzare tutti i dati riguardo alla Virtual Machine è stato utilizzato *jstatd* in un primo momento, ma per un supporto completo a tutte le funzionalità è consigliato l'utilizzo di una connessione JMX.

Dato che Kura ha al proprio interno un firewall, non è possibile definire un intervallo di porte da poter aprire (sono presenti solo decisioni singole) e per avviare una connessione JMX su VisualVM viene utilizzato RMI, quindi bisogna trovare un workaround affinché si possa utilizzare il profiler.La soluzione adottata riguarda il tunnelling della connessione JMX su una specifica porta (è stata utilizzata la 9696).
Una volta effettuato il collegamento però, sono presenti diverse limitazioni delle funzionalità del profiler proprio perché viene utilizzata la JDK 7.
Ad esempio:

- Non è possibile effettuare un sampling della CPU per ottenere informazioni riguardo l'esecuzione effettiva di determinate classi.
- Non è possibile eseguire il campionamento della memoria per analizzare la presenza di eventuali leak.
- Sono assenti alcune funzionalità che riguardano il dump delle informazioni recuperate durante l'esecuzione.

In tutti i casi il problema viene risolto installando ed eseguendo il framework sulla versione 8 di Java.

Oltre a questioni di natura funzionale, analizzando i test condotti da http://java-performance.com/, su vari aspetti di Java, si può capire che nella versione 8 della JDK sono state introdotte moltissime ottimizzazioni anche nelle operazioni più semplici:

- È stata migliorata l'implementazione della classe *String* che ora, utilizzando meno

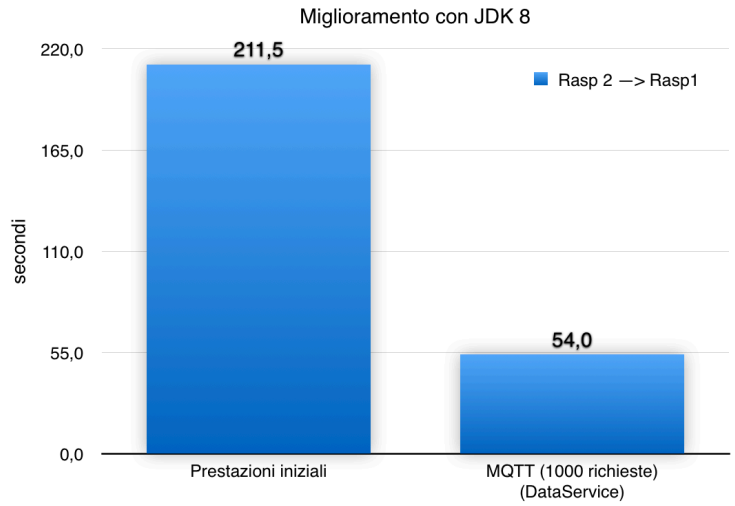
strutture dati di appoggio al proprio interno, permette l'esecuzione delle stesse operazioni in un tempo e con un consumo di memoria notevolmente ridotto.

- È presente una migliore implementazione delle strutture dati *Map* e *Set*.
- È stato introdotto un nuovo framework per la gestione delle date sulla falsa riga di un altro famosissimo framework utilizzato per le versioni precedenti: *Joda Time*.

Anche se al momento non c'è il supporto ufficiale a Java 8 da parte di Kura, sono stati effettuati comunque dei test delle funzionalità del supporto su questa versione.
L'idea è quella di avere un miglioramento generale delle prestazioni alla luce delle modifiche effettuate a basso livello nella JVM (soprattutto su architettura ARM).

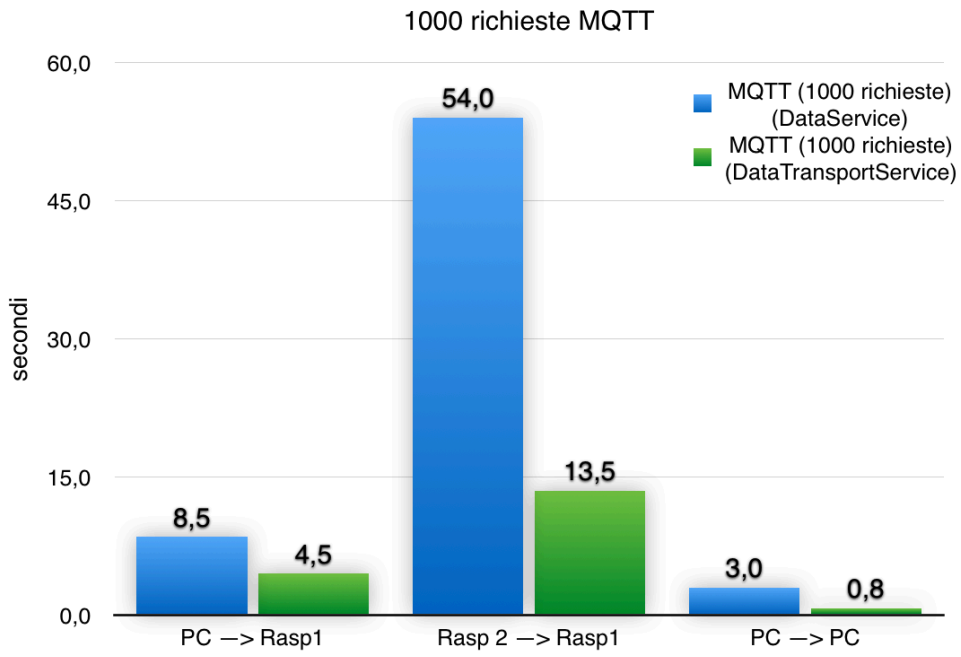
6.5.1. Risultati su MQTT

Senza effettuare nessuna modifica al codice, rieffettuando i test condotti in precedenza e controllando i risultati:

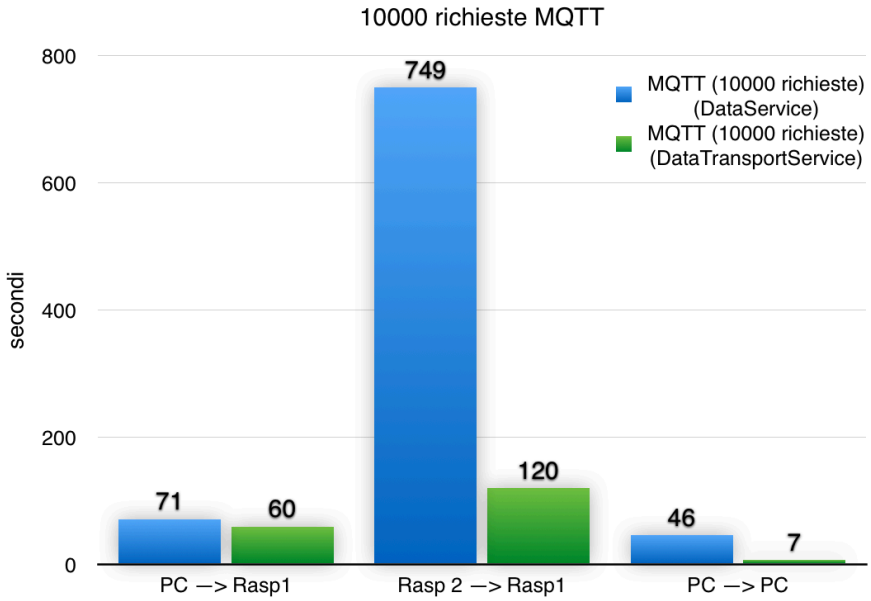


Possiamo notare che su Raspberry Pi, l'utilizzo del DataService, con l'aggiornamento a JDK 8, è migliorato di molto. La differenza è di circa 4 volte, a favore dell'ultima versione di Java. In genere il dispositivo ha una risposta migliore rispetto ai test iniziali e questo probabilmente è dovuto alla migliore implementazione di alcune classi che vengono utilizzate in modo massiccio all'interno del framework Kura e ad un miglioramento generale della JVM su architetture ARM. Infatti per quanto riguarda i test condotti su PC le prestazioni non sono migliorate in maniera così evidente.

Consideriamo a questo punto anche le altre interazioni. Con le nuove condizioni otteniamo per 1000 richieste:



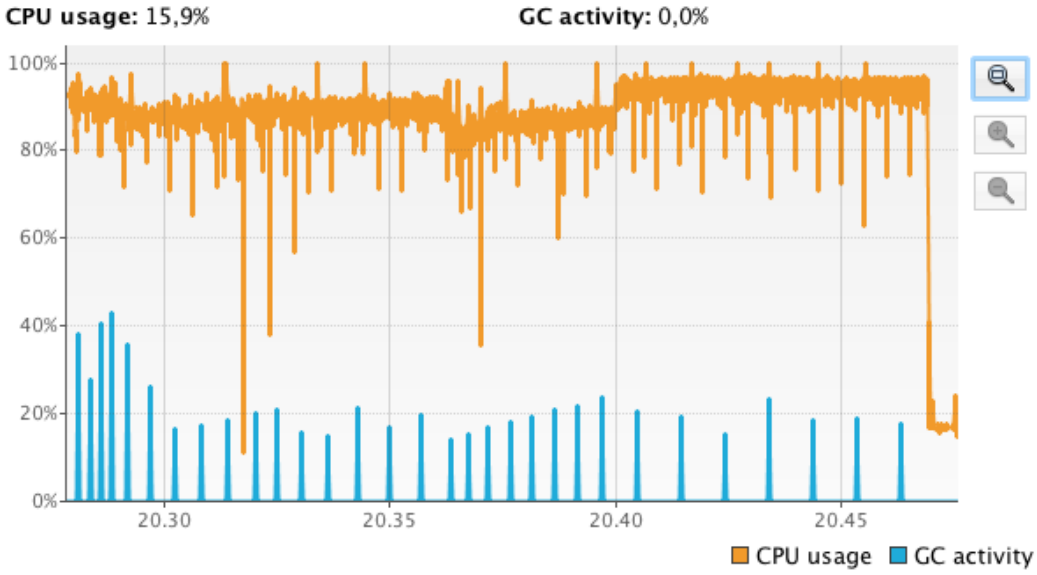
Anche se le prestazioni sono migliorate rispetto alla situazione iniziale, dal grafico risulta che comunque l'utilizzo del DataTransportService è comunque più performance rispetto al caso con DataService di circa 4 volte.
Per 10000 richieste:



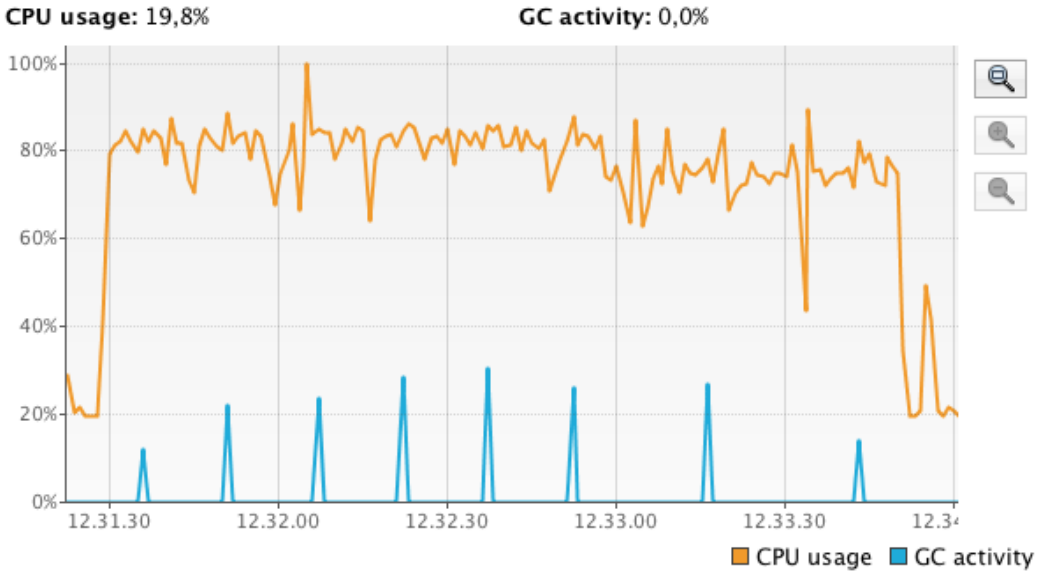
Con l'aumentare del numero di operazioni la differenza diventa ancora più evidente dato che si aggiunge un overhead enorme su chi invia la richiesta, che deve memorizzare il messaggio prima di inviarlo (in questo caso per evitare l'errore DataStoreExceeded è stato aumentato il numero di messaggi inflight che è possibile memorizzare a 100000, che ha come conseguenza il rallentamento del dispositivo).

Infatti questo comportamento lo possiamo notare dagli estratti di VisualVM:

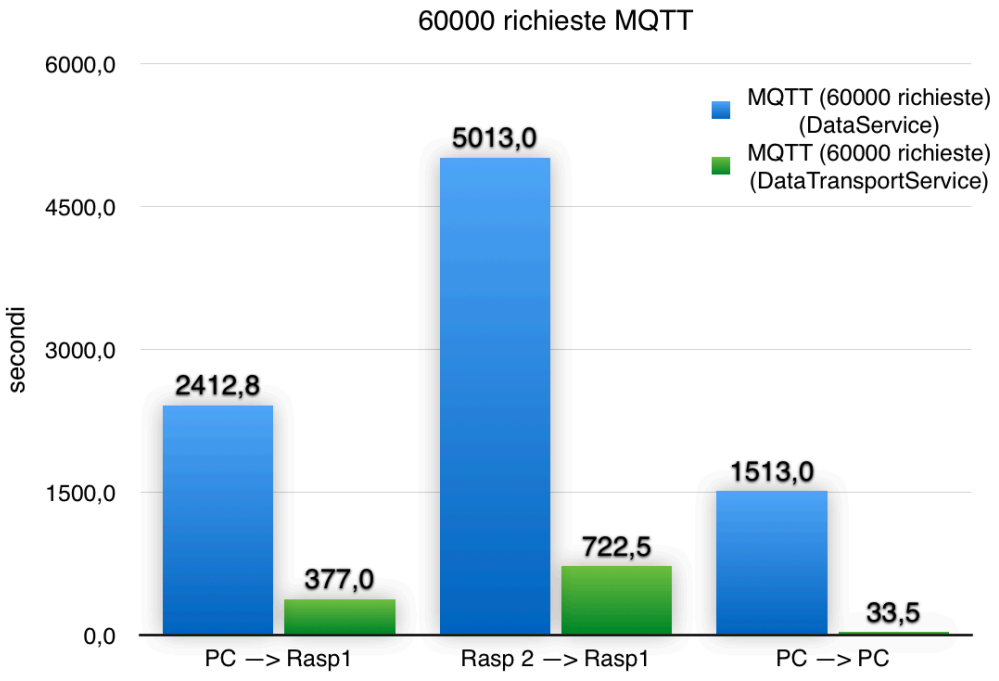
- Carico CPU per il caso *Rasp2 —> Rasp1* con *DataService*:



- Carico CPU per il caso *Rasp2 —> Rasp1* con *DataTransportService*

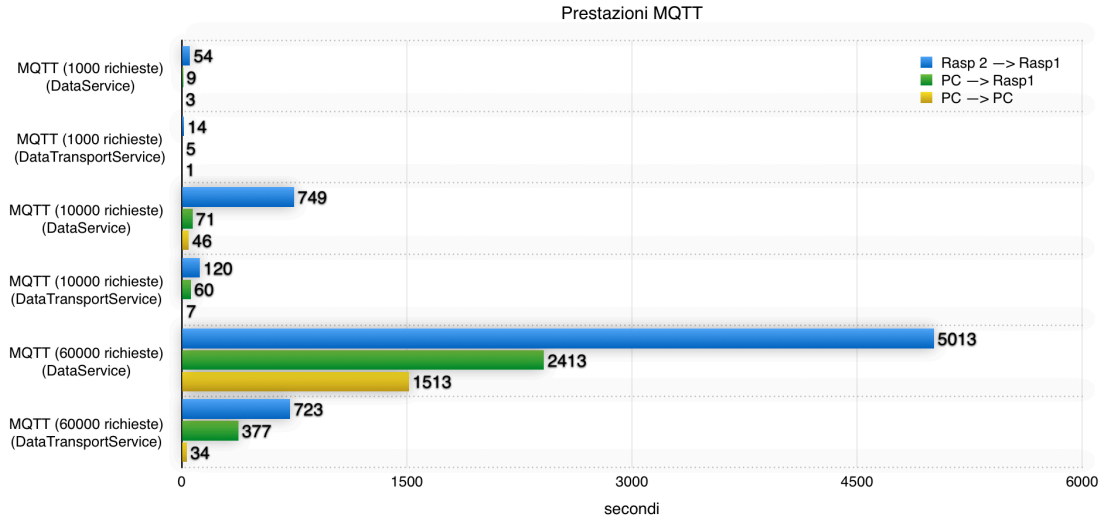


Questo è invece il test per 60000 richieste:



Bisogna fare una precisazione per il caso da PC verso Raspberry Pi: in alcuni test, prima del completamento, il DataService si disconnette dal broker MQTT.

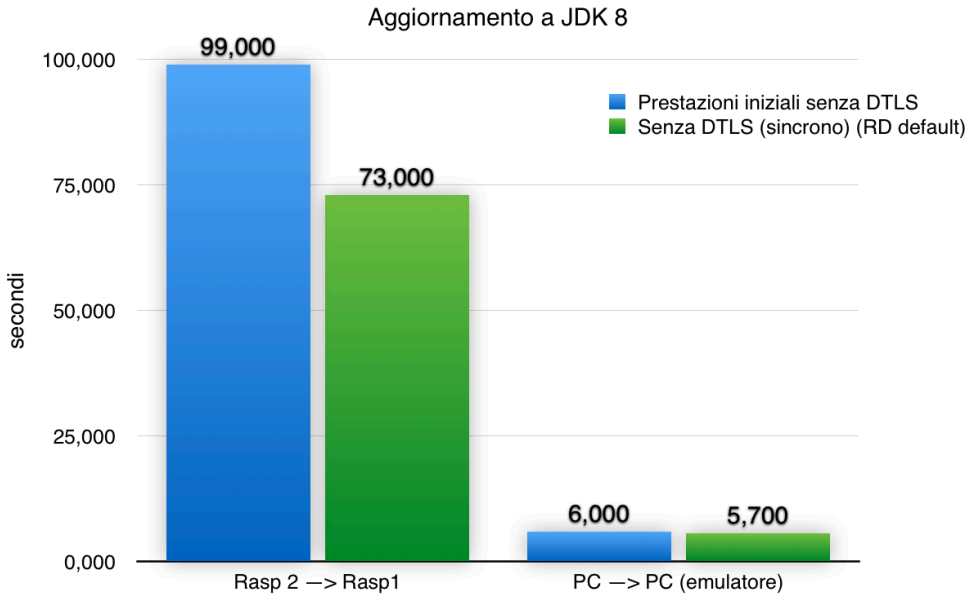
Molto probabilmente questo comportamento è dettato dallo stesso motivo che accompagnava i problemi iniziali: dato che il PC è molto più veloce nell'elaborazione delle richieste, il Raspberry viene sommerso di richieste e non riesce a gestire correttamente le operazioni di keepAlive verso il broker MQTT.
Il grafico del riepilogo delle prestazioni è il seguente:



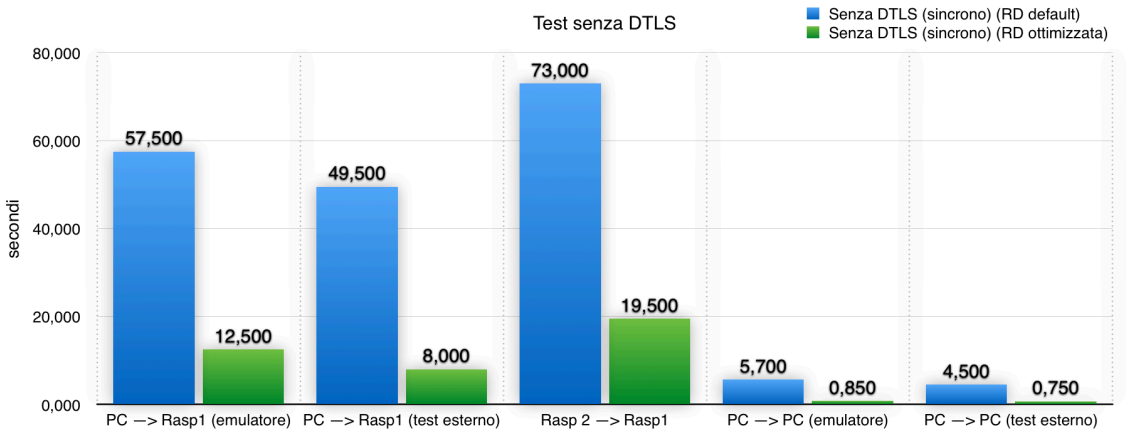
6.5.2. Risultati su RD

I test eseguiti in precedenza riguardavano l'interazione tra un dispositivo Kura ed un altro. Si teneva conto al massimo di un client e quindi di un numero di risorse limitate.
Per valutare in maniera più significativa il supporto sono state effettuate altre prove con nodi virtuali indipendenti che eseguono un POST con 10 risorse e che quindi vanno a creare un numero di entry all'interno della Resource Directory simile a quello del caso d'esempio dell'ufficio. Questo tipo di test verrà riferito nei grafici come "test esterno".

Partiamo prima dall'estrazione dei dati di interesse riguardo l'aggiornamento a JDK 8.
Le prestazioni migliorano leggermente nel caso dell'emulatore su PC e in maniera più consistente su Raspberry Pi, come risulta dai test effettuati:

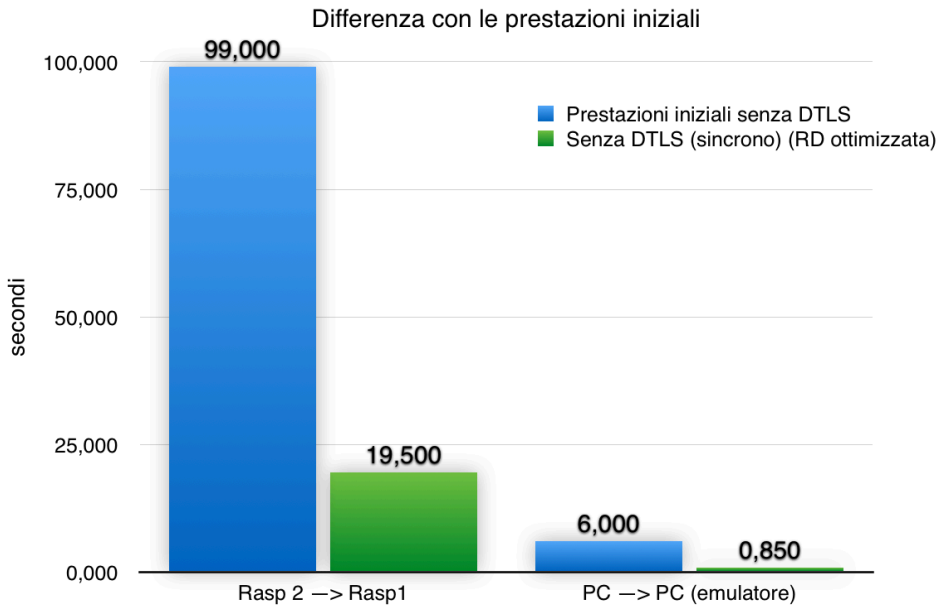


Questi invece sono i miglioramenti ottenuti rieseguendo i test su JDK 8 tra la versione default di Californium e quella che è stata realizzata per il supporto:



Come possiamo notare i risultati ottenuti rimangono comunque consistenti, anche se da JDK 7 a JDK 8 l'implementazione di default ottiene un boost prestazionale di un certo livello, in maniera simile succede anche alla versione custom realizzata per il supporto.

Di seguito invece è il confronto tra la versione iniziale su JDK 7 e quella finale su JDK 8:



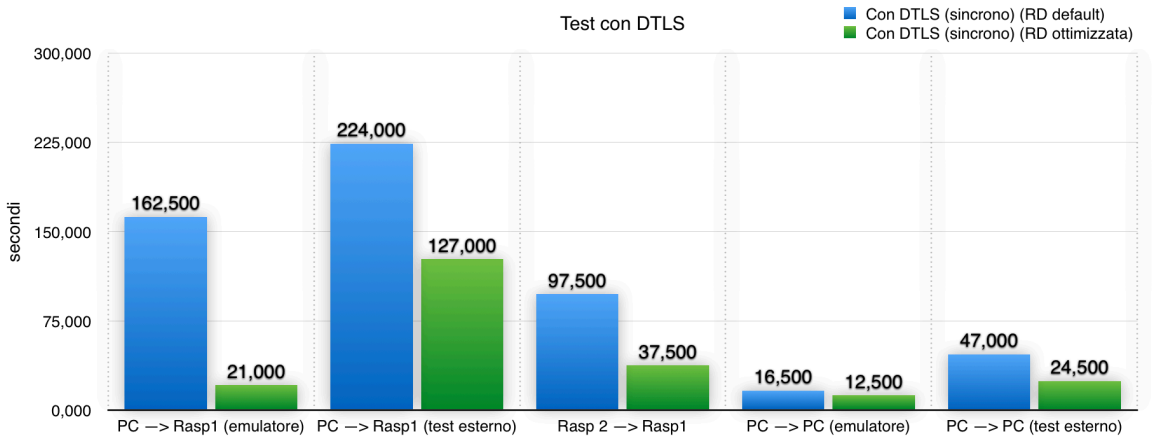
Il miglioramento è maggiore di 5 volte per il RaspPi mentre è maggiore di 7 volte su emulatore su PC.

6.5.3. Risultati con l'utilizzo di DTLS

Vediamo come l'aggiunta del supporto DTLS influisce sulle prestazioni del supporto.
In maniera simile ai test effettuati per quanto riguarda l'utilizzo del normale connettore UDP per CoAP, le prove effettuate su DTLS sono state effettuate in due modi:

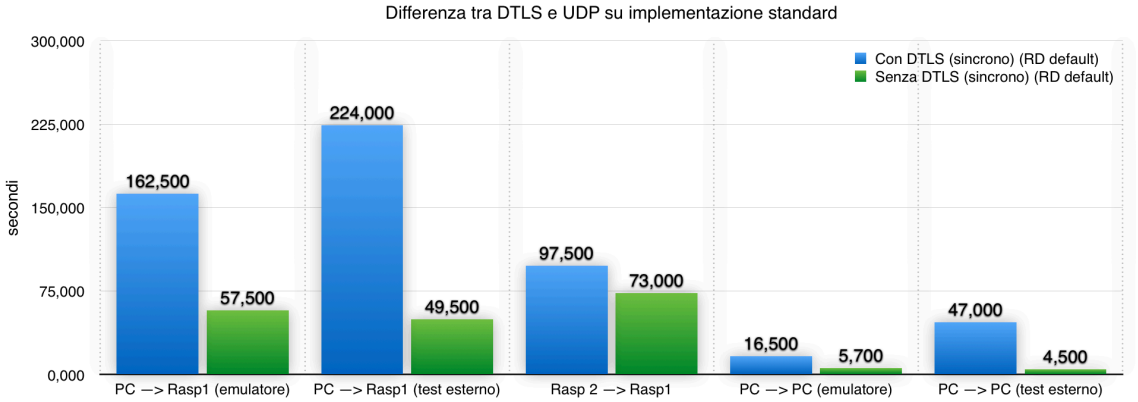
1. Utilizzando la normale interazione tra l'installazione nativa di Kura sui dispositivi oppure tra emulatori su PC. In questo caso viene eseguito l'handshake DTLS solo nella prima interazione, mentre nelle successive si ha lo scambio criptato dei dati.
2. Utilizzando un test esterno in cui vengono creati 500 client diversi e per ognuno, prima che venga inviato il messaggio, viene eseguito l'handshake con il destinatario.

I risultati ottenuti per quanto riguarda le implementazioni su DTLS sono i seguenti:



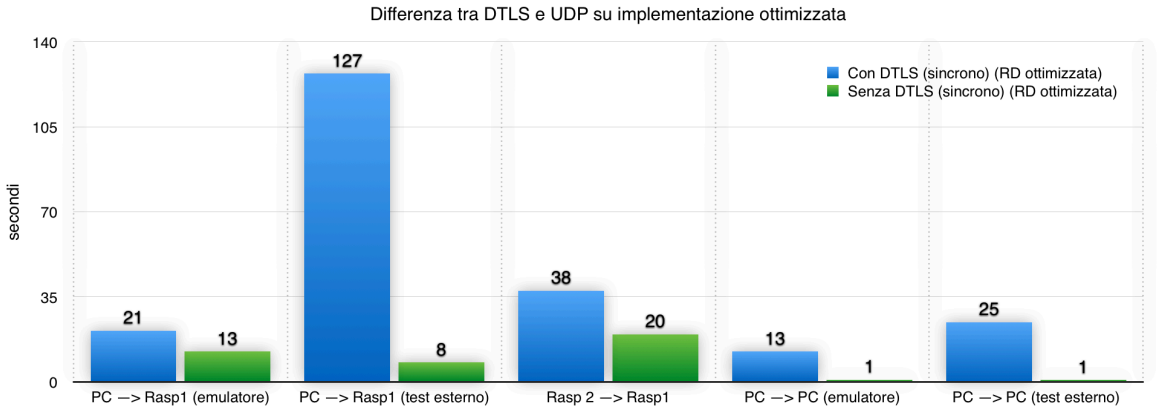
Come possiamo notare la differenza tra le due versioni rimane consistente, e nel caso di interazione tra due Raspberry Pi è di quasi 3 volte.

Questo invece è il grafico che mette in risalto le prestazioni su DTLS e su UDP dell'implementazione di default:



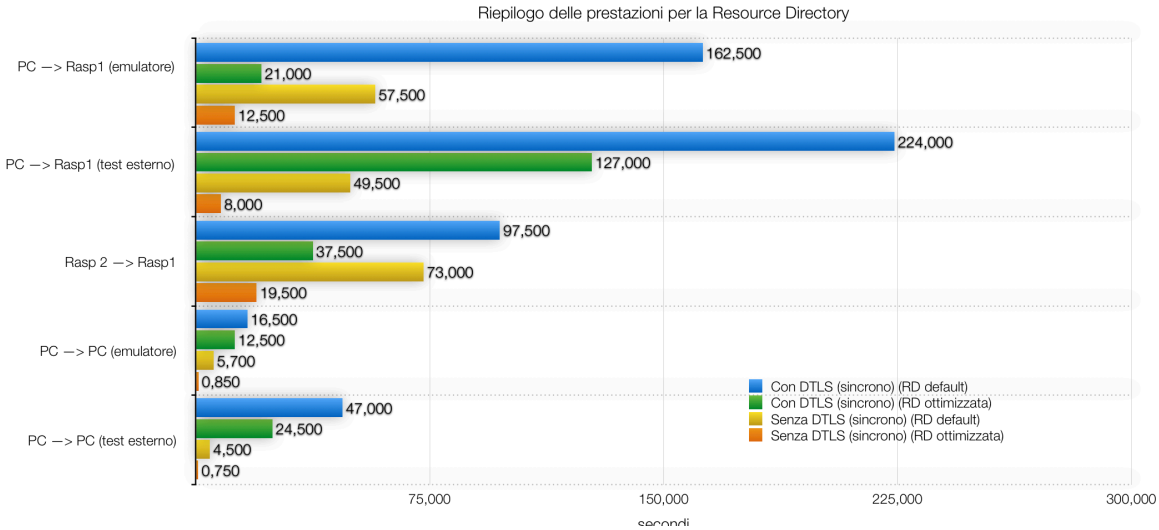
Quando viene effettuato un handshake per ogni richiesta la differenza può diventare enorme, come nel caso del test esterno su emulatore che è maggiore di 10 volte, mentre nel caso di interazione tra due Raspberry dato che nel test abbiamo un solo handshake (con i successivi pacchetti inviati in modo cifrato) è minore di 1,5 volte.

Questi sono i risultati della versione ottimizzata:



Anche in questo caso la differenza è alta ma in alcuni casi possiamo notare come l'implementazione DTLS che utilizza la Resource Directory ottimizzata è addirittura più veloce di quella standard basata su UDP (nel caso di interazione tra PC e Raspberry e tra Raspberry).
L'implementazione Scandium per Californium non ha come scopo quello di avere un supporto performante per dispositivi a bassa capacità computazionale, ma quello di ottenere un supporto completo per aggiungere a Californium la maggior parte delle funzionalità finora discusse per CoAP (al momento comunque c'è molto lavoro da fare soprattutto in chiave Connector).

Il grafico finale dei risultati ottenuti è il seguente:

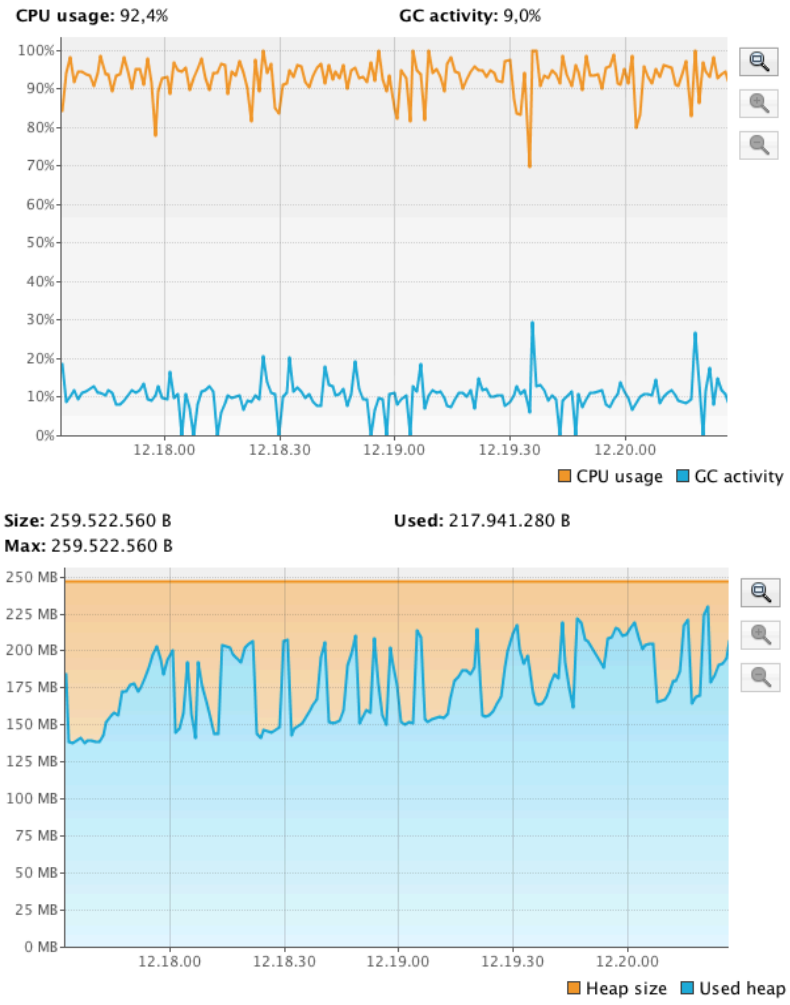


6.6. Ottimizzazione degli intervalli di validità delle risorse su RD

Analizzare soltanto il tempo necessario per il completamento delle operazioni non restituisce un quadro dettagliato della situazione in cui si troverà il dispositivo successivamente.

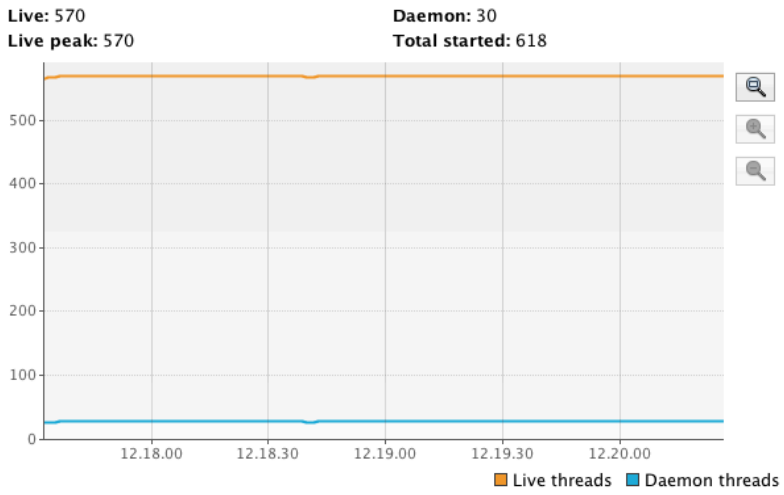
Per questo motivo è stato utilizzato il profiling tramite VisualVM: per verificare che non ci siano complicazioni nel caso si vogliano effettuare altre operazioni. Ed è tramite il profiler che è stato trovato un altro problema nell'implementazione standard di Californium.

Questa è la situazione della CPU e della RAM dopo aver eseguito vari POST, con circa 500 risorse da 4 attributi caricate:



Il consumo della CPU è praticamente superiore al 92% e il consumo di memoria è abbastanza

elevato su un dispositivo come il Raspberry Pi.
Cercando di effettuare altri test la situazione peggiora drasticamente: analizzando il quadro dei thread attualmente utilizzati:



Notiamo che sono presenti circa 570 thread attivi che rendono il dispositivo inutilizzabile.

Per capire perché in presenza di un quantitativo alto di risorse salvate nella RD abbiamo questo comportamento bisogna prima descrivere il contenuto dello standard e come è stato implementato nel progetto Californium: ogni volta che viene inserita una risorsa viene associato un thread per il controllo della scadenza della stessa.
Come definito nella RFC della Resource Directory ogni entry deve avere un intervallo di validità dopo il quale bisogna rimuoverla in modo tale da gestire anche quei nodi che si disconnettono prima di inviare un messaggio DELETE al broker.
Infatti analizzando il codice all'interno di un ResourceNode è presente una classe interna Runnable che esegue la rimozione della risorsa una volta eseguito il timer associato.

È una soluzione che si sposa benissimo con il contenuto della RFC ma, naturalmente, nel caso siano presenti un numero elevato di nodi otteniamo prestazioni inaccettabili dato che il numero di thread aumenta proporzionalmente al numero di broker che effettuano la POST.

Se si vuole rimanere conformi allo standard l'implementazione con thread associato ad ogni nodo è la più corretta possibile. È importante, nel nostro caso e in generale nelle situazioni con un elevato numero di nodi, cercare di rendere il dispositivo il più veloce possibile ma anche ridurre il consumo di memoria cercando di restare quanto più possibili vicini alle

direttive dello standard.

I requisiti che si vogliono soddisfare per il supporto sono:

1. Ridurre l'overhead dato dall'enorme numero di thread.
2. Consentire una certa flessibilità nella scelta della precisione dell'intervallo di validità delle risorse su un nodo.
3. Avere comunque un meccanismo di rimozione dei nodi che sono presenti per più di un certo periodo di tempo.
4. L'associazione di un intervallo temporale deve avvenire velocemente, altrimenti si rischia di rallentare tutta l'operazione di POST.

L'associazione di un intervallo temporale (con eventuale modifica) è una di quelle operazioni che se effettuate con il metodo standard di Java consuma un numero elevato di cicli di clock. Esiste un framework chiamato *Joda Time* che ormai è lo standard de facto per tutte le operazioni di creazione e di modifica di date. È molto più veloce ed efficiente della normale implementazione Java (della JDK 7 dato che nella versione 8 è stata praticamente integrata in modo nativo questa libreria).

Per quanto riguarda il primo punto possiamo implementare un meccanismo più lasco rispetto a quello con thread per ogni nodo: invece di considerare una delete associata ad ogni risorsa utilizziamo un thread generico per la Resource Directory che dopo ogni intervallo effettua una pulizia di quelle scadute.
In genere non è così importante essere precisi al secondo. La data di validità di ogni risorsa con questo nuovo tipo di approccio può essere quindi considerata nel worst case come:

Data attuale + intervallo di validità + MAX(intervallo di aggiornamento del thread)

Facciamo un esempio:

- Una risorsa è stata aggiunta alle 12:30:00 con un intervallo di scadenza di 30 secondi. La data di scadenza è quindi alle 12:30:30.
- Consideriamo l'intervallo di aggiornamento del thread di rimozione pari a 60 secondi con l'ultima esecuzione effettuata alle 12:30:29.
- La risorsa verrebbe rimossa alle 12:31:29 secondi anche se è scaduta ben 59 secondi prima.

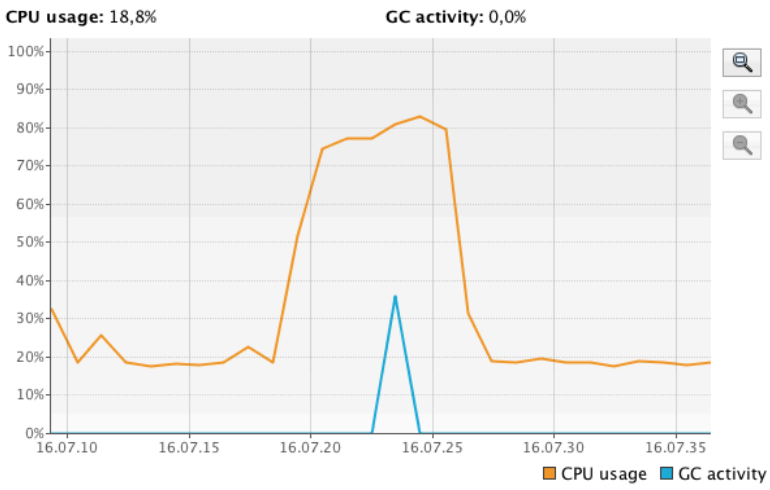
Naturalmente possiamo ridurre il tempo di controllo del thread, andando così a incrementare

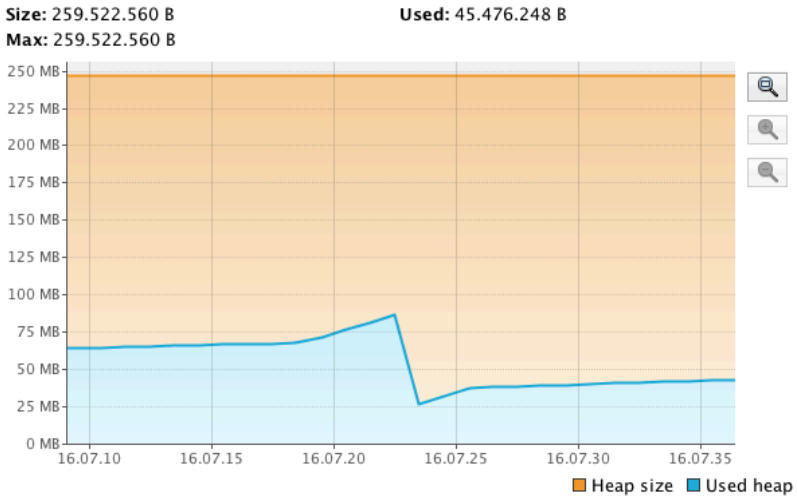
la reattività nella cancellazione delle risorse scadute.

Una considerazione legata al punto 4 è che se esiste questo thread di controllo è importante anche capire velocemente se la data di scadenza è situata cronologicamente prima della data di controllo. È anche per questo motivo che è stata scelta Joda Time, che risulta essere molto performante in questo tipo di operazioni.
Nel nostro caso non è così importante avere sempre le risorse aggiornate al millisecondo nella Resource Directory in quanto al massimo se non è più disponibile non restituirà un risultato e si perderà soltanto una chiamata CoAP che di per se è molto leggera dato che è stata progettata per essere utilizzata in ambito IoT.

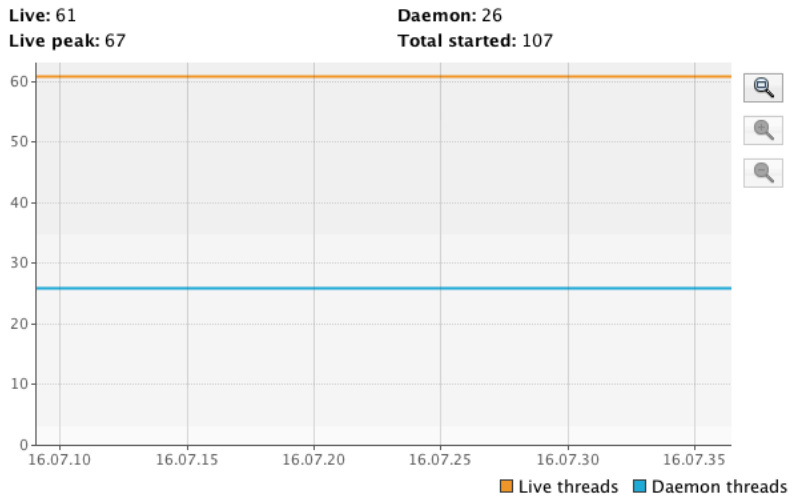
Un'altra scelta fatta è quella di non inserire il controllo della scadenza nel metodo di risposta REST dato che si andrebbero a rallentare le operazioni più semplici, che vengono effettuate un numero di volte molto alto.

Dopo aver implementato i metodi appropriati e rieseguendo il test su 500 risorse come nel caso precedente otteniamo questa situazione:





Il consumo di CPU è nettamente diminuito, così come quello di memoria che ora è più di 3 volte inferiore rispetto all'implementazione standard.
Ma soprattutto il numero di thread dopo l'esecuzione delle POST rimane praticamente identico:



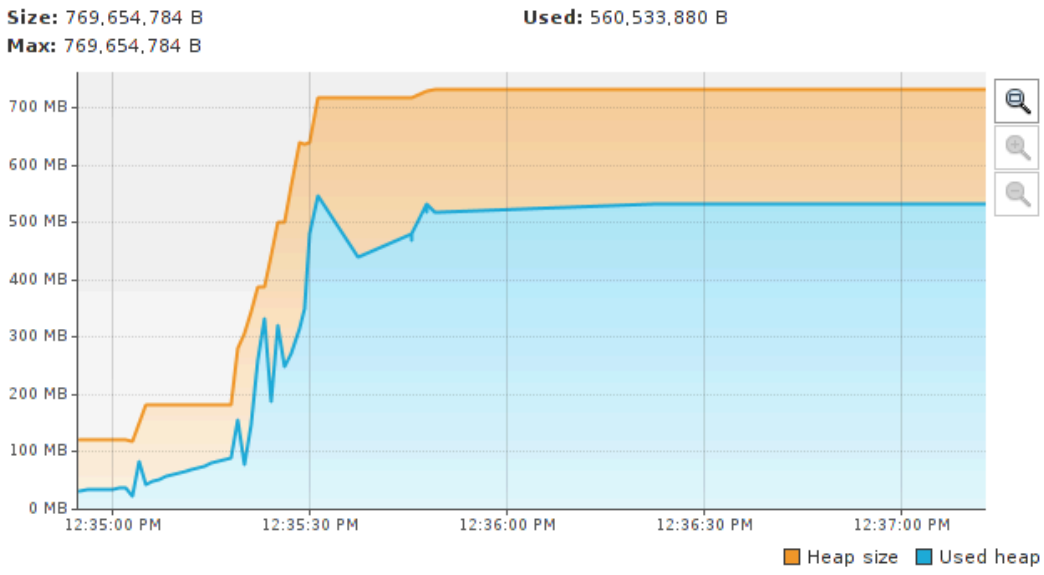
Questo tipo di soluzione ripagherà molto anche successivamente, quando si andranno ad analizzare le performance per le operazioni di lookup locale.

6.7. Test e ottimizzazioni delle operazioni di lookup su RD

Dopo che, in seguito ad una richiesta di tipo remote resource, sono terminate tutte le operazioni di POST sulla Resource Directory è possibile effettuare una lookup locale per la discovery dei nodi che hanno determinati tipi di attributi. Quindi è importante eseguire questa operazione in modo efficiente e veloce.
Sono state applicate le ottimizzazioni sul parsing utilizzate nelle operazioni di POST remoto descritte in precedenza anche nel caso della lookup.

All'interno del progetto *Californium Tools* è presente uno script che permette di effettuare uno stress test per quando riguarda richieste di tipo GET tramite CoAP.
Dato che, per richiamare una lookup locale, si deve eseguire un'operazione GET sul path "*rd-lookup/res*" con l'aggiunta degli attributi da ricercare, è possibile utilizzare coap-bench per calcolare il throughput sia dell'implementazione standard che di quella ottimizzata.

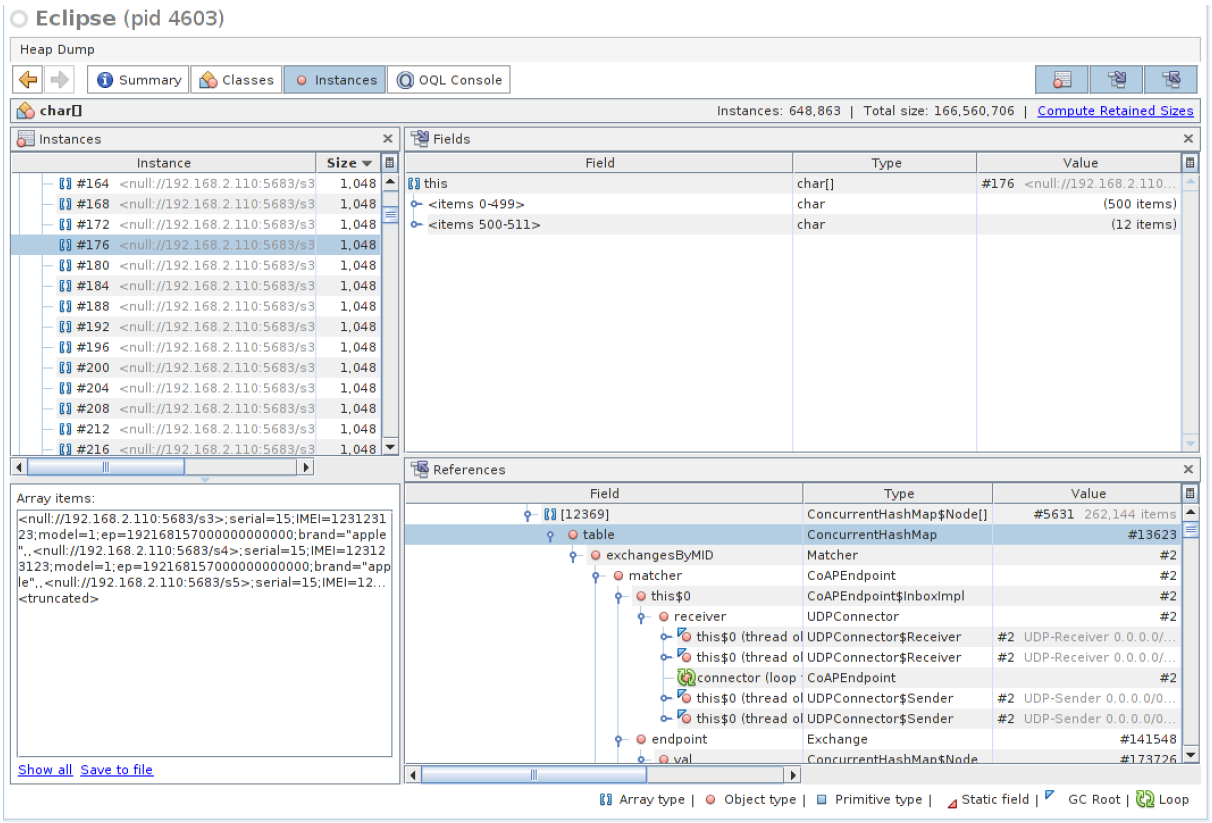
Durante l'esecuzione del test però è stato riscontrato un utilizzo di memoria al di sopra delle aspettative. Questo è lo screenshot di VisualVM durante l'esecuzione di coap-bench su PC:



Nel momento in cui l'utilizzo dell'heap raggiunge i 500 MB l'emulatore smette di essere responsive e dopo la seconda esecuzione del benchmark va in blocco.

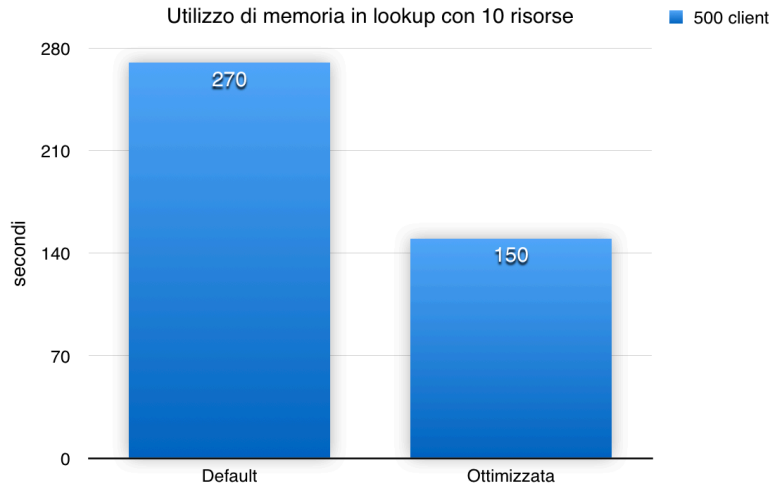
Dopo un'attenta analisi tramite gli strumenti di profiling forniti da VisualVM è stato

riscontrato un leak nel *Matcher* utilizzato da Californium per tenere conto degli scambi effettuati su CoAP:

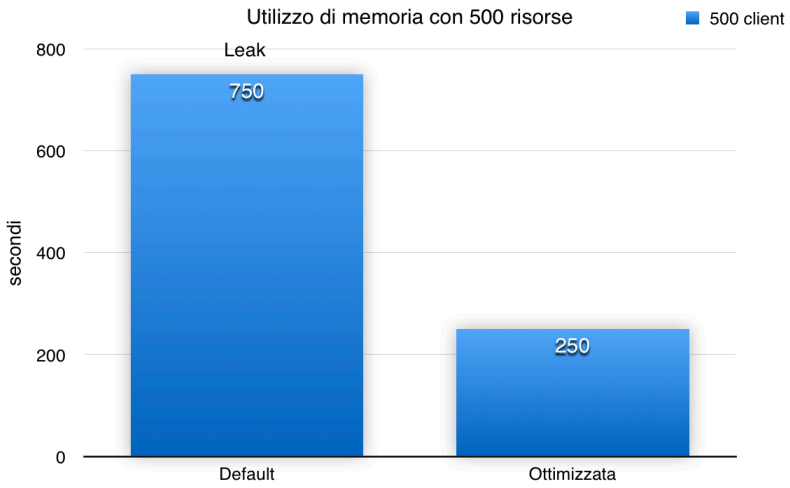


Dallo screen del dump della memoria heap è possibile notare che è presente un enorme spreco di memoria per la conservazione delle risposte dei messaggi che sono stati già scambiati. Anche richiamando il Garbage Collector tramite la funzione di VisualVM la situazione non cambia dato che sicuramente ci sono ancora dei riferimenti interni alle istanze che causano questo tipo di comportamento.

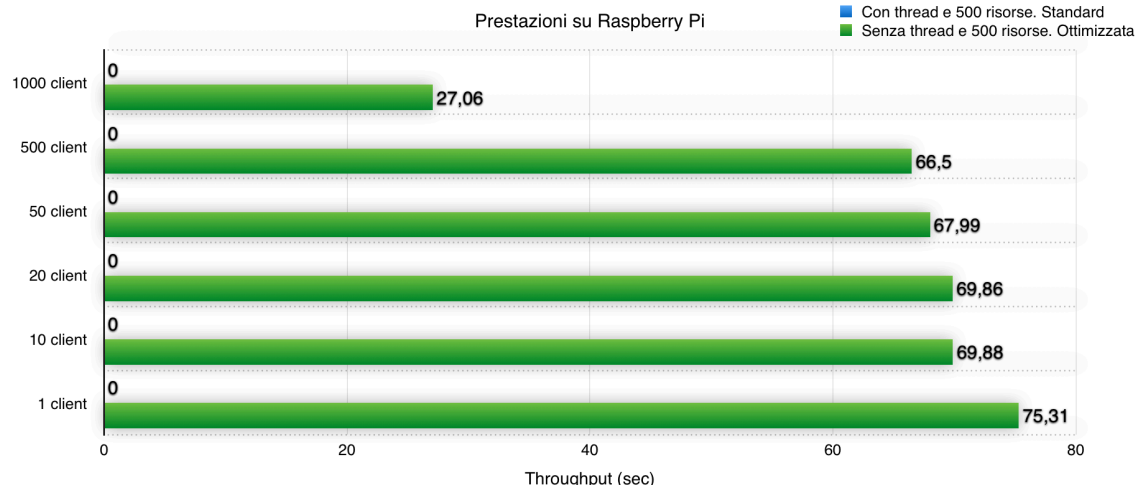
Per risolvere il problema del leak di memoria è stata aggiunta la chiamata per l'esecuzione dell'operazione di clear (cancellazione delle relazioni tra risposte e mittenti) nella classe interna di Californium Matcher dopo ogni lookup.
Sono stati quindi rieseguiti i test e per quanto riguarda l'utilizzo di memoria:



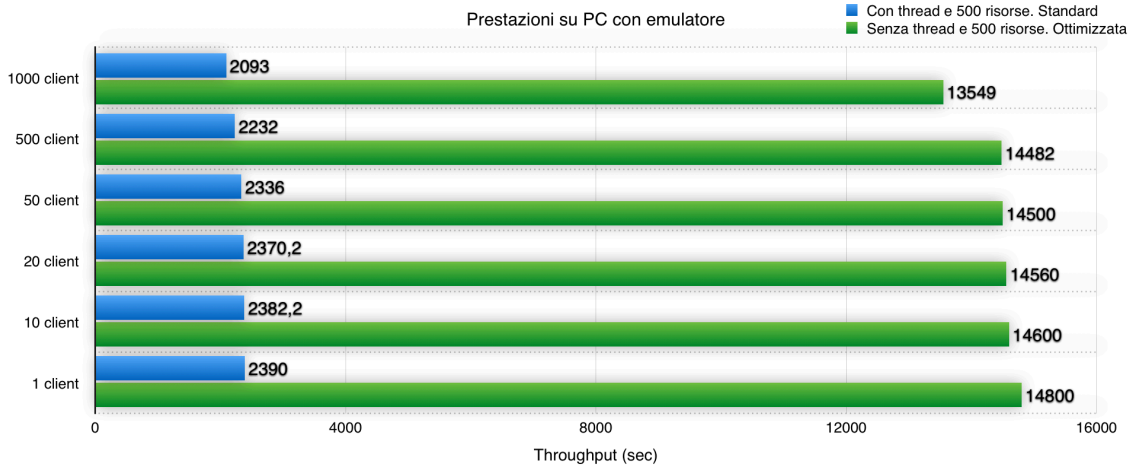
Già con 10 risorse il consumo inizia ad essere consistente nel caso dell'implementazione default.
Con 500 risorse:



Nel caso dell'implementazione standard, come descritto in precedenza, è presente un leak che compromette le funzionalità del dispositivo mentre nel caso di clearing tramite il Matcher si ottiene un consumo massimo di 250 MB.
Per quanto riguarda il throughput massimo raggiungibile su Raspberry Pi:



Il leak su Raspberry Pi compromette il normale funzionamento di Kura portando ad un blocco del framework. Coap-bench infatti restituisce un throughput pari a 0.
Con la versione ottimizzata si riesce ad ottenere un buon funzionamento fino a 500 client mentre con 1000 il throughput diminuisce di quasi 2,5 volte.



Comportamento simile per quanto riguarda l'utilizzo su emulatore installato su PC. Il leak non compromette del tutto il funzionamento del framework ma comunque ne limita di molto le performance. La differenza di prestazioni tra la versione standard e quella ottimizzata è maggiore di 6 volte.

6.8. Considerazioni finali e sviluppi futuri

Una prima nota importante relativa alle prestazioni riguarda la configurazione del Raspberry Pi per l'esecuzione di Kura. Dai test effettuati sulle versioni 7 e 8 della JDK è emerso che

nell'ultima sono state introdotte diverse ottimizzazioni che hanno minimizzato alcuni problemi riscontrati su MQTT e hanno migliorato le prestazioni in generale.
Infatti nel caso di MQTT si ha un aumento delle performance di circa 4 volte rispetto al caso con JDK7 mentre per CoAP si ha un miglioramento di circa il 30%.

Per quanto riguarda i risultati ottenuti attraverso le ottimizzazioni è ora possibile effettuare un maggior numero di richieste MQTT, sia per le comunicazioni verso il CoAPTreeHandler che verso altri broker, mantenendo un carico moderato sulla CPU e sulla memoria RAM.
In particolare, rispetto alla situazione iniziale, la scelta di utilizzare framework esterni come Kryo (per la serializzazione) e i vari test effettuati sui service interni DataService e DataTransportService, hanno dimostrato che l'implementazione finale del supporto ha un miglioramento delle prestazioni di quasi 15 volte su Raspberry Pi e maggiori di 6 volte per comunicazioni tra emulatori su PC con una riduzione massiccia degli errori di comunicazione. Questo si traduce in una maggiore affidabilità per quanto riguarda le trasmissioni tra i broker e probabilità molto ridotta di ottenere errori in fase di invio al server MQTT. Dato che una resource query a volte può richiedere, prima del completamento, di essere propagata su diversi livelli della gerarchia, necessita di dover essere elaborata in un tempo molto breve. Meno errori sono presenti durante tutto il processo di richiesta e più coerente è il contenuto della Resource Directory con la situazione reale di tutte le risorse alla fine delle operazioni.

In ambito CoAP invece sono stati effettuati diversi miglioramenti attraverso l'utilizzo delle librerie Guava di Google con le quali è stato sviluppato un modo efficiente e veloce di effettuare il parsing delle stringhe di richieste e del payload.
Il guadagno in termini di prestazioni rispetto all'implementazione standard della Resource Directory è risultato maggiore di 5 volte per quanto riguarda i dispositivi Raspberry Pi e di 6 volte per quanto riguarda l'utilizzo su emulatore.
Successivamente è stato poi migliorato il comportamento sulle operazioni di riconoscimento delle risorse scadute e la loro rimozione. L'implementazione standard rendeva impossibile il corretto funzionamento del dispositivo dopo l'aggiunta di un numero elevato di risorse. Riducendo il numero di thread e rendendo meno reattivo il rilevamento degli intervalli di scadenza, si è riusciti ad ottenere una considerevole riduzione del consumo di memoria, di almeno 3 volte inferiore rispetto al caso standard. Inoltre, il consumo della CPU è stato diminuito del 20%.
Questo si traduce in un dispositivo molto più reattivo nel caso in cui venga sommerso di

richieste e in un maggiore throughput per il completamento delle operazioni CoAP.
Con riferimento all'esempio degli uffici privati ora è possibile effettuare le 500 richieste contemporaneamente da tutti i dispositivi in un tempo considerevolmente inferiore e senza problemi di timeout.

In aggiunta ai miglioramenti precedenti, è stato preso in considerazione anche il caso di lookup locale utilizzata dopo che le risorse remote sono state correttamente inserite nella Resource Directory tramite operazioni di POST.
In questo caso il miglioramento è stato consistente dato che si è passati da avere un throughput di 0 (il dispositivo non riusciva ad effettuare operazioni per il numero di thread molto elevato) ad avere quasi 70 operazioni al secondo con un numero di client fino a 500. Aumentando il numero di client a 1000 si ottiene un throughput di circa 23 operazioni al secondo. Questo quando si utilizza il Raspberry Pi.
Riguardo al PC, invece, data la maggiore disponibilità di risorse computazionali, si è passati da avere circa 2000 operazioni al secondo a più di 14000.
Ora è quindi possibile gestire un numero molto elevato di risorse temporanee all'interno della Resource Directory. Secondo le specifiche standard presenti nel *draft-ietf-core-resource-directory* e in accordo all'implementazione standard, dato che il numero di thread eseguiti nella JVM era proporzionale alle risorse presenti all'interno della RD, anche con un numero esiguo di nodi era impossibile ottenere prestazioni accettabili.
Con riferimento al caso d'esempio, una volta che sono terminate tutte le operazioni di POST, è possibile ricevere le informazioni sulle 400 risorse disponibili in modo molto veloce dato che il throughput del supporto, anche nel caso di Raspberry Pi, ha un valore adeguato ai requisiti richiesti, soprattutto nel caso di molti client.

Con l'utilizzo di DTLS la situazione cambia e anche con le ottimizzazioni applicate le operazioni di handshake pesano parecchio su ogni richiesta.
Si ha un peggioramento di circa 15 volte tra la versione con DTLS e quella che utilizza il connector UDP. Addirittura, il comportamento su PC peggiora di 25 volte anche se resta di circa 5 volte più veloce rispetto la controparte Raspberry Pi.
Quindi sarebbe utile avere a disposizione una libreria DTLS che abbia un consumo di risorse molto ridotto. Scandium è un'estensione di Californium che si prefigge di essere flessibile da utilizzare e quanto più fedele possibile al protocollo. Essa però è sviluppata in Java e non è ottimizzata per l'uso su dispositivi a bassa capacità computazionale.

Un altro miglioramento, che è possibile introdurre anche in Scandium, è che si effettui automaticamente il riconoscimento delle sessioni scadute in modo tale da non essere costretti ad effettuarne la chiusura ogni volta che termina un'interazione.
Come definito nella parte di progettazione è molto utile non dover rieffettuare l'handshake per ogni comunicazione in modo tale da consumare il minor numero di risorse possibili, soprattutto se si utilizzano soltanto Raspberry Pi.

Un'altra possibile ottimizzazione è quella che riguarda la riduzione del numero di messaggi generati quando si deve effettuare una richiesta che coinvolge un numero alto di nodi, magari anche in domini diversi.
CoAP ed MQTT sono due protocolli nati per l'IoT ma hanno scopi ben differenti.
MQTT è un protocollo che implementa il modello publish-subscribe e permette di ottenere un livello di qualità di servizio migliore rispetto a CoAP ed utilizza TCP.
CoAP invece è nato per ottenere operazioni REST efficienti e veloci utilizzando UDP.
Quindi sarebbe inutile considerare CoAP per gli scambi delle informazioni sui cambiamenti della gerarchia dell'albero dato che in alcuni casi abbiamo bisogno di un livello di QoS di tipo *assured delivery*. Si potrebbe invece utilizzare nel caso in cui si vogliano effettuare richieste di risorse, dato che non abbiamo requisiti così stringenti.
In particolare, si potrebbe utilizzare la modalità multicast così un nodo, al posto di interrogare il proprio parent nella gerarchia, invia il messaggio di richiesta all'indirizzo multicast e soltanto i broker interessati (che hanno almeno una risorsa con gli attributi richiesti) effettueranno le operazioni di POST sulla RD del richiedente.
In questo modo, si riduce anche il numero di messaggi che circolano sulla rete ottenendo così un'implementazione ancora più efficiente in quei casi in cui è effettivamente possibile utilizzare il multicast.
Si potrebbe sfruttare questa modalità solo per le resource query, lasciando così la parte di tree handling al protocollo MQTT.
Purtroppo al momento non è disponibile un'implementazione del multicast su CoAP (nel progetto Californium), ma quando sarà finalizzata, potrà essere integrata nel supporto senza grossi problemi dato che basterà modificare solo il metodo di gestione delle risorse.

7. Conclusioni

L'obiettivo principale del progetto è stato portato a termine insieme a diverse ottimizzazioni alle librerie esistenti, utilizzate per supportare il protocollo CoAP.
Sono state effettuate varie scelte durante lo sviluppo che hanno portato ad un aumento di performance per il protocollo MQTT su Kura e alla riduzione del carico di lavoro, disabilitando l'utilizzo di operazioni non utilizzate dal supporto.
In conclusione è stato realizzato un supporto CoAP per Kura con le seguenti caratteristiche:

1. Estensibilità, dato l'utilizzo di OSGi e dei servizi implementati.
2. Ottima scalabilità, in maniera gerarchica dei nodi che fanno parte dell'albero.
3. Flessibilità, nell'utilizzo con la scelta dinamica del Connector UDP o DTLS.
4. Semplicità di configurazione tramite l'interfaccia web di Kura.
5. Utilizzo di tecnologie CoAP e MQTT progettate per l'IoT.
6. Sicurezza, tramite l'utilizzo di DTLS.
7. A basso costo, grazie alla possibilità di utilizzo su Raspberry Pi.

Sono stati realizzati esattamente 5 bundle OSGi, ognuno con uno scopo ben preciso.
Il service base viene utilizzato per fornire le librerie a bundle esterni oppure per ottenere informazioni sugli indirizzi di rete correntemente utilizzati dal dispositivo su cui è installato Kura.
Il service Californium permette di avviare un normale server CoAP ed è stato opportunamente sviluppato per integrarsi con il bundle del Broker in modo tale da poter attivare o disattivare il supporto DTLS oppure aggiungere e rimuovere dinamicamente le risorse.
Il service CoAPTreeHandler consente di gestire in maniera dinamica l'aggiunta e la rimozione dei nodi nella gerarchia con conseguente notifica delle operazioni eseguite.
Il bundle del broker CoAP è invece il centro di gestione delle risorse e permette di eseguire e ricevere richieste per lo scambio delle risorse remote. Utilizza simultaneamente MQTT e CoAP per la comunicazione.
L'ultimo service è quello relativo alle risorse remote. Si chiama RemoteResource e permette ai client di poter effettuare richieste sulla gerarchia di nodi, per risorse con attributi specifici.

In definitiva è possibile creare una struttura ad albero composta da nodi che comunicano tra di loro per scambiarsi le informazioni riguardo la posizione delle risorse, quindi su come possono essere raggiunte ed interrogate. Tutto questo mantenendo al minimo il numero di

scambi da effettuare e con una gestione automatica delle disconnessioni.

Grazie all'utilizzo di Kura si ha la garanzia di trovarsi di fronte ad una piattaforma che ha davanti un futuro luminoso, ricco di successi e che sicuramente avrà un ruolo di primo piano nella "IoT war" tra le varie aziende.
Sviluppare per questa nuova categoria di prodotti dovrebbe essere un piacere e avere un livello di astrazione abbastanza elevato sicuramente attirerà un numero elevato di sviluppatori che potranno contare su una piattaforma solida e flessibile come Kura.
Dal punto di vista della privacy, invece, sarà fondamentale in futuro avere a disposizione librerie sicure ed estremamente leggere da poter utilizzare nei progetti IoT basati su CoAP per evitare che tutti i dati generati vadano a finire nelle mani sbagliate.

8. Riferimenti

1) CoRE Resource Directory - draft-ietf-core-resource-directory-00
2) The Constrained Application Protocol (CoAP) - RFC 7252
3) Constrained RESTful Environments (CoRE) Link Format - RFC 6690
4) Web Linking - RFC 5988
5) MQTT For Sensor Networks (MQTT-SN) - Protocol Specification - Version 1.2
6) MQTT V3.1 e V3.1.1 Protocol Specification
7) Californium: Scalable Cloud Services for the Internet of Things through CoAP - Kovatsch, Lanter, Shelby
8) Hands-on with CoAP - Kovatsch, Vermillard
9) Sleeping and Multicast Considerations for CoAP - draft-rahman-core-sleeping-00.txt
10) Observing Resources in CoAP - draft-ietf-core-observe-14
11) Blockwise transfers in CoAP - draft-ietf-core-block-15
12) A Low-Power CoAP for Contiki - Kovatsch, Duquennoy, Dunkels
13) Performance Evaluation of MQTT and CoAP via a Common Middleware - Thangavel, Ma, Valera, Tan.
14) Osgi Alliance Core 6.0.0
15) Scalability for IoT Cloud Services - Lanter

Ringraziamenti

Ringrazio il Professore Paolo Bellavista per la pazienza e la cortesia con cui mi ha seguito durante tutto il periodo di tesi.

Ringrazio la mia famiglia che mi ha supportato in questi anni di studio e mi ha permesso di raggiungere questo traguardo.

Le difficoltà che si incontrano in una facoltà come Ingegneria sono indescrivibili.
Più che un corso di studi è un corso di vita: si impara a trattare con le persone, a gestire concetti complessi e ad affrontare le varie situazioni nel modo migliore.

Sono proprio contento di aver fatto questa scelta.

Gianvito