Politecnico di Milano

V Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria delle Telecomunicazioni

Dipartimento di Elettronica e Informazione

Progetto e sviluppo di un proxy avanzato per CoAP

Relatore: Prof. Matteo Cesana

Tesi di Laurea di:
Vincenzo Palmieri Matr. 740206

Anno Accademico 2013-2014

Indice

Elenco delle figure

Elenco delle tabelle

Elenco degli acronimi

6LoWPAN	IPv6 over Low power Wireless Personal Area Networks
ACL	Access Control List
ADC	Analog to Digital Converter
AES-CCM	Advanced Encryption Standard Counter with CBC-MAC
AODV	Ad hoc On-Demand Distance Vector
BLIP	Berkeley Low Power IP stack
BPSK	Binary Phase-Shift Keying
CAP	Contention Access Period
CFP	Contention Free Period
CoAP	Constrained Application Protocol
Cooja	COntiki Os JAva
Cooja	COntiki Os JAva
CSMA/CA	Carrier Sense Multiple Access/Collision Avoidance
CSS	Client Stateless Server
DAS	Destination Advertisement Object
DGRM	Directed Graph Radio Medium

DIO	DODAG Information Object
DIS	DODAG Information Solicitation
DNS	Domain Name System
DODAG	Destination Oriented Directed Acyclic Graph
DSSS	Direct Sequence Spread Spectrum
DTLS	Datagram Transport Layer Security
ETAG	Entity TAG
FFD	Full Function Device
GPS	Global Positioning System
GTS	Guaranteed TimeSlot
HAA	Hardware Abstraction Architecture
HAL	Hardware Abstraction Layer
HIL	Hardware Interface Layer
HPL	Hardware Presentation Level
HSPA	High-Speed Packet Access
HTTP	HyperText Transfer Protocol
IANA	Internet Assigned Numbesr Authority
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Taskforce
IP	Internet Protocol
ISM	Industrial, Scientific and Medical

LTE	Long Term Evolution
MAC	Media Access Control
MCU	MicroController Unit
MRM	Multi-path Ray-tracer Medium
MTU	Maximum Transmission Unit
ND	Neighbour Discovery
O-QPSK	Offset Quadrature Phase Shift Keying
PAN	Personal Area Network
PAN-C	Personal Area Network Coordinator
PAN-ID	Personal Area Network IDentifier
PHY	PHYsical layer
RAM	Random Access Memory
REST	Representational State Transfer
RF	Radio Frequency
RFD	Reduced Function Device
RISC	Reduced Instruction Set Computer
ROLL	Routing Over Low-power and Lossy networks
ROM	Read Only Memory
RPL	Routing Protocol for Low power and Lossy networks
RPL	Routing Protocol for Low power and Lossy networks
TLS	Transport Layer Security
UDGM	Unit Disk Graph Medium

UDP	User Datagram Protocol
URI	Uniform Resource Identifier
WSN	Wireless Senor Network

Sommario

Le reti di sensori sono una particolare tipologia di reti *wireless* caratterizzate da stringenti requisiti in termini di consumo energetico, da capacità di auto-organizzazione e da un'elevata resistenza ai guasti. Esse differiscono notevolmente dalle normali reti utilizzate per le comunicazioni internet; ciononostante, con il continuo sviluppo dell'*Internet delle Cose*, si rende necessario facilitare l'interoperabilità fra queste due reti così diverse. A tale scopo sono stati sviluppati negli ultimi anni una serie di protocolli che favoriscono detta comunicazione. Essi, naturalmente, riguardano tutti i livelli dello *stack* protocollare di rete. A livello applicativo si sta affermando il protocollo CoAP che, presentando analogie con HTTP, facilita l'accesso dal web alle risorse offerte dai nodi sensore e viceversa. CoAP prevede l'utilizzo di nodi che svolgano la funzione di *proxy*; essi però presentano alcune limitazioni. Scopo di questa tesi è lo sviluppo di un *proxy* avanzato dotato di funzionalità aggiuntive e che consenta di attuare un risparmio in termini di consumo energetico e tempo di servizio delle richieste.

Introduzione

Fra i *trend* tecnologici che nell'ultimo decennio stanno subendo un sostanziale mutamento vi è la disponibilità - accompagnata da una sempre crescente aspettativa - della connettività in ogni luogo. Sia che si tratti di controllare la propria casella email, piuttosto che navigare sul web, oggi ci si aspetta di poter accedere a questi servizi online ovunque e ad ogni orario. Ciò è reso possibile grazie a tecnologie *wireless* quali HSPA, WiFi, LTE, etc. ed è già divenuta una realtà consolidata per dispositivi personali quali PC, notebook, smartphone e tablet.

In un futuro non lontano si sentirà l'esigenza non solo di avere una connessione onnipresente, ma di avere connessa in rete qualsiasi *cosa*. E' la cosiddetta *Internet of Things*, grazie alla quale ogni oggetto si rende intelligente, diviene riconoscibile e sarà in grado di comunicare non solo con altri dispositivi della medesima categoria all'interno di una rete isolata, ma con qualsiasi tipologia di dispositivo, ovunque esso si trovi, per mezzo della rete internet.

Fra i fattori abilitanti vi è l'avanzamento tecnologico dell'elettronica digitale, che ha permesso la realizzazione di nodi sempre più piccoli, economici ed a basso consumo energetico, unitamente alla progettazione e sviluppo di architetture di rete e protocolli per le cosiddette *Wireless Sensor Network* (WSN). Le WSN sono una particolare tipologia di reti senza fili in cui i nodi sensori sono caratterizzati da dimensioni contenute, bassi consumi energetici e limitate capacità di elaborazione, i quali comunicano gli uni con gli altri per mezzo di tecnologie *wireless* a corto raggio. Le WSN sono reti collaborative concepite per auto-organizzarsi grazie a meccanismi intelligenti in grado di reagire in modo autonomo ad eventi critici e cambiamenti di topologia. Viste le caratteristiche dei nodi sensori, le reti WSN sono quindi caratterizzate da requisiti molto stringenti in termini di consumo energe-

tico; necessitano pertanto di protocolli specifici per il corretto funzionamento. Negli ultimi anni varie organizzazioni operanti in ambito ICT hanno sviluppato protocolli di rete ottimizzati per questo tipo di reti. In particolare, a livello applicativo, la IETF ha sviluppato il protocollo CoAP (Constrained Application Protocol) il quale, grazie alle sue analogie con HTTP, si propone come standard per fornire interoperabilità a livello di applicazione fra le reti WSN e le reti internet, per l'appunto basate su HTTP.

Fra le varie caratteristiche specificate da CoAP vi è la presenza di nodi sensori che svolgono la funzione di *proxy*. Essendo uno standard relativamente recente le implementazioni software di questo protocollo per alcuni sistemi operativi non sono ancora giunte allo stadio finale di completa conformità ed inoltre alcune funzionalità non sono state ancora sviluppate. E' il caso dei nodi *proxy*; essi, inoltre, per come definiti all'interno dello standard, presentano alcune limitazioni che, in particolari scenari applicativi, possono causare prestazioni scadenti in termini di consumo energetico e ritardo. Scopo di questa tesi è il progetto e lo sviluppo di un *proxy-avanzato* per reti CoAP che permetta di superare tali limitazioni. La piattaforma *hardware* prescelta per lo sviluppo è la TelosB di Memsic mentre il sistema operativo utilizzato è TinyOS. Verranno descritte nello specifico le modifiche apportate rispetto al proxy definito dallo standard e le prestazioni valutate in specifici scenari applicativi. L'analisi delle prestazioni è stata svolta su un *testbed* reale composto da nodi sensore di tipo TelosB in topologie di rete *multi-hop*.

In questa tesi verranno introdotte le reti di sensori, analizzandone caratteristiche e fattori critici, soffermandosi su alcuni protocolli di livello fisico, MAC nonché di livello di rete divenuti ormai standard di riferimento, proseguendo con la descrizione del protocollo di livello applicativo CoAP. Dopo aver analizzato le piattaforme hardware e software utilizzate verranno descritte nel dettaglio le caratteristiche del *proxy* avanzato concludendo con l'analisi delle prestazioni.

Nel dettaglio la tesi è organizzata come segue:

- Capitolo 1: introduce le reti di sensori e descrive ed analizza il protocollo di livello fisico e MAC IEEE 802.15.4, ed i protocolli di livello di rete ZigBee e 6LoWPAN.
- Capitolo 2: descrive le caratteristiche del protocollo di livello applicazione CoAP.

- Capitolo 3: analizza le limitazioni dei nodi *proxy* previsti dallo standard CoAP, quindi descrive lo sviluppo di un *proxy* avanzato che permetta di superare tali limitazioni.

- Capitolo 4: descrive le piattaforme *hardware* e *software* utilizzate per lo sviluppo ed il test delle prestazioni del *proxy* avanzato. Nello specifico vengono descritti la piattaforma *hardware* TelosB, il sistema operativo TinyOS ed il simulatore Cooja.

- Capitolo 5: analizza le prestazioni del *proxy* avanzato sviluppato in questa tesi, in una determinata topologia di rete, considerando diversi scenari applicativi e confrontandolo con il *proxy* standard previsto da CoAP.

- Capitolo 6: conclude il lavoro analizzando i risultati ottenuti e proponendo possibili sviluppi futuri per rendere il *proxy* più facilmente adattabile a differenti scenari applicativi.

Capitolo 1

Wireless Sensor Networks

In questo capitolo verranno introdotte le reti di sensori, analizzandone caratteristiche e fattori critici, i principali scenari applicativi, ed i parametri da tenere in considerazione nella progettazione di una rete WSN. Dopo una rapida analisi degli elementi costitutivi di un nodo sensore, verrà analizzato il protocollo di livello fisico e MAC 802.15.4 divenuto ormai uno standard di riferimento. Si passerà quindi all'analisi dei protocolli di livello superiore quali ZigBee e 6LoWPAN, quindi a cenni sui principali protocolli di *routing* utilizzati da questi protocolli.

1.1 Scenari applicativi

Grazie allo sviluppo della tecnologia microelettronica negli ultimi anni si sono potuti sviluppare dispositivi di ridotte dimensioni e a basso consumo energetico in grado di comunicare fra di loro all'interno di una rete *wireless*, effettuare delle rilevazioni sull'ambiente esterno ed inviare tali dati attraverso la rete. Uno dei parametri fondamentali di questa tipologia di nodi è l'elevata autonomia; tali dispositivi pertanto presentano capacità di calcolo limitate, nonché un raggio di trasmissione ridotto. Una rete di sensori è composta da un elevato numero di dispositivi o *mote* che vengono posizionati in prossimità di un evento da osservare. La disposizione dei sensori non deve essere necessariamente predeterminata in modo tale che i sensori possano essere schierati anche in zone non facilmente accessibili; per far ciò è

necessario che i sensori utilizzino dei protocolli di rete in grado di auto-organizzarsi. I nodi, inoltre, sono dotati di capacità di elaborazione in modo che il carico di lavoro venga distribuito fra i vari sensori sparsi in rete, i quali, anziché inviare dati grezzi possono svolgere delle operazioni di elaborazione sui dati raccolti ed inviare solo i dati d'interesse già elaborati.

Per realizzare tipologie di rete che soddisfino questi requisiti non è possibile affidarsi a tecniche e protocolli in uso in altri scenari di rete in quanto le reti di sensori differiscono da queste ultime, come riportato in [1], per i seguenti fattori:

- I nodi sensori hanno capacità computazionali, di memoria ed energetiche ridotte;
- Il numero di nodi in una rete di sensori può essere di diversi ordini di grandezza più grande rispetto al numero nodi presenti in altre tipologie di rete come le reti ad-hoc;
- La densità spaziale dei sensori può essere molto elevata;
- I sensori sono soggetti a guasti;
- La topologia della rete può variare velocemente;
- I nodi sensori potrebbero non avere un identificativo globale (`ID`) a causa dell'elevato *overhead* e numero di nodi in rete.

Esistono numerose tipologie di *mote* in commercio [2] in grado di misurare le più varie condizioni ambientali; alcune delle tipologie di sensori di cui essi sono dotati includono: sensori di temperatura, umidità, luminosità, pressione, rumore, movimenti di veicoli, analisi del suolo; alcuni sono in grado di misurare velocità, direzione e distanza di oggetti. Questa grande varietà di sensori fa sì che siano molteplici gli scenari applicativi in cui i *mote* possono essere utilizzati. Essi comprendono:

- **Applicazioni militari** [3, 4]: la possibilità di schierare i nodi senza eccessive preoccupazioni sulla topologia che dovranno assumere, unito al costo spesso contenuto dei sensori, rende il loro utilizzo ideale in ambienti ostili come gli scenari bellici: l'eventuale distruzione di alcuni dispositivi, infatti, non influirà negativamente sui costi e sul funzionamento della rete grazie alla capacità di auto-organizzazione delle WSN. Alcuni utilizzi in questo campo possono riguardare: la sorveglianza delle forze amiche ed il loro

equipaggiamento e munizionamento, il controllo dei campi di battaglia anche in ambienti ostili grazie alla possibilità di schierare i sensori per via aerea, il riconoscimento delle forze nemiche [5] ed il riconoscimento di attacchi chimici e biologici.

- **Applicazioni ambientali**: uno dei principali campi di utilizzo delle reti di sensori è il monitoraggio ambientale. Grazie ai costi contenuti ed all'elevata autonomia, eventualmente uniti a meccanismi cosiddetti di *energy scavenging* che si occupano di *racimolare* energia per mezzo di fonti alternative come ad esempio piccoli pannelli solari, i sensori possono essere schierati in vasti ambienti, popolandoli densamente e avendo garanzia di anni di funzionamento. Alcune delle principali applicazioni riguardano il rilevamento di incendi nelle foreste per mezzo dei sensori a bordo [6, 7]; la mappatura della biodiversità ambientale che si affianca a quella effettuata via satellite per garantire una granularità maggiore [8, 9]; il monitoraggio dello spostamento di insetti, piccoli animali ed uccelli [10]; il rilevamento di inondazioni per mezzo di sensori in grado di rilevare la quantità di precipitazioni, il livello dell'acqua ed il tempo in atto inviando i dati ad un *database* centralizzato; l'agricoltura di precisione, ovvero il monitoraggio di pesticidi nell'acqua ed il livello d'inquinamento dell'aria.

- **Applicazioni mediche**: fra le applicazioni in ambito medico vi sono il monitoraggio a distanza di dati fisiologici [11], la localizzazione di dottori e pazienti all'interno di ospedali tramite sensori che vengono applicati direttamente sulla persona [12], sensori che monitorino la somministrazione di medicinali rilevando eventuali allergie.

- **Applicazioni in ambito domestico**: vari dispositivi presenti in casa possono connettersi alla rete e comunicare fra di loro per offrire funzioni *smart* interagendo con le persone [13]; diversi sensori posizionati nelle varie stanze dell'abitazione possono rilevare la presenza di persone all'interno dell'abitazione, studiarne le abitudini e attivare/disattivare dispositivi nonché sistemi di climatizzazione e riscaldamento al fine di minimizzare gli sprechi di energia [14].

- Altre applicazioni: includono monitoraggio di edifici, controllo di qualità sui prodotti, rilevazione e monitoraggio di furti di auto [15], controllo del clima in ambienti di lavoro,

etc.

1.2 Progettazione di una rete WSN

Il progetto di una rete di sensori richiede il soddisfacimento di diversi vincoli e requisiti, spesso contrastanti, che dipendono dallo specifico scenario applicativo. I principali fattori di cui si deve tenere conto includono:

- Tolleranza ai guasti: i sensori di una rete WSN possono smettere di funzionare correttamente a causa di molteplici fattori: mancanza di energia, interferenza radio, danneggiamento (accidentale o meno). A seconda della criticità dell'applicazione diviene più o meno importante la resistenza delle funzionalità di rete qualora alcuni sensori smettano di funzionare, ed eventualmente il tempo necessario per ristabilire il corretto funzionamento della rete: un malfunzionamento di alcuni minuti di una rete composta da sensori atti a rilevare presenze nemiche in ambiente bellico è ben diverso dal malfunzionamento di alcuni minuti della rete di sensori che misura la temperatura in un'abitazione.

- Scalabilità: i protocolli di rete da utilizzare devono tenere conto del numero di nodi che potrebbero entrare a far parte della rete, la quale può essere delle dimensioni più disparate: da poche unità di nodi sensore fino a centinaia se non migliaia.

- Basso costo unitario: visto l'elevato numero di sensori che può costituire una rete WSN, è di fondamentale importanza il costo del singolo sensore sicché il costo totale della rete di *mote* si mantenga contenuto.

- Limitazioni hardware: le dimensioni contenute dei sensori fanno sì che lo spazio a disposizione per la sorgente energetica sia molto limitato, con un impatto sulla quantità di energia immagazzinata. Si rende quindi necessario che l'assorbimento energetico dovuto alle fasi di *sensing*, elaborazione dati e trasmissione radio sia quanto più possibile limitato; ciò impone dei requisiti particolari sull'*hardware* da utilizzare a bordo.

- Robustezza all'ambiente di lavoro: essendo molto vari gli scenari applicativi, altrettanto varie risultano le condizioni ambientali in cui i sensori si troveranno ad operare: da semplici edifici ad ambienti molto ostili con elevata presenza di polveri, temperature estreme ed interferenze radio. I sensori devono quindi essere dotati di particolari doti di robustezza.

- Mezzo di trasmissione: i vari sensori sono connessi fra di loro in rete per mezzo di un sistema senza fili, tipicamente tramite un'interfaccia radio. Si rende quindi necessaria la scelta di un mezzo di trasmissione che possa essere utilizzato in tutto il mondo. La scelta da parte della maggior parte delle aziende produttrici di sensori è ricaduta sull'utilizzo di una banda ISM a 2.4 GHz, libera in tutto il mondo, sulla quale non è imposto l'utilizzo di un protocollo specifico e che ben si adatta ai requisiti di potenza e ridotte dimensioni dei nodi.

- Consumo di energia: a causa delle dimensioni ridotte, le fonti energetiche che equipaggiano i nodi sensore sono di capacità contenuta (da $<$500 mAh a 2000 mAh) diviene quindi fondamentale che il consumo di energia sia il più parco possibile. I tre ambiti principali in cui avviene il consumo di energia sono la fase di *sensing* ovvero i momenti in cui il nodo sfrutta i sensori a bordo per catturare dati sull'ambiente circostante, la fase di *elaborazione dati* dove viene sfruttato il microprocessore a bordo per l'elaborazione dei dati e la fase di *comunicazione radio* che è la più dispendiosa in termini di consumo energetico come mostrato da [1].

1.3 Anatomia di un mote

Un generico nodo sensore è caratterizzato dallo schema a blocchi mostrato in Figura 1.1.

Le parti costituenti sono:

- Unità di alimentazione: a seconda delle dimensioni del sensore può essere costituita da una batteria a bottone dalla capacità di circa 500 mAh ,o nei sensori di dimensioni maggiori, da batterie stilo che offrono una capacità di circa 2000 mAh. Le batterie utilizzate sono spesso di tipo non ricaricabile in quanto garantiscono una più lunga

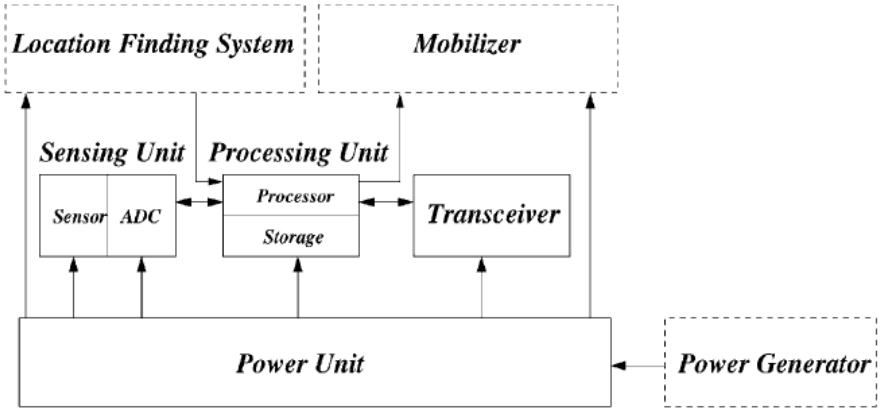


Figura 1.1: Schema a blocchi di un nodo sensore.

durata nel tempo (corrente di auto-scarica inferiore rispetto alle tipologie ricaricabili) oltre al fatto che spesso non è possibile recuperare i sensori che di fatto diventano usa-e-getta.

- Unità generatore di energia (opzionale): i sensori dotati di tale dispositivo sono detti *energy scavenging*. Tipicamente il dispositivo che permette di utilizzare fonti alternative di energia è costituito da un pannello solare. L'energia catturata viene immagazzinata in condensatori o batterie ricaricabili.

- Unità di processamento: è costituita da un micro-controllore (MCU) e da una o più unità di memoria. Il micro-controllore è appositamente studiato per contenere il consumo energetico pertanto è dotato di potenza di calcolo limitata. Sono inoltre dotati di modalità di *sleep* in cui l'assorbimento di corrente è dell'ordine dei nA. Affiancano l'MCU unità di memoria ROM e RAM per l'immagazzinamento e l'esecuzione di sistema operativo, programmi e dati rilevati.

- Unità di *sensing*: i dati vengono rilevati tramite sensori. Tipicamente i dati rilevati dai sensori sono di tipo analogico e vengono convertiti in formato digitale tramite un ADC, quindi inviati al processore che si occuperà di elaborarli ed trasmetterli in rete.

- Unità radio: è un chip ricetrasmettitore costruito per limitare il consumo energetico. E' tipicamente caratterizzato da un basso assorbimento nella modalità *sleep*. Spesso

l'interfaccia è di tipo *half-duplex* ovvero non è in grado di ricevere pacchetti durante la fase di trasmissione.

- Sistema di localizzazione (opzionale): alcuni sensori devono essere in grado di poter trasmettere la propria posizione, pertanto possono essere dotati di sistemi di localizzazione come il GPS. L'utilizzo di questo sistema aumenta notevolmente il consumo energetico da parte del sensore.

- *Mobilizer* (opzionale): a volte si rende necessario poter spostare il sensore nell'ambiente in cui si trova, pertanto viene dotato di questo dispositivo, anch'esso come il sistema di localizzazione avaro di risorse energetiche.

1.4 Livello fisico e MAC: IEEE 802.15.4

Rispettare tutti i requisiti analizzati nel Paragrafo 1.2 richiede l'utilizzo di architetture e protocolli di rete specifici. Nell'ultimo decennio sono state sviluppate numerose soluzioni per la comunicazione nelle reti di sensori che sfruttano i più vari mezzi di comunicazione [16]. Esistono infatti tecniche di comunicazione ottiche, acustiche e ad induzione elettromagnetica; le soluzioni di maggior successo tuttavia usano comunicazioni a radiofrequenza, ovvero la trasmissione di onde elettromagnetiche su bande RF. Le principali tecnologie RF utilizzate ricadono in tre tipologie: *narrow-band* ovvero a banda stretta, che si prefigge di ottimizzare la banda utilizzando tecniche di modulazioni M-arie; a questa si contrappongono le tecniche *spread-spectrum* ed *ultra-wide-band* che invece utilizzano per la trasmissione dell'informazione bande molto più ampie di quelle effettivamente necessarie.

Alla categoria *spread-spectrum*, ed in particolare sfruttando la tecnica DSSS, appartiene il protocollo che di fatto è diventato uno standard nelle reti di sensori: IEEE 802.15.4 [17]. Questo protocollo è stato stato sviluppato dalla IEEE e si occupa di definire i livelli PHY e MAC dello *stack*. Nato nel 2003, 802.15.4 può operare su diverse frequenze, tutte *license-free*, delle quali una disponibile a livello mondiale. La lista delle frequenze, assieme ad alcuni parametri di lavoro, è mostrata in Tabella 1.1.

Banda RF	Range Frequenza (MHz)	Modulazione	Data Rate	Numero Canale	Area Geografica
868 MHz	868.3	BPSK	20 kb/s	0 (1 canale)	Europa
915 MHz	902-928	BPSK	40 kb/s	1-10 (10 canali)	America, Australia
2.4 GHz	2405-2480	O-QPSK	250 kb/s	11-26 (16 canali)	Globale

Tabella 1.1: Lista delle frequenze supportate da IEEE 802.15.4.

I *range* di frequenza ad 868 MHz e 915 MHz offrono dei vantaggi rispetto a quello a 2.4 GHz come la presenza di meno utenti (e quindi dispositivi che affollano la banda) e la trasmissione che risente meno degli assorbimenti e delle riflessioni; all'atto pratico però è la banda a 2.4 GHz ad essere la più utilizzata in quanto è disponibile a livello mondiale, offre un *data-rate* più elevato e, pertanto, consumi energetici in trasmissione minori poiché la radio resta in modalità TX per un tempo inferiore.

Il protocollo IEEE 802.15.4 include inoltre dei meccanismi che permettono la scelta del canale di comunicazione, selezionando quello con minore interferenza. Il raggio di trasmissione per dispositivi con potenza di 0 dBm è di circa 100 m in spazi aperti, che si riducono a circa 30 m in ambienti chiusi a causa di assorbimento e riflessione del segnale dovuta a muri ed oggetti.

Come detto, il punto debole dei nodi sensore è l'autonomia causata da una fonte di energia di capacità contenuta; gran parte dell'energia consumata dai *mote* è dovuta alla fase di trasmissione radio. Per limitare il consumo energetico dovuto a questo fattore il *duty cicle* di 802.15.4 è molto basso sicché i tempi di trasmissione sono molto corti e gli intervalli fra una trasmissione e l'altra molto lunghi. Contribuisce inoltre a ridurre il consumo energetico la tipologia di modulazione utilizzata.

1.4.1 Tipologie di dispositivi

I nodi di una rete IEEE 802.15.4 non svolgono tutti la stessa funzione pertanto ne esistono di varie tipologie:

- *PAN-Coordinator*: è un nodo che deve essere sempre presente in rete ed il cui ruolo deve essere svolto da un solo sensore. I suoi compiti includono: la ricerca di un canale radio adatto, l'assegnazione di un identificativo alla rete (PAN-ID) e di un indirizzo

a se stesso, la gestione delle richieste di iscrizione alla rete da parte degli altri nodi, l'inoltro dei messaggi da un nodo ad un altro in alcune topologie.

- *Local-Coordinator*: è presente nelle topologie ad albero, ve ne può essere più di uno e come il PAN-C deve gestire le richieste di iscrizione alla rete e l'inoltro dei pacchetti.
- *End-device*: questa tipologia di nodi non svolge funzioni di gestione della rete ma si limita a comunicare con gli altri nodi della rete, in alcune topologie esclusivamente con il proprio coordinatore.

A queste tipologie di nodi logici corrispondono due tipologie di sensori fisici che differiscono per caratteristiche hardware e software:

- *Full Function Device* (FFD): è un dispositivo in grado di eseguire tutte le funzionalità offerte dal livello MAC di 802.15.4, pertanto i dispositivi *PAN-Coordinator* e *Local-Coordinator* appertengono a questa categoria
- *Reduced Function Device* (RFD): è in grado di eseguire un set ridotto di funzioni 802.15.4 in quanto dotato di *hardware* meno prestante; un nodo con tali caratteristiche pertanto può essere esclusivamente un *end-device*.

1.4.2 Topologie di rete

IEEE 802.15.4 supporta diverse topologie di rete. Esse non sono gestite direttamente da questo protocollo ma da protocolli di livello superiore come ad esempio ZigBee. Una rete deve essere formata da minimo due nodi, almeno uno dei quali deve essere un *PAN-coordinator*.

1.4.2.1 Topologia a stella

La topologia basilare è la cosiddetta «a stella». Essa è mostrata in Figura 1.2. Al centro della rete vi è un sensore che svolge il ruolo di *PAN-coordinator*, gli altri dispositivi prendono il nome di *end-device*. Gli *end-device* possono comunicare esclusivamente con il *PAN-coordinator*, sarà pertanto compito di quest'ultimo inoltrare i pacchetti alle diverse destinazioni. Come è intuibile questo sistema presenta alcuni svantaggi: il nodo PAN-C svolge un compito più gravoso rispetto agli altri sensori; si deve pertanto tenerne conto nel

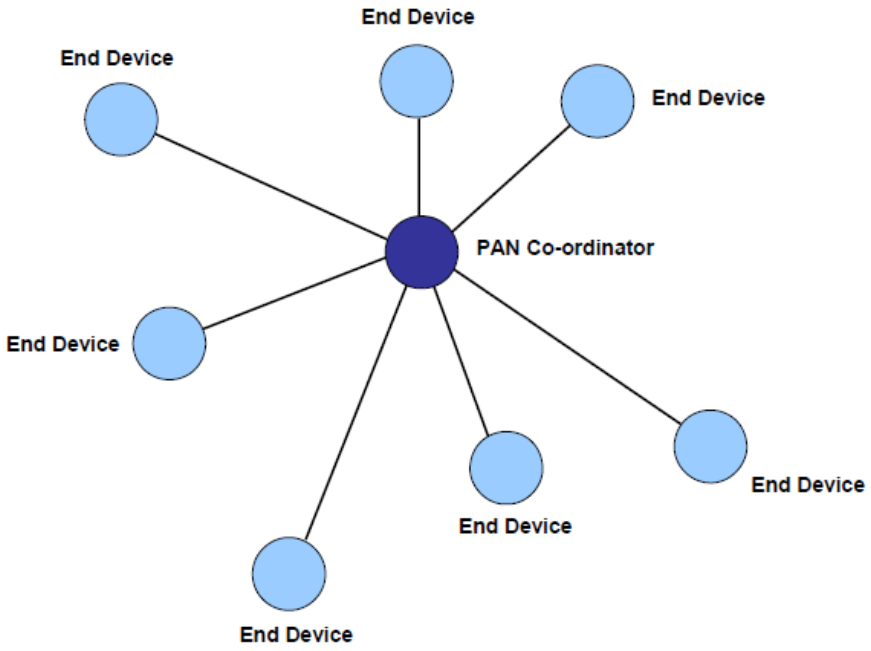


Figura 1.2: Topologia a stella.

dimensionare l'*hardware* di questo dispositivo; se il traffico è ingente esso potrebbe inoltre divenire un collo di bottiglia per la rete nonché, in caso di perdita di connessione fra esso ed il nodo destinatario di un pacchetto, non sarà possibile far pervenire il messaggio a quest'ultimo per mezzo di un altro percorso.

1.4.2.2 Topologia ad albero

Lo schema di questa topologia è mostrato in Figura 1.3. La topologia è caratterizzata da:

- un nodo radice, ossia il *PAN-Coordinator*
- dei nodi padre, ovvero i *Co-ordinator*
- dei nodi foglia, gli *end-device*.

Ogni nodo può comunicare solo con il proprio nodo padre e con i propri nodi foglia; il nodo padre inoltre svolge la funzione di coordinatore locale per i propri nodi foglia.

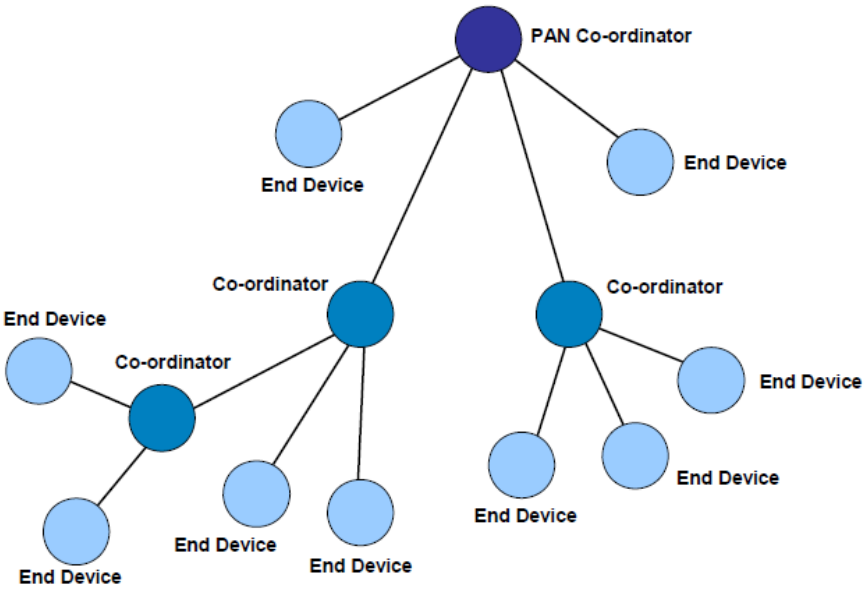


Figura 1.3: Topologia ad albero.

1.4.2.3 Topologia mesh

E' mostrata in Figura 1.4. Anche la topologia *mesh* è caratterizzata da un nodo centrale che svolge la funzione di *PAN-Coordinator*; a differenza della topologia a stella però, gli altri nodi della rete possono comunicare fra di loro creando in questo modo *route* alternative per una medesima destinazione. In caso di interruzione di un *link*, quindi, un messaggio può comunque arrivare a destinazione, venendo inoltrato di nodo in nodo.

1.4.3 Stack protocollare

Come accennato in 1.4 lo standard IEEE 802.15.4 riguarda solamente i primi due livelli dello *stack* protocollare di rete: il livello fisico ed il livello di collegamento, come mostrato nella Figura 1.5.

1.4.3.1 Livello fisico (PHY)

Il livello fisico si occupa da un lato di interfacciarsi con il mezzo di trasmissione fisico, in questo caso la radio, dall'altro dello scambio dei dati con il livello superiore (livello MAC). Si occupa quindi di modulare/de-modulare i bit, della sincronizzazione dei pacchetti, di

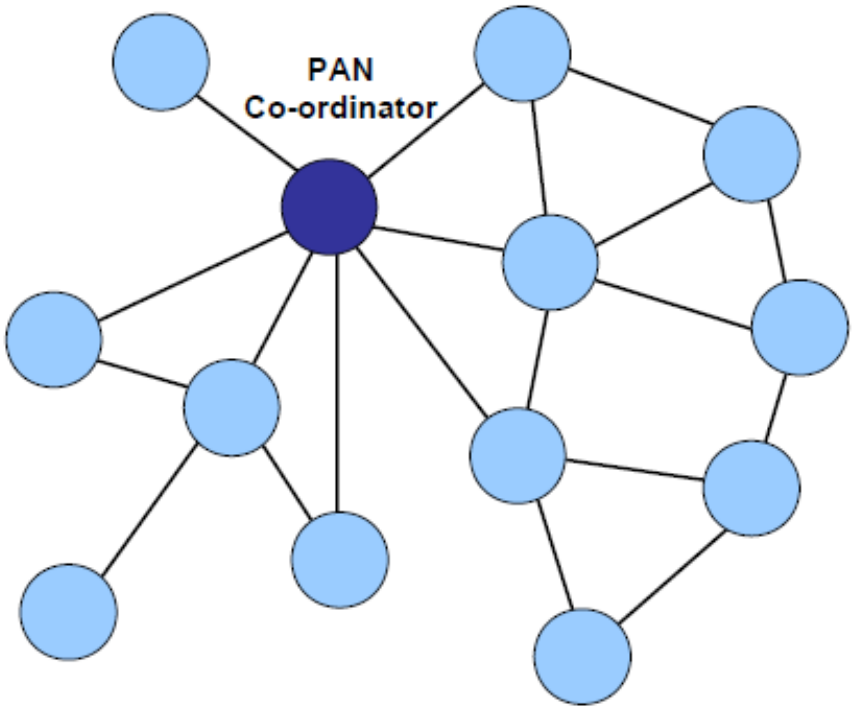


Figura 1.4: Topologia *mesh*.

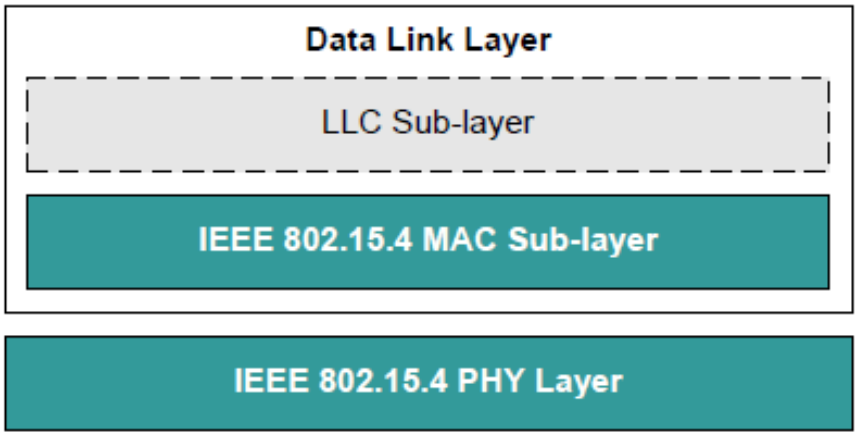


Figura 1.5: *Stack* protocollare 802.15.4.

ottenere i dati dal livello MAC (trasmissione) nonché di fornirli (ricezione) così come fornire dei meccanismi che permettano il controllo della radio.

1.4.3.2 Livello di collegamento (sottolivello MAC)

I compiti svolti dal livello MAC sono:

- Fornire servizi per associare/dissociare i dispositivi con la rete
- Fornire controllo d'accesso ai canali di comunicazione
- Generazione dei *Beacon*
- Gestione dei *Guaranteed Timeslot* (GTS)
- Fornire i dati ai livelli superiori
- Fornire meccanismi per il controllo della radio e delle funzionalità di rete

1.4.4 Caratteristiche funzionali

Fra i compiti svolti da 802.15.4 vi sono

- Gestione del canale: il protocollo deve gestire le varie problematiche riguardanti il canale, ovvero la loro allocazione, il controllo della disponibilità del canale e la preservazione da possibili trasmissioni.
- Assegnazione del canale: come specificato nella Tabella 1.1 il protocollo IEEE può lavorare a diverse frequenze ognuna della quali può fornire più di un canale di comunicazione utilizzabile. E' compito del protocollo effettuare una scansione dei vari canali sfruttando un meccanismo di *Energy Detection Scan* in modo tale che i livelli superiori possano scegliere quello più adatto, tipicamente quello meno rumoroso. Quando inoltre un dispositivo deve entrare a far parte di una rete, esso deve conoscere quale canale viene usato dalla rete nonché il PAN ID della stessa. Per far ciò può procedere in modo attivo inviando dei *beacon* da far rilevare ai coordinatori di rete, oppure in modalità passiva ascoltando il canale per i *beacon* trasmessi dai coordinatori.

- Reiezione dei canali non utilizzati: poiché in alcune frequenze supportate dal protocollo 802.15.4 è presente più di un canale utilizzabile, nel caso venga rilevato nei canali adiacenti un livello di segnale pari od inferiore a quello utilizzato e nei canali non adiacenti un segnale di livello fino a 30 dB più forte, devono essere attuate tecniche di reiezione di tali segnali.

- *Clear Channel Assesment*: Quando viene utilizzato il protocollo di accesso multiplo CSMA/CA, è necessario assicurarsi che il canale di comunicazione non sia occupato dalla trasmissione di un altro dispositivo; pertanto il protocollo prevede un periodo di ascolto del canale per assicurarsi che questo sia libero; se l'esito dell'ascolto è positivo la trasmissione viene avviata, in caso negativo viene calcolato un periodo di tempo casuale durante il quale il sensore rimane in stand-by e, scaduto il quale, esso ritenterà l'ascolto; questo intervallo di tempo prende il nome di *back-off period*.

1.4.5 Gestione dei dispositivi

Fra i vari compiti di cui il protocollo 802.15.4 deve occuparsi vi è anche la gestione dei dispositivi. Sono quindi implementati dei meccanismi per eseguire l'associazione di un dispositivo ad una rete così come la sua dissociazione. L'associazione è effettuata tramite l'invio di una richiesta di associazione ad un nodo coordinatore che deve essere stato in precedenza trovato tramite l'utilizzo di un sistema di scansione attiva o passiva. Altro aspetto che riguarda la gestione dei dispositivi è la scelta dei nodi coordinatori: può infatti accadere che in rete non sia presente un solo dispositivo di tipologia FFD, pertanto la scelta di quale dovrà diventare, ad esempio, *PAN-Coordinator* può essere predeterminata oppure lasciata al caso. Può inoltre accadere che un nodo perda contatto con il proprio coordinatore, ciò può essere dovuto sia ad un malfunzionamento del nodo coordinatore che a mutate condizioni del canale di trasmissione; in questo caso il nodo rimasto «orfano» può eseguire un *Orphan Channel Scan* per tentare un ri-collegamento con il nodo coordinatore originario.

1.4.6 Indirizzamento dei dispositivi

Sono previsti due tipi di indirizzamenti:

- Indirizzo MAC: di lunghezza pari a 64 bit, viene memorizzato all'interno dei *chip* di rete direttamente dal produttore dell'*hardware*. Gli indirizzi vengono forniti dalla IEEE pertanto non possono esistere due dispositivi con il medesimo indirizzo.

- Indirizzo breve: di lunghezza pari a 16 bit, è un indirizzo locale che identifica il nodo all'interno della rete e che viene allocato dal coordinatore di rete. Essendo più corto dell'indirizzo MAC consente di ridurre la dimensione dei pacchetti e quindi ottimizzare il consumo di banda nonché quello energetico.

1.4.7 Trasferimento dati

Poiché un aspetto fondamentale delle reti di sensori è la longevità della rete stessa, i nodi foglia spendono parecchio tempo in modalità *sleep* al fine di preservare la riserva di carica. Pertanto quando un nodo coordinatore invia un messaggio per un nodo foglia, non sempre questo è in ascolto ed il messaggio può quindi andare perduto. In questi casi, quando il nodo foglia ritorna attivo, esso deve occuparsi di effettuare un *polling* al nodo coordinatore per verificare se vi siano messaggi in sospeso ad esso destinati. Dopo la ricezione di un messaggio può essere inviato un messaggio di *acknowledgement*.

1.4.8 Modalità di funzionamento

Oltre che per entrare a far parte di una rete, i messaggi di *beacon* possono essere utilizzati per scandire l'accesso al canale. Vengono ora brevemente descritte le due modalità di funzionamento di una rete 802.15.4.

1.4.8.1 Modalità beacon enabled

Questa modalità prevede che il coordinatore di rete invii periodicamente dei messaggi di *beacon*. I *beacon* hanno il compito di marcare l'inizio e la fine di una *supertrama* e quindi permettere la sincronizzazione dei vari nodi. La struttura di una supertrama è mostrata in Figura 1.6. La supertrama contiene 16 *time-slot* all'interno dei quali i sensori possono trasmettere. Questi *slot* non sono riservati a specifici sensori pertanto la comunicazione all'interno di essi avviene a contesa, sfruttando il protocollo di accesso multiplo CSMA/CA.

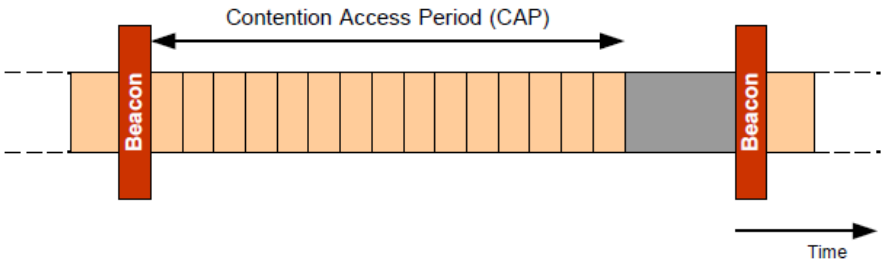


Figura 1.6: Supertrama *beacon enabled*.

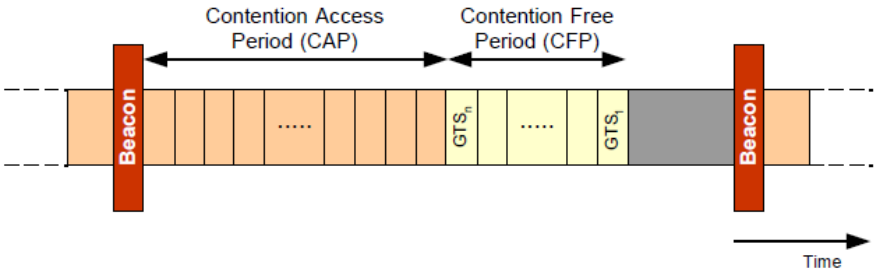


Figura 1.7: Supertrama *beacon enabled* con GTS.

Per tale motivo questa porzione di supertrama prende il nome di *Contention Access Period*. Al termine del CAP vi è un intervallo in cui non vengono scambiati dati e durante il quale i nodi possono entrare in modalità *sleep* per risparmiare energia. E' anche possibile riservare alcuni di questi *slot* a specifici sensori, è il caso della supertrama mostrata nella Figura 1.7. La porzione di *slot* riservata viene chiamato *Contention Free Period* (CFP) e l'assegnazione degli *slot* viene specificata tramite i *beacon*. L'utilizzo dei *beacon* costringe i vari nodi a «svegliarsi» periodicamente per l'ascolto, il che provoca un incremento dei consumi energetici.

1.4.8.2 Modalità non-beacon enabled

La modalità che non utilizza *beacon* permette quindi un risparmio di energia in quanto un sensore può uscire dalla fase di *sleep* solamente quando deve trasmettere qualcosa, il che può avvenire con una frequenza molto inferiore al periodo della supertrama. Nel caso in cui un nodo debba ricevere dei dati può effettuare un *polling* al dispositivo coordinatore per conoscere se vi siano pacchetti in attesa di essere consegnati.

1.4.9 Routing

Il protocollo di *routing* dei pacchetti utilizzato dipende dalla topologia della rete.

- Topologia a stella: poiché i nodi possono comunicare esclusivamente con il nodo padre, i pacchetti vengono ad esso inviati, il quale si occuperà di spedirli al nodo figlio di destinazione.

- Topologia ad albero: in questo scenario i nodi padre, dopo aver ricevuto il messaggio da uno dei loro nodi foglia, analizzano il destinatario e, se esso appartiene ad uno dei propri nodi foglia, provvedono ad inviarglielo, altrimenti il messaggio viene inviato al nodo padre di livello superiore. Questo meccanismo implica che i vari nodi padre abbiano una *tabella di routing* aggiornata, contenente l'indirizzo di tutti i nodi figli. Un aspetto negativo di questo sistema è che due nodi pur vicini fisicamente potrebbero appartenere a due rami dell'albero diversi ed il messaggio, per giungere a destinazione, necessiterebbe di numerosi *hop*.

- Topologia *mesh*: non essendovi una topologia ben precisa i nodi non sono in grado di conoscere la mappa della rete, per questo motivo è possibile utilizzare la trasmissione *broadcast* che si occupa di inoltrare i pacchetti a tutti i nodi della rete, oppure altri protocolli di *routing* specifici elaborati nei livelli di *stack* superiore che verranno analizzati in seguito.

1.4.10 Sicurezza

IEEE 802.15.4 è in grado di occuparsi anche della sicurezza; esso offre tre diversi livelli di sicurezza:

- Modalità `Unsercured`: in questa modalità nessun meccanismo di sicurezza viene implementato.

- Modalità `ACL (Access Control List)`: in questa modalità i dispositivi sono in possesso di una lista contenente gli indirizzi dei nodi con cui sono abilitati a scambiare pacchetti. All'arrivo di un pacchetto il dispositivo analizza l'indirizzo sorgente e se

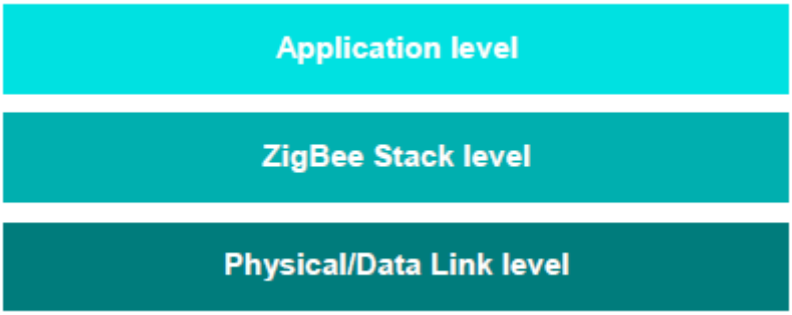


Figura 1.8: *Stack* protocollare di una rete WSN ZigBee.

esso è presente nella lista, fornisce il contenuto dei pacchetti ai livelli superiori che si occuperanno della gestione del messaggio.

- Modalità `Secured`: La modalità *secured* comprende le funzionalità di: *controllo di accessi* (come descritto nella modalità ACL), *cifratura* (i dati sono cifrati all'origine e decifrati alla destinazione utilizzando la medesima chiave), *integrità* (viene aggiunto un *Message Integrity Code* per rilevare eventuali tentativi di manomissione del contenuto dei pacchetti), *sequenza dei messaggi* (tramite un contatore si è in grado di valutare quanto recente sia un messaggio ricevuto).

1.5 Livello di rete ed applicativo: ZigBee vs 6LoWPAN

Come finora analizzato il protocollo IEEE 802.15.4 si occupa di definire i meccanismi di funzionamento dei livelli di *stack* protocollare più bassi. I livelli di rete più elevati vengono invece definiti in altri protocolli come ZigBee [18] e 6LoWPAN [19].

1.5.1 ZigBee

ZigBee è una specifica di protocolli di rete ad alto livello. E' stato sviluppato dalla *ZigBee Alliance*; gli studi sono iniziati nel 1998 mentre la specifica 1.0 è stata rilasciata nel 2005. Nella Figura 1.8 viene mostrato lo *stack* di una rete WSN che utilizza ZigBee.

Il livello di *stack* ZigBee serve ad interfacciare i livelli di rete più bassi (PHY e MAC) forniti da 802.15.4 con il livello più alto, ovvero il livello applicazione. I compiti svolti da

ZigBee riguardano l'organizzazione della rete, il *routing* e la sicurezza (cifratura). ZigBee, come 802.15.4, supporta tre diverse topologie di rete: a stella, ad albero e *mesh*. Nel caso di una rete a stella è direttamente lo *stack* di rete ZigBee ad occuparsi dell'inoltro dei pacchetti in rete rendendo l'operazione trasparente al livello applicativo. Se invece non viene utilizzato il livello di rete ZigBee sarà l'applicazione in esecuzione sui nodi coordinatori a dover eseguire le operazioni di inoltro. Anche nel caso di topologia ad albero lo *stack* di rete ZigBee può occuparsi dell'inoltro dei pacchetti in modo del tutto trasparente alla applicazioni. Per quanto riguarda le reti ti tipologia *mesh*, ZigBee sfrutta il protocollo AODV che mette a disposizione una funzione di *route discovery*: il nodo sorgente, per mezzo del suo coordinatore, invia in *broadcast* una richiesta di *route discovery*; il messaggio, che viene inoltrato in rete, contiene l'indirizzo del nodo di destinazione e sarà ricevuto da tutti i coordinatori della rete, uno dei quali nodo padre del nodo di destinazione. Esso quindi risponderà con un messaggio di *route reply* che seguirà il percorso inverso registrando il numero di *hop* e la *qualità dei link* radio. In questo modo i vari coordinatori possono aggiornare le loro tabelle di *routing* inserendo il miglior percorso per la destinazione specificata. Il miglior percorso è tipicamente quello con numero di *hop* minori.

Oltre a fornire un sistema di *route dicovery* ZigBee mette a disposizione servizi di *device & service discovery*: tramite essi un nodo sensore può richiedere informazioni su un altro nodo, quali il proprio indirizzo, la lista di indirizzi dei nodi ad esso associato (in caso il nodo interrogato sia un *coordinatore/router*), le caratteristiche del nodo come la sua tipologia, le sue capacità, il tipo di alimentazione, le applicazioni in esecuzione.

1.5.2 6LoWPAN

Un altro degli aspetti che riveste fondamentale importanza in una rete WSN è l'interoperabilità con altre reti, come ad esempio quella internet basata sul protocollo IP. Un limite di ZigBee è dato dall'utilizzo di un protocollo proprietario per il livello di rete che pertanto necessita di un apposito *gateway* che traduca i pacchetti in formato IP e viceversa. L'utilizzo del protocollo 6LoWPAN permette di superare questa limitazione in quanto esso utilizza direttamente il protocollo IPv6.

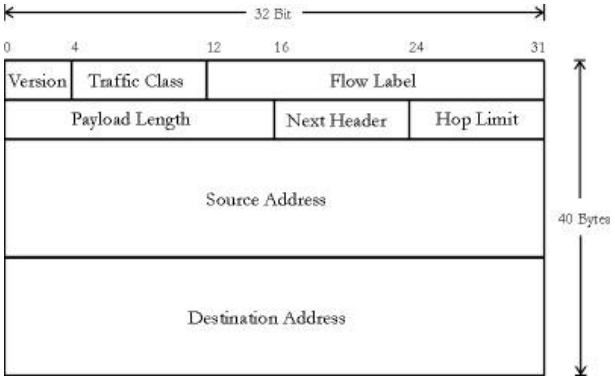


Figura 1.9: *Header* di un pacchetto IPv6.

Figura 1.10: Struttura compressione *header* IPv6 ed UDP.

Il protocollo IPv6 non è stato progettato per reti di sensori senza fili pertanto presenta delle problematiche quando utilizzato su questa tipologia di dispositivi. Per risolvere tali problematiche 6LoWPAN è caratterizzato da un livello di adattamento. Un primo problema è dovuto alla lunghezza dell'*header* IPv6 che è pari a 40 byte, mentre la dimensione di un pacchetto IEEE 802.15.4 è pari a 127 byte dai quali vanno rimossi 25 byte di *overhead*; il che si traduce in 102 byte disponibili per i livelli superiori. Se inoltre si utilizzano protocolli di sicurezza come l'AES-CCM gli ottetti rimanenti sono 81. Come è evidente il solo *header* IPv6 occuperebbe circa il 50% della dimensione disponibile. La soluzione trovata è quella di comprime l'*header* IPv6 ed UDP tramite un sistema *stateless*. L'*header* IPv6 è mostrato in Figura 1.9. I vari campi che lo compongono possono essere compattati (gli indirizzi), trascurati, altri derivati dall'*header* di livello inferiore (es.: la lunghezza del *payload*). Le operazioni di compressione svolte sugli specifici campi dell'*header* sono specificati in un apposito *header* di compressione al quale segue l'*header* compresso (Figura 1.10). Analogamente si procede per la compressione dell'*header* UDP. Nel caso di comunicazioni *link-local*, l'*header* compresso ha dimensione di soli 7 byte a fronte dei 48 byte degli *header* UDP + IPv6. Nel caso di comunicazioni *unicast* globali la dimensione dell'*header* compresso sale a 12 byte.

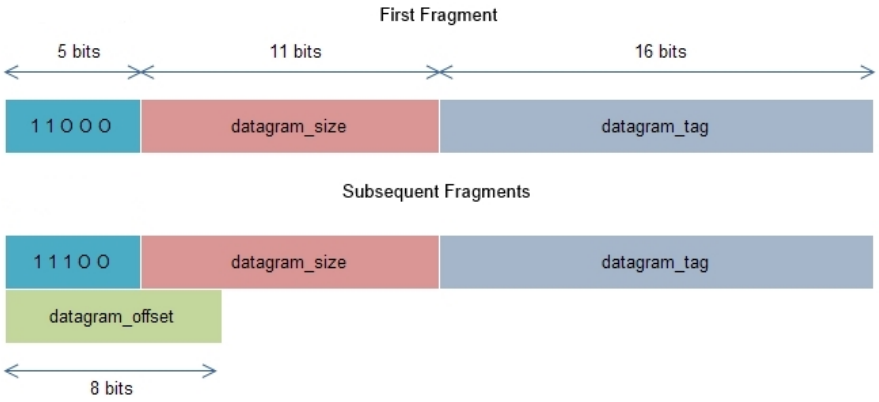


Figura 1.11: *Header* di frammentazione.

Un'altra problematica che insorge volendo utilizzare il protocollo IPv6 su 802.15.4 è la lunghezza minima della MTU di IPv6 che è pari a 1280 byte laddove il *payload* massimo offerto da 802.15.4 è dell'ordine dei 100 byte come descritto in precedenza. Si rende quindi necessario un sistema di frammentazione dei pacchetti IPv6 anch'esso attuato dal livello di adattamento 6LoWPAN. A tale scopo è definito un apposito *header* di frammentazione nel quale vengono specificati la lunghezza in byte dei frammenti, un ID di frammentazione (comune a tutti i frammenti), e l'offset cui il frammento fa riferimento (tralasciato nel primo frammento), Figura 1.11. E' facile intuire che l'utilizzo della frammentazione di pacchetti IPv6 di grandi dimensioni causa una decadenza delle prestazioni, ovvero elevati ritardi, specialmente nel caso in cui uno o più pacchetti contenenti un frammento vengano persi: in questo caso è necessario re-inviare completamente tutti i pacchetti che contengono il pacchetto IPv6 frammentato.

Un altro aspetto di cui 6LowPAN deve occuparsi è il cosiddetto *neighbour discovery* ovvero un meccanismo che permetta ai vari nodi di rete di conoscere quali sono i nodi ed i *router* vicini. Il sistema di *neighbour discovery* utilizzato da IPv6 non è però adatto alle reti WSN in quanto si basa sul concetto tradizionale di link; nelle reti WSN i link possono non essere bidirezionali, i nodi della rete sfruttano spesso lunghi periodi di *sleep* e si preferisce evitare l'utilizzo di messaggi *multicast* come invece avviene nel *Neighbour Discovery* di IPv6. Sono pertanto state apportate alcune ottimizzazioni al protocollo ND IPv6 standard come riportato nello specifico da [20]. Le principali riguardano: le comunicazioni sono attivate

dall'*host* per consentire i periodi di *sleep*, eliminazione della risoluzione di indirizzi di tipo *multicast* da parte dell'*host*, un modello dei *link* e delle *subnet* più appropriato alle reti WSN, distribuzione di prefissi e contesto a livello *multi-hop*, *duplicate address detection* estesa a livello *multi-hop*.

1.6 Routing: IETF RPL

Il protocollo IEEE 802.15.4 non si occupa di gestire il *routing* che viene invece effettuato ad un livello più alto della rete. E' nata quindi l'esigenza di sviluppare un protocollo di *routing* pensato appositamente per le reti di sensori. Esso deve garantire un percosso dei pacchetti breve, quindi un basso consumo di energia e deve essere a conoscenza della topologia globale della rete. L'IETF ha quindi formato un nuovo gruppo di lavoro, il ROLL, per sviluppare un nuovo protocollo di *routing* che prende il nome di RPL (*Routing Protocol for Low power and Lossy networks*) [21]. Esso utilizza un approccio *proattivo* di tipo *distance-vector*: le informazioni di *routing* sono acquisite prima che sia necessario inviare un pacchetto ed ai link di collegamento è associato un costo. I vari *router* salvano localmente le informazioni sul *next hop* all'interno della loro *routing table*.

RPL si basa sulla creazione di DODAG (*Destination Oriented Directed Acyclic Graph*) ovvero grafi aciclici diretti orientati alla destinazione che si basano per la loro creazione su una funzione obiettivo. Essa è caratterizzata da una metrica che può essere l'affidabilità del *link*, la latenza, la banda a disposizione, e da vincoli come ad esempio l'esclusione di alcuni nodi o *link* dal DODAG. I vari grafi vengono creati per mezzo di scambio di messaggi denominati DIO, DAS e DIS.

Capitolo 2

CoAP: Constrained Application Protocol

Questo capitolo descrive il protocollo di livello applicativo CoAP che sta divenendo uno standard di riferimento. Verrà analizzata l'architettura alla base di CoAP, la tipologia e struttura dei messaggi, i servizi offerti e i ruoli che i vari nodi possono svolgere all'interno della rete.

2.1 REST: REpresentational State Transfer

L'utilizzo di servizi *web-based* è ormai diventato di uso comune in moltissime applicazioni; essi sono basati sull'architettura tipica del web ossia REST. Le reti come 6LoWPAN permettono l'invio di pacchetti di grosse dimensioni affidandosi però alla frammentazione dei pacchetti, operazione che influisce pesantemente sull'affidabilità della consegna dei pacchetti. Per tale motivo il protocollo HTTP non è il più adatto in questi scenari; lo scopo di CoAP [22]è quello di realizzare un'architettura REST adatta alle reti di sensori che, come finora visto, sono caratterizzate da nodi con caratteristiche hardware limitate nonché da collegamenti che non fanno dell'affidabilità il loro forte. Lo scopo di CoAP non è quello di comprimere semplicemente il protocollo HTTP affinché sia adatto alle WSN, quanto quello

di creare un sottoinsieme di funzionalità di tipo REST compatibili con HTTP.

L'architettura REST, definita nel 2000 nella tesi di dottorato di Roy Fielding [23], raccoglie un insieme di principi su come deve essere composta l'architettura di rete. Le principali caratteristiche sono:

- *Client-Server*: la tipologia di architettura da utilizzarsi è quella cosiddetta *client-server*, che si trova tipicamente nelle applicazioni *web-based*. Un'entità, il *server*, offre una serie di servizi e resta in ascolto di eventuali richieste per i servizi offerti. Un'altra entità, il *client*, esegue una richiesta per un servizio a cui è interessato. Il *server* alla ricezione della richiesta decide se soddisfare o meno la richiesta ed invia un messaggio di risposta al *client*.

- *Stateless*: un secondo requisito dell'architettura REST è che la comunicazione sia di tipo *stateless* (CSS: *Client Stateless Server*) ovvero ogni richiesta effettuata dal *client* verso il *server* deve contenere tutte le informazioni necessarie affinché il *server* possa comprendere la richiesta; non si devono ovvero sfruttare delle informazioni memorizzate lato *server*. Se da un lato ciò migliora l'affidabilità e la scalabilità, dall'altro può provocare un peggioramento delle prestazioni di rete dovute ad un incremento del traffico causato dall'invio di dati ripetitivi.

- *Cache*: al fine di migliorare le prestazioni, i *client* sono dotati di *cache*; una risposta ottenuta a seguito di una richiesta deve essere contrassegnata come *cacheable* oppure *non-cacheable*. Se una risposta è di tipo *cacheable* un client può sfruttare nuovamente quella risposta presente in *cache* per le successive *request* equivalenti. Ciò permette di migliorare la scalabilità, diminuire il traffico in rete ed inoltre avere una minore latenza nell'ottenere risposte in quanto si sfrutta il dato locale. D'altro canto se non usata correttamente i dati in *cache* potrebbero contenere informazioni ormai non più aggiornate.

- *Risorse*: le varie informazioni che un *server* mette a disposizione di un client sono definite *risorse*. Esse sono identificate da un *resource identifier*; tramite esso è possibile accedere alla risorsa o modificarla/crearla.

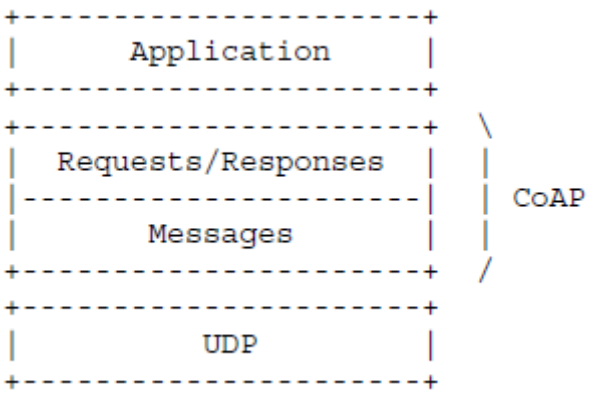


Figura 2.1: Livelli CoAP.

2.2 Tipologia di messaggi CoAP

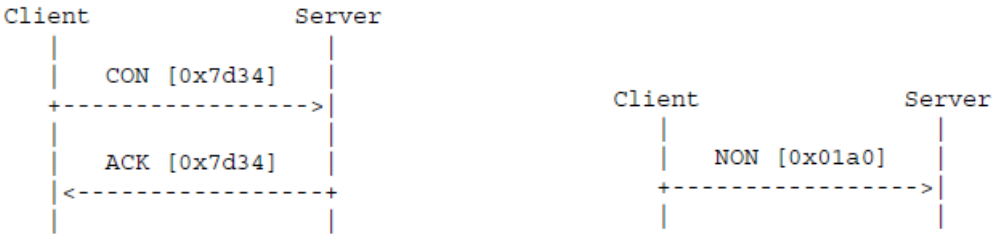
Come enunciato l'architettura di funzionamento di CoAP è del tipo *client/server*, simile a quella di HTTP. Una richiesta CoAP è equivalente ad una HTTP ed è inviata da un *client* CoAP verso un *server* CoAP per richiedere un'azione (tramite la specifica di un metodo) su una risorsa identificata da una URI. Il *server*, quindi, risponde con una risposta che può includere la risorsa. Gli scambi di messaggi sono *asincroni* e vengono scambiati tramite il protocollo di livello di trasporto UDP (Figura 2.1) .

In CoAP si definiscono 4 tipologie di messaggi:

- *Confirmable*: per avere maggiore affidabilità si può contrassegnare un messaggio come *confirmable*; in questo caso il messaggio richiede una risposta (*Acknowledgement*). Supponendo di non avere perdite di pacchetti ad ogni messaggio di tipo *confirmable* seguirà un *acknowledgement*.

- *Non-confirmable*: i messaggi contrassegnati come *non-confirmable* non richiedono l'invio di un *acknowledgement*; questa tipologia di messaggio è utile per l'invio di pacchetti inviati regolarmente, come quelli che contengono letture dai sensori.

- *Acknowledgement*: sono le risposte ai messaggi *confirmable*. Supponendo che una richiesta sia di tipo *confirmable*, l'*acknowledgment* può contenere la risposta (la risorsa) alla richiesta; in questo caso si parla di *piggybacked response*. Nel caso la risposta non



(a) Messaggio di tipo *confirmable*.

(b) Messaggio di tipo *non-confirmable*.

Figura 2.2: Tipologia di messaggi CoAP.

sia immediatamente disponibile essa verrà inviata successivamente in un messaggio separato (*separate response*).

- *Reset*: un messaggio di *reset* viene inviato a seguito di una richiesta che il server non è in grado di processare in quanto mancante del contesto.

Per avere una certa affidabilità, come detto, si può inviare una richiesta di tipo *confirmable* alla quale seguirà un *acknowledgement*; se dopo un certo timeout l'*acknowledgement* non viene ricevuto, si procede ad inviare nuovamente la richiesta. Per associare gli *acknowledgment* alle richieste entrambi i messaggi contengo un identificativo. La Figura 2.2 mostra la differenza fra un messaggio *confirmable* ed uno *non-confirmable*.

Quando un *client* effettua una *request* possono verificarsi tre casi:

- Il *client* effettua una richiesta di tipo *confirmable* ed il server avendo la risorsa a disposizione risponde con un ACK contenente la risorsa. (Fig. 2.3a)
- Il *client* effettua una richiesta di tipo *confirmable* ma il *server*, non avendo la risorsa a disposizione, risponde dapprima con un ACK per evitare che il *client*, scaduto il *timeout*, invii nuovamente la richiesta, quindi, una volta ottenuta la risorsa, risponde con un messaggio *confirmable* cui seguirà un ACK. (Fig. 2.3b)
- Il *client* effettua una richiesta di tipo *non-confirmable* ed il *server* risponde di norma con un messaggio *non-confirmable*, il protocollo prevede che sia possibile rispondere anche con un messaggio di tipo *confirmable*. (Fig. 2.3c)

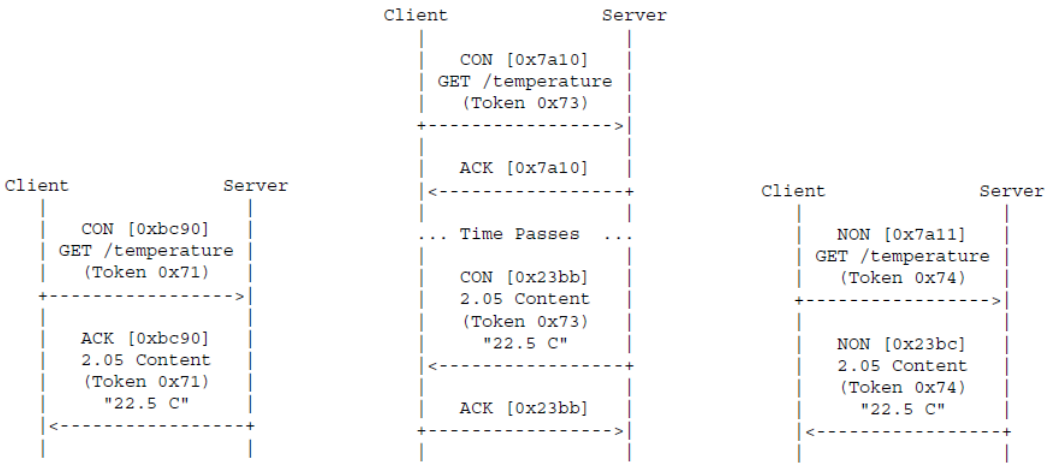


Figura 2.3: Richieste e risposte.

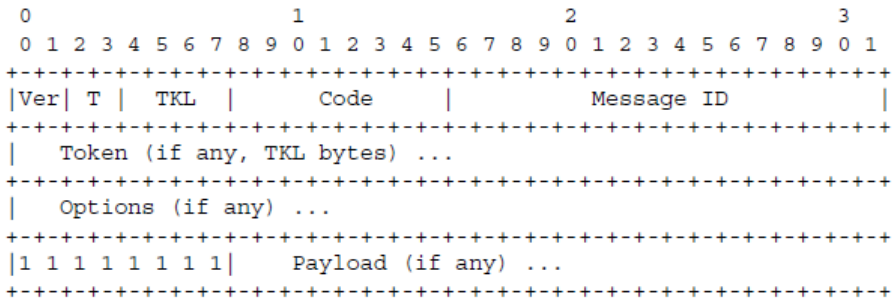


Figura 2.4: Struttura di un pacchetto CoAP.

2.3 Formato pacchetti CoAP

Caratteristica dei pacchetti CoAP è la compattezza dei messaggi. La struttura di un pacchetto CoAP è mostrata in Figura 2.4. Esso è composto da un *header* di soli 4 byte, che può eventualmente essere seguito da un *token*, da delle opzioni e da un *payload*.

L'*header* è composto dai seguenti campi:

- `Version` (`Ver`): 2 bit; indica la versione di CoAP utilizzata. Il valore attuale è `01` ovvero `Versione 1`.
- `Type` (`T`): 2 bit; indica la tipologia di messaggio: `0 = confirmable, 1 = non-confirmable, 2 = acknowledgement, 3 = reset`.

- `Token Length` (`TKL`): 4 bit; indica la lunghezza in byte del campo *token* che può assumere valori da 0 ad 8 byte. Le lunghezze da 9 a 15 sono al momento riservate.

- `Code`: 8 bit; i primi 3 bit indicano la classe (`c`) i seguenti 5 il dettaglio (`d`) a formare un codice di tipo `c.dd`. La classe c sta ad indicare una richiesta (`0`), un risposta (`2`), un errore del client(`4`), un errore del server(`5`). Il codice `0.00` indica un messaggio vuoto.

- `Message ID`: 16 bit; è utilizzato per identificare i messaggi, quindi per riconoscere i duplicati ed associare messaggi di tipo *ACK/Reset* ai messaggi di tipo *Confirmable/Non-confirmable*.

L'*header* può essere seguito dal *Token* la cui lunghezza è specificata nel campo `TKL` e viene utilizzato per associare le richieste con le risposte. Dopo il *Token* può esservi il campo opzioni che può contenere più di una opzione; segue quindi il *payload* che se presente è preceduto dal cosiddetto `Payload Marker` che contiene il valore `0xFF` indicante la fine delle opzioni e l'inizio del *payload*. Il *payload* termina al termine del pacchetto UDP pertanto la sua dimensione è calcolata basandosi sulla lunghezza del pacchetto UDP.

2.3.1 Le opzioni in CoAP

In CoAP sono definite una serie di opzioni (identificate da un *option number*) che possono essere utilizzate nei messaggi; per ogni opzione viene specificato:

- `Option Delta`: campo utilizzato per calcolare l'*option number* dell'opzione che si sta specificando. Nel campo va specificato un valore `delta` il quale, sommato al valore di *option number* dell'opzione precedente, fornisce il valore dell'*option number* corrente. Il valore dell'*option number* precedente alla prima opzione si assume sia pari a zero. Il primo *option delta* specificato è quindi l'effettivo numero dell'opzione; è possibile inoltre includere più istanze della stessa opzione specificando un *option delta* pari a zero. Con questo sistema, quindi, le varie opzioni sono elencante in ordine crescente di *option number*. Il campo ha lunghezza pari a 4 bit; valori compresi fa 0 e 12 indicano il *delta*, 13 e 14 indicano che il *delta* è specificato nel successivo campo lungo rispettivamente 8 e 16 bit.

- `Option Value Length`: questo campo di lunghezza 4 bit indica la lunghezza del campo *option value*; i valori fra 0 e 12 indicano la lunghezza in byte dell'*option value*; i valori 13 e 14 indicano che la lunghezza dell'*option value* è specifica subito prima del campo *option value* in un campo di lunghezza 8 o 16 bit rispettivamente. Per ogni opzione è previsto un range di lunghezze ammissibile.

- `Option Value`: campo che contiene il valore dell'opzione, il cui formato dipende dal tipo di opzione, così come la sua lunghezza che peraltro è specificata nel campo *option value length*.

Per il campo *option value* esistono 4 tipi di formato:

- `empty`: una sequenza di zero byte;

- `opaque`: una sequenza «opaca» di byte;

- `uint`: un intero non negativo in formato *big-endian* di lunghezza specificata nel campo *Option Value Length*;

- `string`: una stringa *unicode* codificata con UTF-8.

La struttura della sezione opzioni di un pacchetto CoAP è mostrata in Figura 2.5.

Le opzioni inoltre si suddividono nelle seguenti categorie:

- *Elective*: quando viene ricevuta un'opzione sconosciuta appartenente a questa classe essa viene semplicemente ignorata.

- *Critical*: quando viene ricevuta un'opzione sconosciuta appartenente a questa classe all'interno di una request *non-confirmable* viene generata una risposta `4.02 (Bad Option)`; se invece l'opzione sconosciuta è presente all'interno di una risposta, essa viene ignorata così come se è contenuta in un messaggio *non-confirmable*.

- *Safe-to-forward* o *Proxy Unsafe*: nel caso in cui il *proxy* non riconosca l'opzione specificata, a seconda di come viene classificata l'opzione, esso deve inoltrare o meno l'opzione.

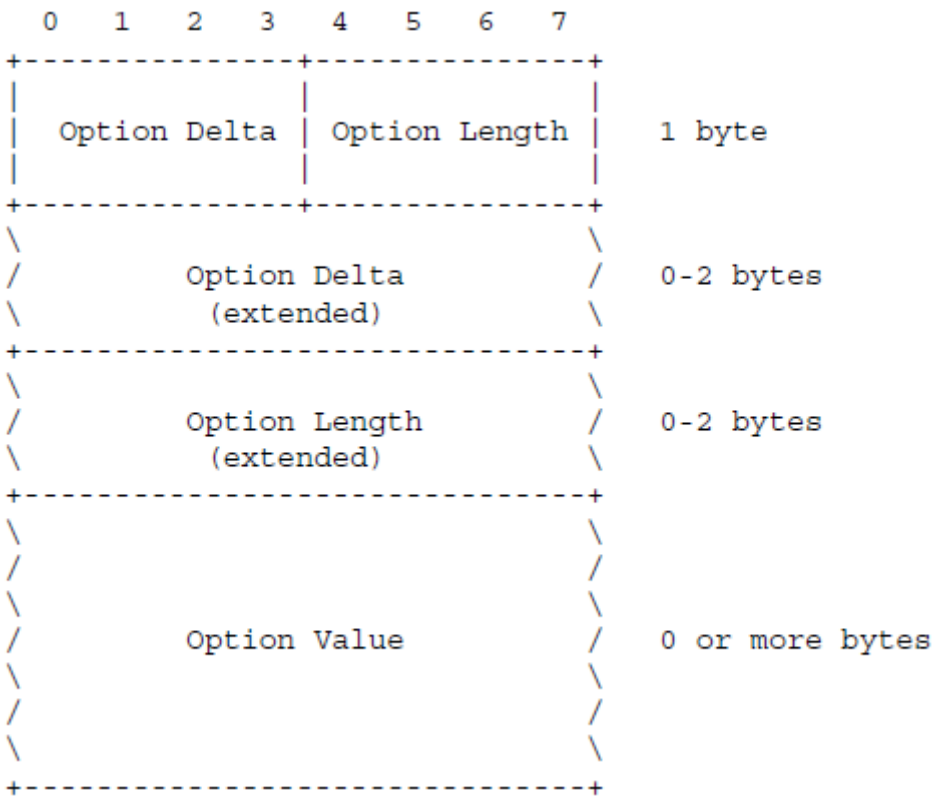


Figura 2.5: Struttura sezione Opzioni di un pacchetto CoAP.

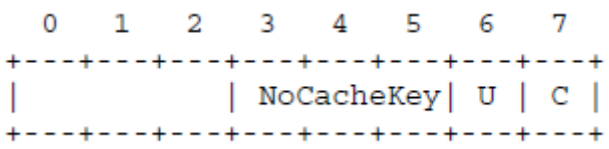


Figura 2.6: Schema mappatura numero opzioni.

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

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

Figura 2.7: Lista opzioni CoAP.

Un'opzione viene riconosciuta come `«elective»` se il suo *option number* è pari (ultimo bit uguale a zero); essa è invece di tipo `«critical»` se il suo *option number* è dispari (ultimo bit uguale ad uno). Essa è `«unsafe»` se il penultimo bit è pari ad uno, ed è una `«Cache-key»` se, oltre a non essere *unsafe*, non ha tutti i bit dal quarto al sesto pari ad 1. Lo schema di mappatura del numero delle opzioni è mostrato in Figura 2.6.

CoAP prevede inoltre dei valori di default per le varie tipologie di opzioni, pertanto non è necessario inviare un'opzione all'interno di una richiesta o risposta se si intende specificare il suo valore di default. Una lista delle opzioni previste da CoAP è mostrato nella Figura 2.7.

- *Uri-Host*, *Uri-Port*, *Uri-Path* ed *Uri-Query*: sono utilizzati per identificare la risorsa d'interesse in una richiesta; il valore di default di *Uri-host* è l'indirizzo IP del destina-

tario della richiesta, quello di *Uri-port* è la porta UDP del destinatario; *Uri-path* ed *Uri-query* compongono la URI della risorsa cui si vuole accedere.

- *Proxy-Uri* e *Proxy-Scheme*: la prima opzione viene utilizzata per effettuare una richiesta ad un *proxy* che si occupi direttamente o tramite altro *proxy* di gestire la richiesta. Se il *proxy* non è disponibile ad offrire questo servizio deve rispondere con un apposito messaggio (`5.05 Proxying not suppprted`). *Proxy scheme* specifica il metodo di ricostruzione dell'URI.

- *Content format*: indica il formato di rappresentazione del *payload* (`text/plain`; `charset=utf-8`; `application/link-format`; `application/xml`; `application/octet-stream`; `application/exi`; `application/json`).

- *Accept*: indica i tipi di *Content format* che il client può accettare.

- *Max-Age*: indica il tempo che una risposta può rimanere in *cache* prima di essere considerata obsoleta.

- *ETag*: l'*entity tag* è un `ID` che serve a distinguere diverse entità di una stessa risorsa che può variare nel tempo.

- *Location-Path* e *Location-Query*: servono per ricostruire l'URI.

- *If-Match* ed *If-None-Match*: sono chiamate *opzioni di richiesta condizionali* in quanto possono essere utilizzate dal *client* per chiedere al *server* di procedere con la risposta solo se determinate condizioni sono soddisfatte.

- *Size1*: fornisce informazioni sulla dimensione della rappresentazione della risorsa contenuta nel *payload*.

2.4 Trasmissione dei messaggi

I messaggi CoAP vengono scambiati fra *endpoint* in modo *asincrono* tramite l'utilizzo di UDP che non garantisce elevata affidabilità. Per avere garanzia di consegna dei pacchetti CoAP implementa richieste e risposte di tipo *confirmable*. Per associare le richieste alle

risposte viene utilizzato un `Message ID`, parametro che non deve essere riutilizzato per identificare un altro messaggio prima che un determinato *time-out* sia scaduto. Per quanto concerne la dimensione dei messaggi CoAP essi devono costruiti cercando di minimizzare le dimensioni necessarie; la sua lunghezza dovrebbe essere non più grande di quella di un pacchetto IP al fine di evitare frammentazioni.

2.4.1 Messaggi di tipo confirmable

Un *endpoint* che vuole inviare un messaggio di tipo *confirmable* specifica questa opzione in un apposito campo dell'*header*. Il messaggio può essere una richiesta, una risposta oppure un messaggio di *reset*. L'*endpoint* che riceve questo tipo di messaggio deve rispondere con un *acknowledgement* oppure nel caso in cui non sia in grado di soddisfare la richiesta, esso deve rispondere con un messaggio di *reset*. E' fondamentale che i messaggi di risposta ad un messaggio *confirmable* contengano il medesimo ID del messaggio originario. L'*endpoint* che invia un messaggio *confirmable* attende la conferma di ricezione del messaggio inviato e nel caso in cui questo non arrivi provvede a re-inviare il messaggio dopo un intervallo di tempo che cresce esponenzialmente con il numero di tentativi. Dopo un numero specificato di tentativi non andati a buon fine l'*endpoint* rinuncia ad inviare il messaggio. Non è necessario che l'*endpoint* esegua tutti i tentativi di ritrasmissione previsti nel caso in cui l'invio del messaggio non sia più necessario (decisione presa a livello applicazione) oppure in caso di ricezione di messaggi di errore ICMP: la richiesta può essere interrotta prima. Questo sistema di ritrasmissione con *back-off period* esponenzialmente crescente è anche utile ad evitare congestioni di rete.

2.4.2 Messaggi di tipo non-confirmable

Non sempre conviene utilizzare messaggi di tipo *confirmable*, ad esempio quando si inviano messaggi con cadenze regolari; al fine quindi di ridurre il traffico in rete è possibile affidarsi ai messaggi di tipo *non-confirmable*. Un *endpoint* che riceve un messaggio di questo tipo non deve rispondere con un *acknowledgement*, nel caso in cui però esso generi un errore può inviare una risposta. Il mittente di un messaggio *non-confirmable* non ha modo di sapere

se esso è stato correttamente ricevuto pertanto potrebbe decidere di inviare più copie di uno stesso messaggio; per questo motivo, come per la possibilità che un messaggio venga duplicato in rete, anche i messaggi non *confirmable* presentano un ID in modo tale che il o i destinatari possano processarlo una sola volta e scartare le copie.

2.5 Tipologie di messaggi

Proprio come HTTP, CoAP utilizza un sistema richiesta/risposta: un client CoAP effettua una richiesta ad un server il quale risponde. Contrariamente ad HTTP però lo scambio non avviene dopo aver stabilito una connessione fra client e server ma i messaggi vengono scambiati in modo asincrono.

2.5.1 Richieste

Una richiesta è un messaggio CoAP nel quale viene specificato un metodo (`GET`, `PUT`, `POST` o `DELETE`) che interagisca con la risorsa, oltre ad un identificativo della risorsa, un *payload* e altre informazioni riguardanti la richiesta.

2.5.2 Risposte

Dopo aver ricevuto una richiesta il *server* provvede a rispondere. Esistono tre classi di risposta identificate da uno specifico codice:

- Successo: la richiesta è stata ricevuta, interpretata correttamente ed accettata.
- Errore del *client*: la richiesta contiene un errore e quindi non può essere soddisfatta.
- Errore del *server*: la richiesta era valida ma il *server* non è in grado di soddisfarla.

Le risposte possono essere inviate con tre diverse modalità:

- *Piggybacked*: la risposta alla richiesta era subito disponibile pertanto è inserita nel messaggio di *Acknowledgment* che viene inviato poiché la richiesta era di tipo *confimable*. Il contenuto della risposta non deve essere necessariamente ti tipo «successo» ma può contenere anche un «errore».

- *Separate*: nel caso in cui la risposta ad una richiesta non sia subito disponibile oppure la richiesta è stata effettuata in modalità *non-confirmable*, essa è inviata in un messaggio separato.

- *Non-confirmable*: nel caso la richiesta sia di tipo *non-confirmable* anche la risposta lo sarà.

2.6 Metodi

Vengono qui discussi i diversi metodi supportati da CoAP. Nel caso venga inviata una richiesta con un metodo non riconosciuto oppure non supportato, il *server* deve rispondere con un messaggio `4.05 Method Not Allowed`.

I metodi supportati da CoAP sono:

- `GET`: E' il metodo da utilizzarsi per ottenere la rappresentazione della risorsa specificata dall'URI. Se è specificata l'opzione *Accept* ciò indica qual è il formato di rappresentazione che il client preferisce. Se è presente l'opzione ETAG ciò sta ad indicare che il *client* richiede la validazione dell'ETag e la rappresentazione della risorsa deve essere inviata solo se la validazione ha esito negativo. Le risposte possibili in caso di successo sono: `2.05 Content`; `2.03 Valid`.

- `POST`: Questo metodo viene utilizzato per la creazione, l'aggiornamento o la cancellazione della risorsa la cui URI è specificata nella richiesta. In caso di esito positivo le risposte possibili sono: `2.01 Created`; `2.02 Deleted`; `2.04 Changed`.

- `PUT`: Questo metodo viene utilizzato affinché la risorsa identificata dall'URI contenuto nella richiesta venga aggiornata o, in caso non sia presente, venga creata. In caso di successo le risposte possibili sono: `2.01 Created`; `2.04 Changed`.

- `DELETE`: Viene utilizzato per eliminare la risorsa identificata dall'URI presente nella richiesta. In caso di successo la risposta è `2.02 Deleted`.

2.7 Codici di risposta

Vengono qui descritti i vari codici di risposta previsti da CoAP.

2.7.1 `Success 2.xx`

- `2.01 Created`: Usata in risposta ad una richiesta `POST` o `PUT`, indica che la risorsa è stata creata. Se la risposta contiene le opzioni `location-path` o `location-query`, non è stata utilizzata l'URI specificata nella richiesta ma quella specificata nelle opzioni del presente messaggio di risposta.
- `2.02 Deleted`: Conferma, in seguito a richieste `POST` o `DELETE`, che la risorsa è stata resa non più disponibile.
- `2.03 Valid`: Indica che l'*ETag* specificato nella richiesta è valido; pertanto la risposta non contiene *payload*.
- `2.04 Changed`: In risposta a `POST` e `PUT` indica che il valore della risorsa è stato modificato.
- `2.05 Content`: In risposta ad una richiesta di tipo `GET`, il *payload* contiene la rappresentazione della risorsa.

2.7.2 `Client Error 4.xx`

- `4.00 Bad Request`: Risposta analoga alla «`400 Bad Request`» di HTTP.
- `4.01 Unauthorized`: Il *client* non ha l'autorizzazione affinché il server esegua la richiesta che gli è stata sottoposta. Prima di procedere nuovamente ad eseguire la medesima richiesta il *client* dovrebbe ottenere credenziali di livello superiore.
- `4.02 Bad Option`: Il *server* non è stato in grado di interpretare una o più opzioni specificate nella richiesta. Prima di inviare nuovamente la richiesta il *client* dovrebbe modificare la richiesta.
- `4.03 Forbidden`: Risposta analoga alla «`403 Forbidden`» di HTTP.

- `4.04 Not Found`: Risposta analoga alla «`404 Not Found`» di HTTP.
- `4.05 Method Not Allowed`: Risposta analoga alla «`405 Method Not Allowed`» di HTTP.
- `4.06 Not Acceptable`: Risposta analoga alla «`406 Not Acceptable`» di HTTP.
- `4.12 Precondition Failed`: Risposta analoga alla «`412 Precondition Failed`» di HTTP.
- `4.13 Request Entity Too Large`: Risposta analoga alla «`413 Request Entity Too Large`» di HTTP. Dovrebbe contenere un'opzione che specifica qual è la dimensione massima di *Request Entity* che il server supporta.
- `4.15 Unsupported Content-Format`: Risposta analoga alla «`415 Unsupported Media Type`» di HTTP.

2.7.3 `Server Error 5.xx`

- `5.00 Internal Server Error`: Risposta analoga alla «`500 Internal Server Error`» di HTTP.
- `5.01 Not Implemented`: Risposta analoga alla «`501 Not Implemented`» di HTTP.
- `5.02 Bad Gateway`: Risposta analoga alla «`502 Bad Gateway`» di HTTP.
- `5.03 Service Unavailable`: Risposta analoga alla «`503 Service Unavailable`» di HTTP.
- `5.04 Gateway Timeout`: Risposta analoga alla «`504 Gateway Timeout`» di HTTP.
- `5.05 Proxying not Supported`: il *server* non esegue la funzionalità di *proxy* per l'URI specificata.

2.8 URI

Per localizzare ed identificare le risorse, CoAP utilizza due diversi schemi URI denominati «`coap`» e «`coaps`», quest'ultimo da utilizzarsi con il sistema di sicurezza `DTLS`. Gli schemi

«coap» e «coaps» possono essere paragonati a quelli di HTTP: «http» ed «https». Gli schemi sono i seguenti:

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

L'host può essere fornito come indirizzo IP e quindi raggiunto direttamente, oppure come nome di dominio e pertanto l'*end-point* potrà affidarsi ad un servizio DNS per la risoluzione del nome. L'host deve sempre essere specificato, in caso sia assente la richiesta va scartata. Il parametro port fa riferimento alla porta UDP da utilizzarsi; nel caso non venga specificata è utilizzata la porta di default 5683 per lo schema «coap» e la porta 5684 per lo schema «coaps». Il parametro path identifica la risorsa all'interno dell'host specificato; è di tipo gerarchico ed i vari segmenti sono separati dal carattere «/». Il parametro query serve per caratterizzare ulteriormente la risorsa; esse sono spesso caratterizzate come coppia key=value; sono separate dal path tramite il carattere «?»; in una URI possono esservi più di una *query*, ognuna delle quali inizia con il carattere «&». CoAP supporta il prefisso di *path* «/.well-known/» che premette di scoprire informazioni sull'*host*.

2.9 Proxy

Un *proxy* è un *endpoint* CoAP ai quali i *client* possono rivolgersi per delegare l'esecuzione delle richieste. A seconda che il *proxy* venga appositamente scelto dal *client* oppure che si sostituisca al *server* di destinazione di una richiesta si parla di:

- *Fordward-proxy*: in questo caso il *client* sceglie di affidarsi ad un *proxy* e l'URI della risorsa desiderata viene specificata nell'opzione «Proxy-URI». Se il *proxy* non è disponibile ad inoltrare la richiesta deve rispondere con un apposito messaggio: «5.05 Proxying Not Supported». Nel caso in cui utilizzi una *cache* può sfruttare l'informazione presente se essa è valida, come descritto in 2.10.
- *Reverse-proxy*: In questo caso il *proxy* mette a disposizione delle risorse che non sono proprie a tutti gli effetti, ma in realtà offerte da altri *server* in rete. Il *proxy*, analizzando il pacchetto contenente la richiesta, può riscontrare che egli è a conoscenza

di dove si trovi la risorsa d'interesse (perché magari in precedenza ha effettuato una richiesta di *discovery* di servizi e risorse su vari nodi della rete) e pertanto procede per conto del *client* a gestire l'operazione sulla risorsa.

2.10 Cache

Al fine di ridurre il tempo di risposta, nonché il traffico in rete, gli *endpoint* possono effettuare la *cache* delle risposte, ovvero riutilizzare un messaggio di risposta ricevuto in precedenza per rispondere ad una nuova richiesta. Una risposta presente in *cache* può essere riutilizzata se la nuova richiesta usa lo stesso metodo nonché le medesime opzioni di quella usata per ottenere la risposta presente in *cache*; la risposta, inoltre, deve essere ancora «fresca». Una risposta è considerata «fresca» se la sua `«Max-Age»` , indicata in un apposito campo opzioni, non è stata superata. Qualora una risposta in *cache* sia scaduta l'*endpoint* può contattare il *server* con una `GET` specificando l'ETag della risposta presente in *cache* al fine di farla validare se essa è in effetti ancora valida.

2.11 Discovery

CoAP offre anche un servizio di *discovery* di servizi e risorse: un *client* potrebbe non conoscere quali sono i nodi in rete sui quali è in esecuzione un *server* CoAP e men che meno quali sono le risorse che esso mette a disposizione. Un *client* quindi può utilizzare l'indirizzo *multicast* `«All CoAP Nodes»` per scoprire i *server*. Un *server* che offre il servizio di *discovery* deve quindi supportare una connessione sulla porta CoAP di *default*. Nel caso in cui i sensori utilizzino una rete 6LoWPAN è auspicabile che supportino anche una porta situata nel *range* `61616-61631` per sfruttare al meglio la compressione dell'*header*. Il servizio di *discovery* di risorse riveste una grande importanza specialmente negli ambiti di applicazione `machine2machine` dove non vi è interazione umana fra le varie comunicazioni.

2.12 Multicast

CoAP supporta l'invio di richieste ad indirizzi appartenenti ad un gruppo IP *multicast*. Un *server* che vuole offrire servizi da richieste *multicast* deve quindi entrare a far parte di un indirizzo *multicast* `«All CoAP nodes»`; la IANA ha assegnato a tal scopo l'indirizzo `FF0X::FD`. Oltre a questo indirizzo è comunque possibile utilizzare il più generico `FF02::1` ossia l'indirizzo IPv6 `«All nodes link-local multicast»`. Le richieste inviate ad un indirizzo *multicast* devono essere di tipo *non-confirmable*. Quando un *server* riceve una richiesta *multicast* non è tenuto a rispondere: egli può ignorare la richiesta specialmente se la risposta non conterrebbe informazioni utili (*payload* vuoto oppure messaggio di errore). Quando invece i *server* che hanno ricevuto una richiesta *multicast* intendono rispondere, essi non devono inviare una risposta immediatamente dopo aver ricevuto la richiesta, al fine non sovraccaricare la rete; la trasmissione del messaggio di risposta deve invece essere ritardata di un tempo casuale compreso in un intervallo di tempo appositamente calcolato sfruttando dei parametri quali il numero dei nodi nella rete, il *bitrate* di trasmissione e la dimensione dei dati da trasferire. Il messaggio di risposta deve contenere il medesimo *token* della richiesta affinché i due messaggi possano essere associati.

2.13 Sicurezza in CoAP

CoAP offre quattro diverse modalità di funzionamento per quanto riguarda l'aspetto sicurezza. Durante una fase cosiddetta di *provisioning* ai vari nodi vengono fornite le eventuali chiavi e liste di accesso. Le quattro modalità sono:

- `NoSec`: in questa modalità nessun protocollo di sicurezza a livello CoAP viene implementato; possono eventualmente essere presenti meccanismi di sicurezza a livello più basso come descritto in 1.4.10.

- `PreSharedKey`: è attivata la funzionalità `DTLS`. Ogni nodo ha una lista di *pre-shared key* ad ognuna delle quali è associata una lista di nodi con cui una determinata chiave può essere usata. Il caso limite prevede l'utilizzo di una chiave dedicata per ogni nodo.

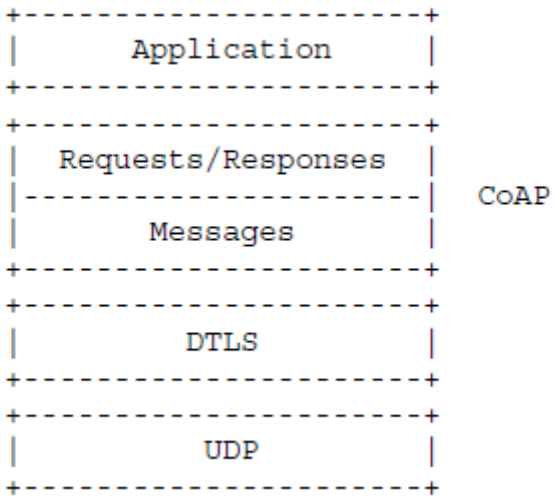


Figura 2.8: CoAP DTLS-Secured.

- `RawPublicKey`: è attivata la funzionalità `DTLS`. Ogni nodo ha una coppia di chiavi asimmetriche senza certificato. Il dispositivo inoltre è riconosciuto grazie alla sua identità calcolata tramite la chiave pubblica e come per il caso precedente è in possesso di una lista di nodi con cui comunicare.

- `Certificate`: è attivata la funzionalità `DTLS`. Ogni nodo ha una coppia di chiavi asimmetriche ed un certificato `X.509`.

La modalità `NoSec` utilizza lo schema URI di tipo `«coap»`, mentre le altre tre modalità lo schema URI `«coaps»`. Nella Figura 2.8 sono mostrati i vari livelli di rete utilizzando CoAP in modalità *DTLS-Secured*.

Il sistema è analogo al sistema di sicurezza HTTP che utilizza `TLS`. Non tutte le modalità di DTLS sono sempre supportate a causa delle limitate risorse *hardware* dei nodi. Lo scambio di messaggi avviene nel seguente modo: il *client* CoAP è anche *client* DTLS pertanto prima di inviare una richiesta deve contattare il *server* di interesse inizializzando una sessione e, solo dopo l'avvenuto *handshake* DTLS, può inviare la richiesta.

2.14 Proxy CoAP/HTTP

Grazie al supporto da parte di CoAP di un sottoinsieme di funzionalità HTTP, effettuare il *proxying* da e verso HTTP non è un'operazione troppo complessa. Il *proxying* può avvenire:

CoAP-HTTP: in questo caso i *client* CoAP vogliono accedere ad una risorsa presente su *server* HTTP. La richiesta deve contenere nelle opzioni *Proxy-Uri* o *Proxy-Scheme URI* di tipo `«http»` oppure `«https»`. Il *proxy* che riceve una tale richiesta deve occuparsi di contattare il *server* HTTP ed eseguire le operazioni sulla risorsa specificata, quindi restituire il risultato al *client*.

HTTP-CoAP: un *client* HTTP vuole accedere ad una risorsa presente su un *server* CoAP tramite un intermediario. In questo caso la richiesta HTTP conterrà una URI di tipo `«coap»` oppure `«coaps»`. Analogamente al caso precedente il *proxy* che riceve una richiesta si occuperà di inoltrarla al *server* CoAP ed una volta ricevuta la risposta inoltrerà anch'essa al *client* HTTP originario.

Capitolo 3

Progetto di un proxy CoAP avanzato

Questo capitolo tratta lo sviluppo di un *proxy avanzato* per CoAP, oggetto di questa tesi. Vengono dapprima analizzate le limitazioni dei *proxy standard* definiti da CoAP, quindi una serie di modifiche per superare tali limitazioni e ottenere prestazioni migliori in termini di tempo di servizio e consumo energetico.

3.1 Un proxy CoAP avanzato

Le caratteristiche CoAP analizzate nel Capitolo 2 sono basate sull'RFC 7252 rilasciato in Giugno 2014. Le attuali implementazioni software di CoAP sono pertanto basate su release più vecchie; quella disponibile su TinyOS è stata da poco aggiornata dal draft-08 (rilasciato nel Novembre 2011) al draft-13 (rilasciato nel Dicembre 2012). Le funzionalità attualmente offerte sulla piattaforma TinyOS non comprendono quindi tutte quelle descritte nell'apposito capitolo. Fra le funzionalità ancora non implementate vi è il *proxy*.

Scopo di questa tesi è lo sviluppo di un *proxy* CoAP che sia dotato di funzionalità aggiuntive rispetto a quanto previsto dallo standard, al fine di superare alcune limitazioni

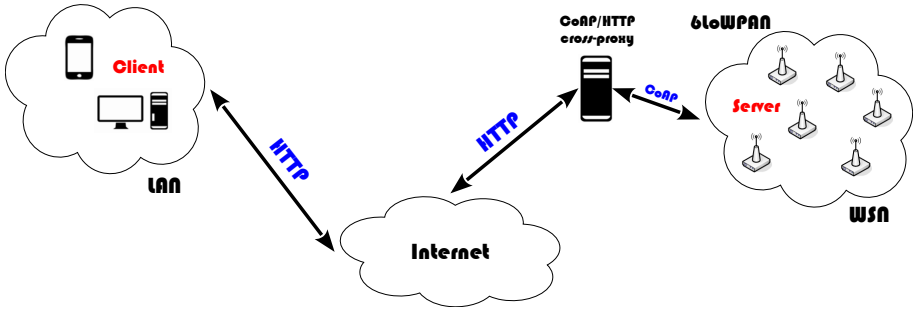


Figura 3.1: Architettura di comunicazione generale.

che si presentano in particolari scenari applicativi ed offrire migliori prestazioni in termine di consumo energetico dei nodi e tempo necessario per il servizio delle richieste.

Lo schema di un'architettura di rete generale è mostrato in Figura 3.1. Alcuni dispositivi *client* situati all'interno di una rete LAN intendono comunicare con dispositivi *server* CoAP situati in una rete WSN localizzata altrove. La due reti sono connesse tramite internet. I *client* dunque inviano delle richieste tramite protocollo HTTP; esse viaggiano nella rete internet, quindi giungono ad un nodo *cross-proxy* HTTP-CoAP che si occupa di tradurre il protocollo HTTP in CoAP; le richieste vengono infine inoltrate ai *server* CoAP presenti nella rete WSN. Essi, se previsto, risponderanno alle richieste ed i messaggi di risposta seguiranno il percorso inverso. Naturalmente la presenza della rete internet e del «cross-proxy» nel percorso seguito dai pacchetti influisce negativamente sul ritardo di consegna dei pacchetti e sulla sua variabilità. Per valutare quindi le effettive prestazioni del *proxy avanzato*, oggetto di sviluppo di questa tesi, la topologia utilizzata è quella mostrata in Figura 3.2.

Lo standard CoAP prevede già l'utilizzo di *server proxy*, sia di tipo *forward-proxy* che *reverse-proxy* come descritto in 2.9. Essi si occupano di inoltrare le richieste che vengono sottoposte dai *client* e quindi fornire le eventuali risposte. Nel caso di richieste `GET` multiple il *proxy* provvederà ad inoltrare le singole richieste, quindi le singole risposte. Anche considerando l'utilizzo di un *reverse-proxy* esso è in grado al più di fornire direttamente le risorse offerte dai nodi in rete ma conservando la modalità di rappresentazione originale. Questo meccanismo di funzionamento non sempre è il migliore: in alcuni scenari applicativi infatti si potrebbero preferire dati *elaborati* ai dati *grezzi*.

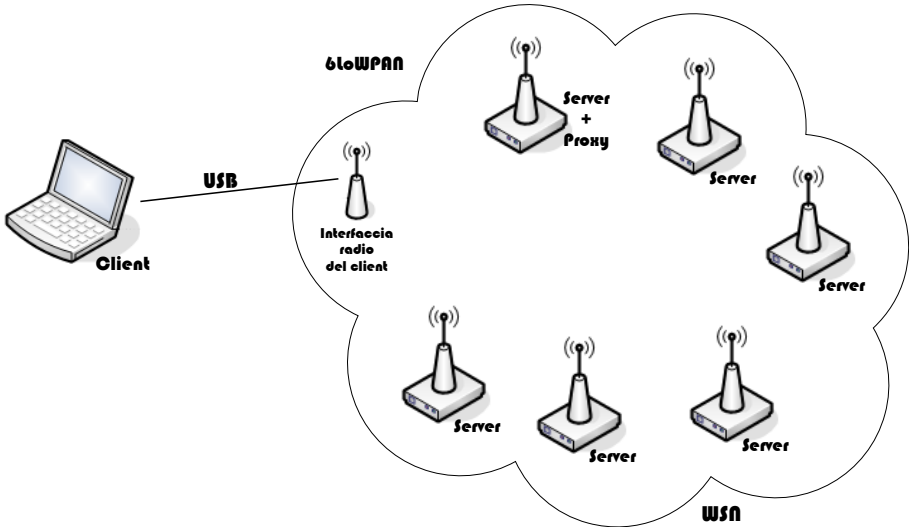


Figura 3.2: Dettaglio topologia di rete.

Si prenda ad esempio in considerazione una rete di sensori composta da:

- nodi *server* che quando interrogati da un'apposita richiesta CoAP, siano in grado di fornire il valore di temperatura misurata;
- un nodo client periodicamente interessato a conoscere il valore minimo, massimo e medio fra le temperature misurate.

In una rete priva di *proxy* è il *client* stesso che deve occuparsi di inviare le singole richieste a tutti i nodi in rete, come mostrato in Figura 3.3; esso riceverà quindi da ognuno di essi una risposta contenente il valore di temperatura rilevata. Dopo aver raccolto tutte le informazioni il *client* procederà all'elaborazione dei dati per ricavare i parametri d'interesse. Con questa modalità di funzionamento tutto il carico di lavoro è incentrato sul *client*, inoltre l'intervallo di tempo che intercorre dall'invio della prima richiesta all'ottenimento dei valori cui è interessato può rivelarsi relativamente elevato.

In uno scenario applicativo come quello sopra descritto l'utilizzo di un *forward-proxy* del tipo definito dallo standard CoAP non porterebbe comunque a dei benefici, si verificherebbe anzi un peggioramento delle prestazioni! Il *client*, infatti, dovrebbe comunque elaborare le richieste CoAP singolarmente, senza alcun risparmio di energia rispetto al caso precedente,

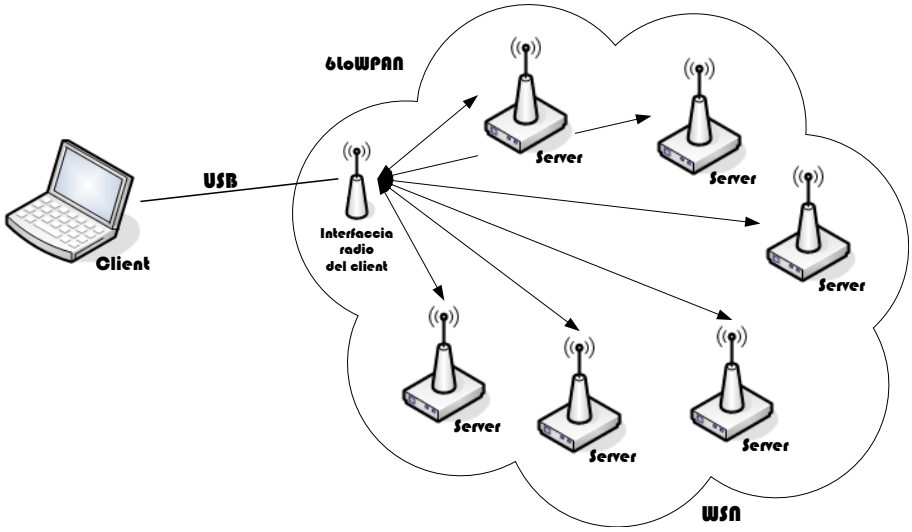


Figura 3.3: *Request* e *response* nella modilità *unicast* diretta.

mentre il tempo totale per la ricezione delle risorse aumenterebbe a causa del passaggio intermedio dovuto al *proxy*.

Per ottenere migliori prestazioni in termini di consumo energetico (lato *client*) e di tempo necessario per l'ottenimento dei dati, si rende quindi necessario apportare delle modifiche allo schema di funzionamento dei *proxy* descritto nell'RFC di CoAP. Un primo passo consiste nel modificare il metodo con il quale vengono formulate le *request* verso *proxy*. Anziché contattare il *proxy* per mezzo di una *request* contenente i dati di una risorsa relativa ad un singolo nodo, essendo la risorsa d'interesse la medesima su tutti i *server*, diviene più conveniente inviare al *proxy* un'unica *request* contenente la lista degli indirizzi di tutti i nodi che si vuole contattare o, nel caso in cui si sia interessati a ricevere la risorsa da tutti i *server* della rete, specificare un indirizzo IP *multicast*, come ad esempio l'indirizzo `ff02::1` `«link-local multicast»`.

Il *proxy*, ricevuta ed analizzata la richiesta, provvede ad inviare richieste ai nodi interessati sfruttando la funzionalità *client* CoAP, come mostrato in Figura 3.4; nell'eventualità che si vogliano contattare tutti i nodi della rete lo fa utilizzando un'unica richiesta *multicast* per risparmiare energia e ridurre i tempi (Paragrafo 5.2.3). Una volta ricevute le *response* dai server interrogati, il *proxy* può procedere ad inoltrarle al *client*.

L'utilizzo di questo sistema permette al *client* di risparmiare energia in trasmissione in

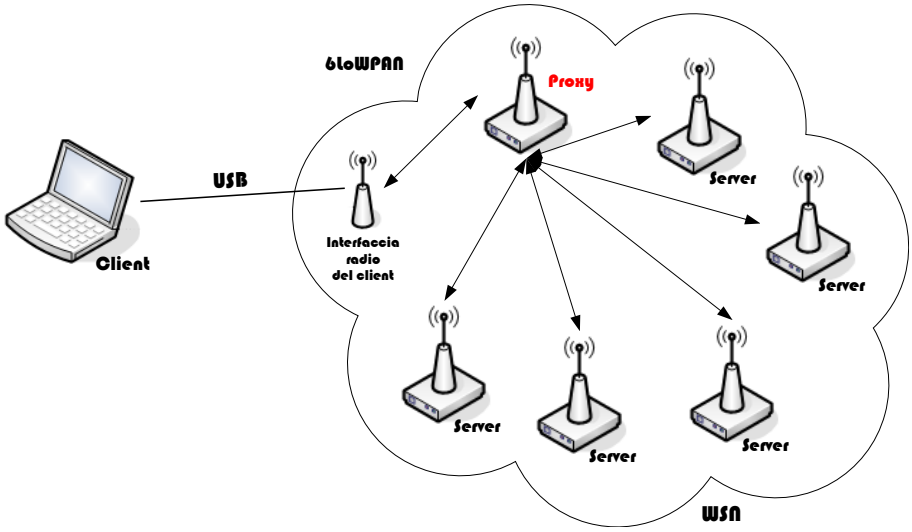


Figura 3.4: *Request* e *response* utilizzando un *proxy* avanzato.

quanto esso deve effettuare una sola *request* (verso il *proxy*); l'energia consumata in ricezione tuttavia non cambia in quanto il *client* riceverà le risposte dei singoli server (inoltrate dal *proxy*); esso dovrà poi procedere all'elaborazione dei dati grezzi con ulteriore consumo energetico.

Per attuare un ulteriore risparmio di energia il passo successivo è quello di spostare la fase di elaborazione dei dati dal nodo *client* al nodo *proxy*. Così facendo non solo il *client* vede ridursi il suo consumo di energia poiché non deve più effettuare l'elaborazione dei dati, ma anche il consumo dovuto alla ricezione radio diminuirà in quanto esso riceverà, nel caso preso in esame, un solo messaggio di *response* contenente solamente i dati di temperatura minima, massima e media, anziché *response* multiple contenenti i valori di temperatura dei singoli nodi in rete. Il risparmio di energia è duplice e sarà quindi tanto più grande quanti più sono i sensori in rete da interrogare.

Le modifiche finora apportate procurano un beneficio sotto l'aspetto del consumo energetico (come verrà dettagliatamente mostrato nel Capitolo 5) ma non apportano alcun miglioramento sotto l'aspetto del tempo necessario per l'ottenimento delle risorse d'interesse. Volendo migliorare quest'ultimo parametro, e tenendo presente che nello scenario considerato il *client* è periodicamente interessato alle risorse, è possibile dotare il *proxy* di *cache*. Esso provvederà quindi periodicamente a contattare i *server* CoAP presenti in rete ottenen-

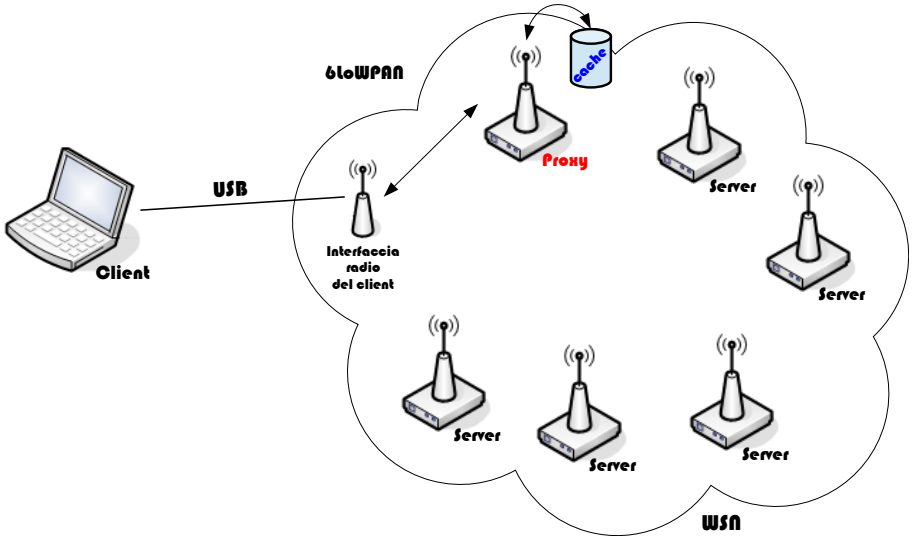


Figura 3.5: *Request* e *response* utilizzando un *proxy* avanzato dotato di *cache*.

do il dato di risorsa aggiornato. Dopo aver ricevuto tutti i dati necessari elaborerà i dati grezzi per ottenere i dati d'interesse per il *client*, i quali verranno memorizzati nella *cache* in attesa della *request*. Giunto il momento in cui il *client* effettua la richiesta, il *proxy* potrà immediatamente provvedere ad inviare una *response* CoAP prelevando i dati necessari dalla *cache*, come mostrato in Figura 3.5.

Il tempo necessario a servire detta richiesta diviene quindi equivalente a quello necessario per fornire una risorsa locale, sebbene i dati forniti siano frutto di informazioni provenienti da tutta la rete.

L'utilizzo di questo sistema richiede però l'utilizzo di alcune accortezze: è infatti necessario impostare correttamente la frequenza di aggiornamento della *cache* dimensionandola in base alle necessità del client: una frequenza troppo elevata rispetto alle effettive richieste provocherebbe infatti un inutile consumo energetico nel nodo *proxy* e nei vari nodi *server*; una frequenza troppo bassa d'altra parte pur offrendo un basso consumo energetico non consentirebbe al *client* di ottenere informazioni aggiornate. Può essere il *client* stesso a fornire la frequenza con cui intende richiedere le risorse, per mezzo di un apposita opzione contenuta nella *request*. Oltre alla frequenza con la quale il *proxy* deve eseguire l'aggiornamento, un altro parametro da considerare è l'istante in cui il *proxy* deve eseguire l'invio delle *request*

CoAP ai vari *server* della rete. E' preferibile ritardare il più possibile questo istante in modo tale che i dati siano il più «freschi» possibile quando giungerà la richiesta dal client CoAP, ma è fondamentale che le richieste siano inviate con sufficiente anticipo in modo tale che, nel momento in cui giungerà la *request* da parte del client, il *proxy* abbia ricevuto tutte le *response* ed abbia avuto il tempo per elaborare i dati. La scelta di questo istante di tempo dipende da diversi fattori, alcuni dovuti alla rete altri alle caratteristiche del *proxy*. Fra i fattori influenzati dalla rete vi sono: il numero di nodi da interrogare, la modalità con cui essi verranno interrogati (*unicast* oppure *multicast*), la topologia della rete (la presenza di numerosi percorsi *multi-hop* allunga i tempi per la ricezione di una risposta) e non ultimi il traffico presente in rete e la rumorosità del canale (un canale molto occupato o rumoroso provoca ritardi esponenziali a causa del sistema di *back-off* utilizzato). Fra i parametri dipendenti dal *proxy* si possono citare: carico di lavoro, velocità computazionale, complessità dell'elaborazione. Questi parametri inoltre, a causa delle caratteristiche intrinseche delle reti WSN sono variabili nel tempo con la conseguenza che anche il tempo necessario a servire le richieste subirà variazioni. Nel *proxy* è stato quindi implementato un meccanismo che tiene traccia dell'istante di tempo in cui esso invia la prima richiesta e del momento in cui riceve l'ultima *response* dai server interrogati; questi dati sono utilizzati per calcolare il tempo impiegato affinché le richieste vengano servite; basandosi su tale parametro può quindi impostare l'anticipo per le successive *request*.

Oltre a benefici in termini di consumo energetico e ritardo, l'utilizzo di un *proxy* avanzato garantisce maggiore flessibilità alla rete. Le possibili applicazioni infatti possono essere molteplici e la sola riprogrammazione del nodo *proxy* può aumentare le funzionalità offerte. I nodi server infatti, limitandosi ad offrire dati grezzi provenienti da misurazioni hardware non necessiteranno di riprogrammazioni del firmware al variare dell'applicazione.

3.2 Sviluppo del proxy avanzato

Il *proxy-avanzato* è quindi un nodo che svolge contemporaneamente le funzionalità di *client* e *server* oltre ad operazioni di elaborazione. Per prima cosa si è deciso di impostare il nodo *proxy* come *Root* del DODAG, di fargli svolgere cioè la funzione di nodo radice dell'*albero*

di routing. In questo modo i vari nodi della rete contatteranno il *proxy* per la formazione del DODAG ed esso avrà nella propria tabella di *routing* la lista di tutti i nodi presenti in rete. Ciò è ottenuto grazie all'utilizzo di apposite interfacce:

```
uses {
[...]
interface RootControl;
interface ForwardingTable;
[...]
}
```

Nello specifico l'interfaccia `RootControl` consente di controllare se il nodo è *root* dell'albero, nonché di impostarlo come tale o rimuoverlo da questo ruolo. L'interfaccia `ForwardingTable` invece permette di accedere alla *tabella di routing*. Il nodo designato come *proxy* pertanto, subito dopo aver effettuato il *boot* e l'attivazione dell'interfaccia radio, provvederà ad identificarsi in rete come *DODAG Root*:

```
event void Boot.booted() {

    [...]
    call RootControl.setRoot();
    call RadioControl.start();
    [...]

}
```

L'effetto di tale configurazione è mostrato di seguito:

```
destination gateway interface
fec0::2/128 fe80::212:6d45:506e:9c37 1
fec0::3/128 fe80::212:6d45:506e:f5e8 1
```

Esso rappresenta la tabella di *routing* del nodo *proxy* avente indirizzo `fec0::1`; sono presenti due *entry* rappresentanti due nodi *server* presenti in rete ed aventi indirizzo `fec0::2` e `fec0::3`. Il parametro *gateway* rappresenta il loro stesso indirizzo in quanto raggiungibili direttamente. Organizzando invece i sensori in modo tale che il nodo con indirizzo `fec0::3` sia posizionato al di fuori del raggio di copertura del *proxy*, ma possa essere raggiunto tramite

il server `fec0::2`, ciò che si presenta interrogando la tabella di *routing* del *proxy* è quanto segue:

```
destination gateway interface

fec0::2/128 fe80::212:6d45:506e:9c37 1

fec0::3/128 fe80::212:6d45:506e:9c37 1
```

Come si vede il *gateway* di riferimento per entrambe le *entry* è l'indirizzo del server `fec0::2` che in questo caso funge da *hop* intermedio. Ciò è possibile grazie allo *stack* protocollare utilizzato, ovvero BLIP, per mezzo del quale ogni nodo di rete è anche un *router*.

Nel caso in cui un *client* sia interessato a conoscere quali sono i *server* presenti in rete può contattare il *proxy*, che è anche un *server* CoAP, e richiedere la sua *routing table* che è messa a disposizione tramite un'apposita risorsa (URI: `«/rt»`, si veda la Tabella 4.4).

Una volta venuto a conoscenza dei *server* disponibili il client può fornire al *proxy* la lista dei *server* che desidera vengano interrogati. Per far ciò è stata implementata un'apposita risorsa presso il *proxy* che permette di fornirgli, oltre alle informazioni sui *server*, anche l'informazione su quale sia l'URI della risorsa di interesse. Il *client*, dunque, invia una *request* di tipo `PUT` specificando tali parametri. La descrizione della risorsa è mostrata di seguito:

```
#ifdef COAP_RESOURCE_SERVLIST

    { KEY_SERVLIST, "sl", sizeof("sl"),

        COAP_MEDIATYPE_APPLICATION_OCTET_STREAM, 1, 1, 0},

#endif
```

La risorsa, che si è deciso di denominare `«Server List»`, ha URI `«/sl»`; la rappresentazione `MIME` del *payload* è di tipo `«octet-stream»`, è di tipo `«writable»` (la richiesta effettuata dal client su questa risorsa infatti è di tipo `PUT`) e la gestione della risorsa è di tipo `«split-phase»`. Il *payload* relativo a questa risorsa è mostrato in Figura 3.6.

Il significato dei singoli campi è spiegato di seguito:

- `URI Risorsa`: in questo campo è specificata l'URI relativa alla risorsa della quale il *client* è interessato a ricevere la rappresentazione, dopo opportuna elaborazione.

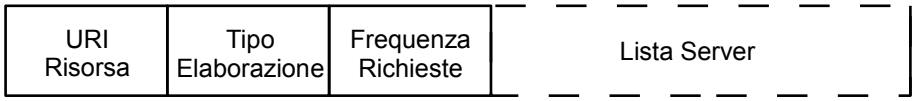


Figura 3.6: *Payload* relativo alla risorsa `«Server List»`.

- `Tipo Elaborazione`: qui viene specificato il tipo di elaborazione che il *proxy* deve eseguire sulle risorse aventi URI `«URI Risorsa»` recuperate dai server. Alcuni esempi di elaborazione possono riguardare il calcolo del valor medio, minimo e massimo.
- `Frequenza Richieste`: indica la frequenza con la quale il *client* è interessato a ricevere la risorsa elaborata. Questo dato è sfruttato dal *proxy* per minimizzare il consumo energetico e contemporaneamente offrire dati «freschi».
- `Lista Server`: è un campo di lunghezza variabile e contiene la lista degli indirizzi dei Server CoAP che il *proxy* deve contattare per ottenere le risorsa da elaborare successivamente.

Ricevuta la richiesta il *proxy* può mobilitarsi per recuperare la risorsa desiderata dal *client* presso i *server* specificati. Naturalmente il *proxy*, oltre ad ottenere la rappresentazione di una risorsa per conto del *client*, può anche provvedere a modificare le risorse presenti sui vari *server*, ovviamente se queste supportano il metodo `PUT`. Fra quelle messe a disposizione sui TelosB è il caso della risorsa `«Led»`. Un esempio è mostrato di seguito:

```
struct route_entry *entry;
struct sockaddr_in6 sa6;
coap_list_t *optlist = NULL;
int n;
int cur_entry;
entry = call ForwardingTable.getTable(&n);
for (;cur_entry < n; cur_entry++) {
    if (entry[cur_entry].valid) {
        sa6.sin6_addr = entry[cur_entry].prefix;
        sa6.sin6_port = htons(COAP_CLIENT_PORT);
        coap_insert( &optlist,
        new_option_node(COAP_OPTION_URI_PATH,
        sizeof("l") - 1, "l"), order_opts);
        printf("\nInvio richiesta coap a ");
        printf_in6addr(&sa6.sin6_addr);
        printf(".\n");
        call CoAPClient.request(&sa6, COAP_REQUEST_PUT,
        optlist, 1, dato);
        t_tx[n_tx]= call Timer1.getNow();
        n_tx++;
    }
else {printf("\n%i di %i: Entry non valida",cur_entry,n);}
}
```

In questo scenario il *proxy*, accedendo alla propria tabella di *routing*, ottiene gli indirizzi IPv6 dei singoli *server* e procede a contattarli uno dopo l'altro inviando loro una richiesta di tipo `PUT` («`COAP_REQUEST_PUT`») destinata alla risorsa «`Led`» la cui URI è «`l`». Il valore con cui verrà modificata la risorsa è contenuto nella variabile «`dato`». Grazie al *timer* (`t_tx[n_tx]= call Timer1.getNow();`) inoltre l'applicazione tiene traccia dei tempi

necessari affinché le richieste vengano servite per gli scopi discussi nel Paragrafo 3.1.

Capitolo 4

Piattaforme Hardware e Software di riferimento

In questo capitolo verranno descritte le piattaforme *hardware* e *software* utilizzate per lo sviluppo ed il test del *proxy* avanzato di cui al Capitolo 3. Nello specifico la piattaforma *hardware* prescelta è la TelosB; il sistema operativo utilizzato è TinyOS; come simulatore per valutare i consumi energetici si è optato per Cooja.

4.1 Piattaforma software utilizzata: TinyOS

Uno dei requisiti fondamentali dei sensori utilizzati nelle reti WSN è il basso assorbimento energetico; questo requisito può essere soddisfatto tramite l'utilizzo di *hardware* con prestazioni limitate. Non è pertanto possibile utilizzare un sistema operativo *general purpose* in quanto le risorse richieste da tale tipologia di sistemi operativi non sono compatibili con le limitate prestazioni *hardware* tipiche di questi sensori. Si rende quindi necessario rivolgersi a sistemi operativi appositamente studiati. Negli ultimi anni sono state sviluppate numerose alternative, quelle di maggior successo sono: TinyOS, Contiki e LiteOS.

Per condurre i test oggetto di questo capitolo si è scelto di utilizzare il sistema operativo TinyOS [24]: nato nel 2000, è stato fra i primi ad essere sviluppato. Il sistema operativo ha

licenza *open source* ed è scritto nel linguaggio di programmazione nesC (network embedded systems C). nesC, che può essere considerato un dialetto del linguaggio C, è un linguaggio ad eventi, basato sui componenti ed ottimizzato per supportare piattaforme con limitate risorse *hardware* (quantità di memoria limitata e bassa potenza di calcolo della CPU).

Come riportato da [25] gli aspetti caratterizzanti di nesC sono:

- Separazione della costruzione e della composizione: i programmi sono costituiti da componenti che vengono assemblati («wiring») per formare il programma completo.
- Specifica dei componenti per mezzo di interfacce: le interfacce possono essere usate o fornite dal componente: le ultime indicano le funzionalità che il componente fornisce all'utente, le prime rappresentano le funzionalità di cui il componente necessita per svolgere il proprio lavoro.
- Interfacce bidirezionali: da un lato specificano una serie di funzioni che devono essere implementate da chi fornisce l'interfaccia (i «comandi») dall'altra una serie di funzioni che devono essere implementate dall'utilizzatore dell'interfaccia (gli «eventi»).
- I componenti sono collegati fra di loro tramite interfacce.

4.1.1 Comandi, eventi e task

Come specificato nel paragrafo precedente le interfacce mettono a disposizione dell'utente una serie di `«comandi»` ovvero funzioni che eseguono delle operazioni su dei dati in ingresso restituendo dei dati in uscita. I comandi vengono eseguiti per mezzo della primitiva `call`. Viene di seguito mostrata la sintassi per l'implementazione e la chiamata di un comando:

```
tipoConcorrenza command interfaccia.comando(lista parametri){

    [...implementazione comando...]
    return paramRestituito;

}
paramRestituito = call interfaccia.comando(listaParametri);
```

Gli «eventi» invece rappresentano dei gestori delle interruzioni *hardware* quale può essere l'arrivo di un messaggio, l'invio di un messaggio, la scadenza di un *timer*. Essi sono segnalati tramite la primitiva «signal».

La sintassi per la loro implementazione e segnalazione è la seguente:

```
tipoConcorrenza event interfaccia.evento(listaParametri){

    [...implementazione evento...]
    return paramRestituito;

}
paramRestituito = signal interfaccia.comando(listaParametri);
```

Le tipologie di concorrenza possono essere:

- Asincrona («async»): l'esecuzione di comandi od eventi asincroni causa un'interruzione dell'esecuzione delle altre operazioni, che riprenderanno una volta completata l'esecuzione del comando che ha generato il blocco.
- Sincrona («sync»): non può gestire eventi generati da *interrupt hardware*, ma può essere utilizzata solo dai *task*.

Vi sono quindi i «task», ovvero porzioni di codice eseguite senza essere bloccate da un altro comando. Quando un *task* viene chiamato esso non viene subito eseguito ma inserito in una coda di esecuzione; un apposito *scheduler* provvede ad eseguire i *task* uno dopo l'altro. Qualora la coda sia vuota il processore viene commutato in modalità *sleep* per risparmiare energia. I *task* possono effettuare `call` di comandi, segnalare eventi ed eseguire altri *task*. I *task* vengono lanciati per mezzo della primitiva «post».

Di seguito viene mostrata la sintassi per l'implementazione e la chiamata di un *task*:

```
task void nomeTask(listaParametri){

    [...implementazione...]

}
post nomeTask(listaParametri);
```

4.1.2 Interfacce e componenti

Nelle interfacce sono specificate le funzioni che devono essere fornite dal modulo che dichiara di implementarle, nonché gli eventi che devono essere gestiti da chi le utilizza. Sono caratterizzate da un codice modulare. Le interfacce sono composte da una lista di comandi ed eventi, ognuno con la descrizione dei parametri in ingresso ed in uscita.

Un componente può utilizzare (`«uses»`) o fornire (`«provides»`) un'interfaccia e può essere di due tipi:

- modulo (`«module»`): i moduli implementano i comandi dichiarati nelle interfacce che esportano e gli eventi generati dai componenti che vengono utilizzati.
- configurazione (`«configuration»`): le configurazioni invece definiscono il modo in cui più componenti devono essere collegati; possono a loro volta fornire delle interfacce.

Per realizzare un'applicazione occorre collegare i moduli tramite dei file di configurazione, il cosiddetto *wiring dei componenti*.

4.1.3 Architettura di astrazione hardware

TinyOS è basato su un sistema di astrazione dell'*hardware*, la cui architettura è mostrata in Figura 4.1, denominato *Hardware Abstraction Architecture* (HAA).

E' possibile suddividere questa architettura in tre livelli:

- *Hardware Presentation Level* (HPL): Al livello più basso troviamo l'HPL, che si interfaccia direttamente con l'*hardware*, modificando i registri e gestendo gli *interrupt*. Essendo direttamente connesso con l'*hardware* esso è specifico per la piattaforma; è pertanto necessario utilizzare i moduli appropriati per l'*hardware* che si intende utilizzare.
- *Hardware Abstraction Layer* (HAL): Esso è ancora dipendente dall'*hardware* ed il suo compito è quello di mascherare ai livelli superiori la gestione *hardware* sottostante.
- *Hardware Interface Layer* (HIL): Compito del questo livello è quello di fornire un'astrazione dell'*harware* sottostante, mettere ovvero a disposizione degli applicativi funzioni che siano indipendenti dalla piattaforma *hardware* utilizzata.

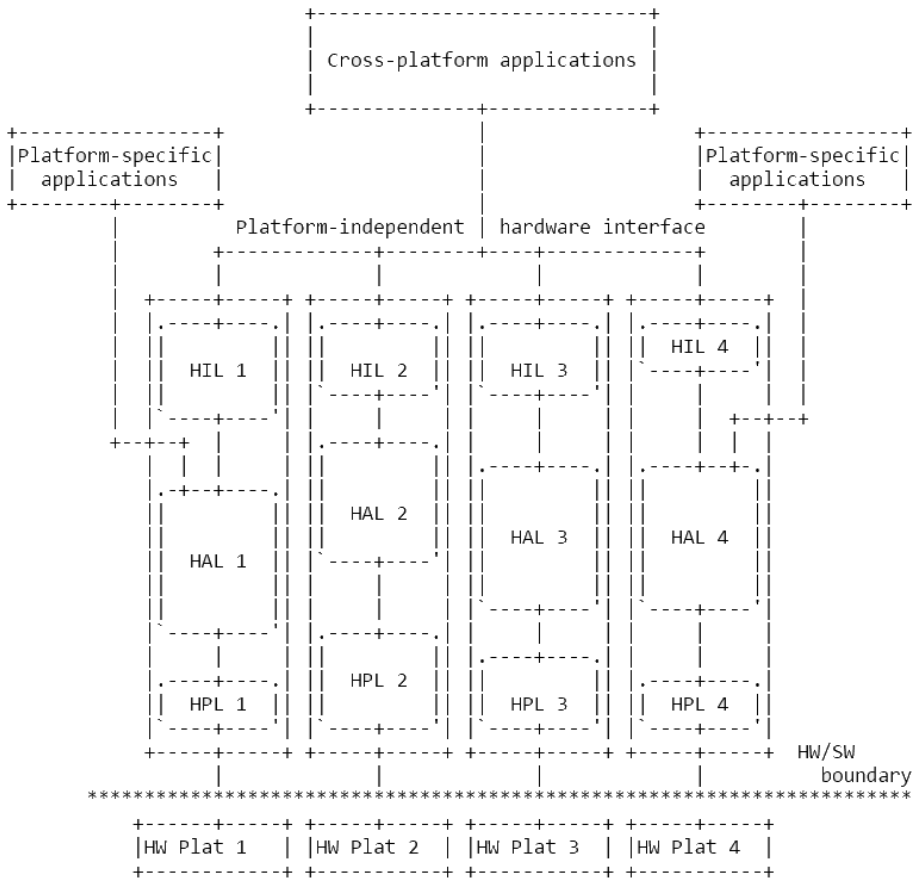


Figura 4.1: *Hardware Abstraction Architecture.*

Piattaforme	
EPIC	Mica2dot
Imote2	NXTMOTE
Shimmer	Mulle
IRIS	TMote Sky/TelosB
TelosB/Kmote	TinyNode
MicaZ	Zolertia Z1
Mica2	UCMote Mini

Tabella 4.1: Lista delle piattaforme supportate da TinyOS.

Livello	Protocollo
Applicazione	CoAP, HTTP
Trasporto	TCP, UDP
Rete (IP/Routing)	IETF RPL
Rete (Adattamento)	IETF 6LoWPan
Collegamento (MAC)	802.15.4e
Fisico	802.15.4-2006

Tabella 4.2: *Stack* protocollare utilizzato da BLIP.

4.1.4 Piattaforme hardware supportate

TinyOS supporta un vario numero di piattaforme, le principali sono indicate nella Tabella 4.1.

Per quanto riguarda lo *stack* protocollare IPv6 ci si è affidati a BLIP (*Berkeley Low Power IP stack*) che implementa i vari protocolli IP necessari al funzionamento di una rete IPv6. I protocolli utilizzati da BLIP nei vari livelli dello *stack* sono mostrati nella Tabella 4.2.

Nelle reti IP i nodi possono essere *host* oppure *router*; ogni sensore che esegue lo *stack* di rete BLIP è un *router* in grado di inoltrare pacchetti. L'utilizzo di 802.15.4 a livello di trasporto limita l'MTU (Maximum Transmission Unit) a 100 byte, ma grazie all'uso di 6LoWPAN è possibile la trasmissione di *payload* aventi dimensione fino a 1280 byte per mezzo della frammentazione eseguita a livello 2.5. Le piattaforme di sensori supportate da BLIP sono elencate in Tabella 4.3.

A livello di applicazione è stata sviluppata per TinyOS un'implementazione di CoAP basata sulle librerie *libcoap* sviluppate dall'Università di Brema. L'ultima implementazione disponibile è basata sul draft-ietf-core-coap-13. L'unica piattaforma al momento compatibile

Piattaforma	Radio
EPIC	CC2420
TelosB	CC2420
IRIS	RF230
MicaZ	CC2420
Shimmer	CC2420

Tabella 4.3: Piattaforme supportate da BLIP.

Risorsa (URI)	GET	PUT	Descrizione
/st	X		Temperatura
/sh	X		Umidità
/sv	X		Tensione
/r	X		Tutte (tensione + temperatura + umidità)
/l	X	X	LED
/rt	X		Tabella di routing

Tabella 4.4: Risorse CoAP disponibili su TelosB.

è la TelosB. Alcune delle risorse attualmente a disposizione sono mostrate nella Tabella 4.4.

4.2 Il simulatore Cooja

TinyOS è dotato di un simulatore, denominato TOSSIM [26], sviluppato ad alto livello per renderlo indipendente dalla piattaforma. Purtroppo esso non è compatibile con lo *stack* BLIP né è in grado di fornire informazioni sul consumo energetico dei sensori. Per analizzare quindi il comportamento del *proxy* avanzato sviluppato in questa tesi sì è operato direttamente sui sensori TelosB per quanto concerne la valutazione dei ritardi, si è invece scelto di utilizzare il simulatore Cooja per valutare i consumi energetici.

Cooja è il simulatore fornito assieme al sistema operativo Contiki; può lavorare su diversi livelli di protocollo e supporta molteplici piattaforme *hardware* [27, 28] fra cui i TelosB. L'ambiente di lavoro è dotato di interfaccia grafica che fornisce una rappresentazione della rete e dello scambio di pacchetti. In Figura 4.2 è mostrato uno *screenshot* raffigurante la simulazione di un client CoAP intento ad inviare una *request multicast* ai vari *server* presenti in rete.

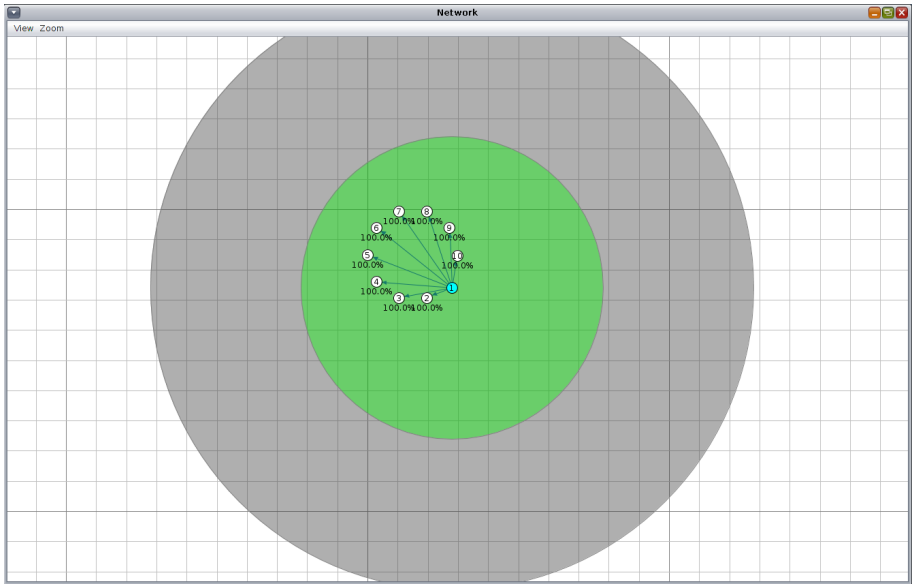


Figura 4.2: Scambio di pacchetti in Cooja.

Nello scenario preso in esame i vari nodi *server* risultano tutti all'interno del raggio di portata del nodo *client* (area verde). Cooja offre 4 modelli di propagazione radio [29]:

- *UDGM Constant Loss*: è la modalità utilizza per condurre i test oggetto di questa tesi. Con questo modello di propagazione il raggio di trasmissione dei nodi è un disco: tutti i nodi all'interno del disco ricevono perfettamente i pacchetti trasmessi, quelli al di fuori del disco non ne ricevono alcuno.

- *UDGM Distance Loss*: simile al modello precedente ma con l'aggiunta di interferenze. E' possibile specificare tramite appositi parametri la probabilità di successo della trasmissione e della ricezione dei pacchetti.

- *DGRM*: Con questo modello è possibile specificare a livello di *link* i parametri di successo nell'invio/ricezione dei pacchetti. E' possibile inoltre considerare i ritardi di propagazione.

- *MRM*: Questo modello utilizza la tecnica *ray-tracing*; viene considerata la presenza di oggetti che possono attenuare il segnale nonché provocare rifrazioni, riflessioni e diffrazioni.

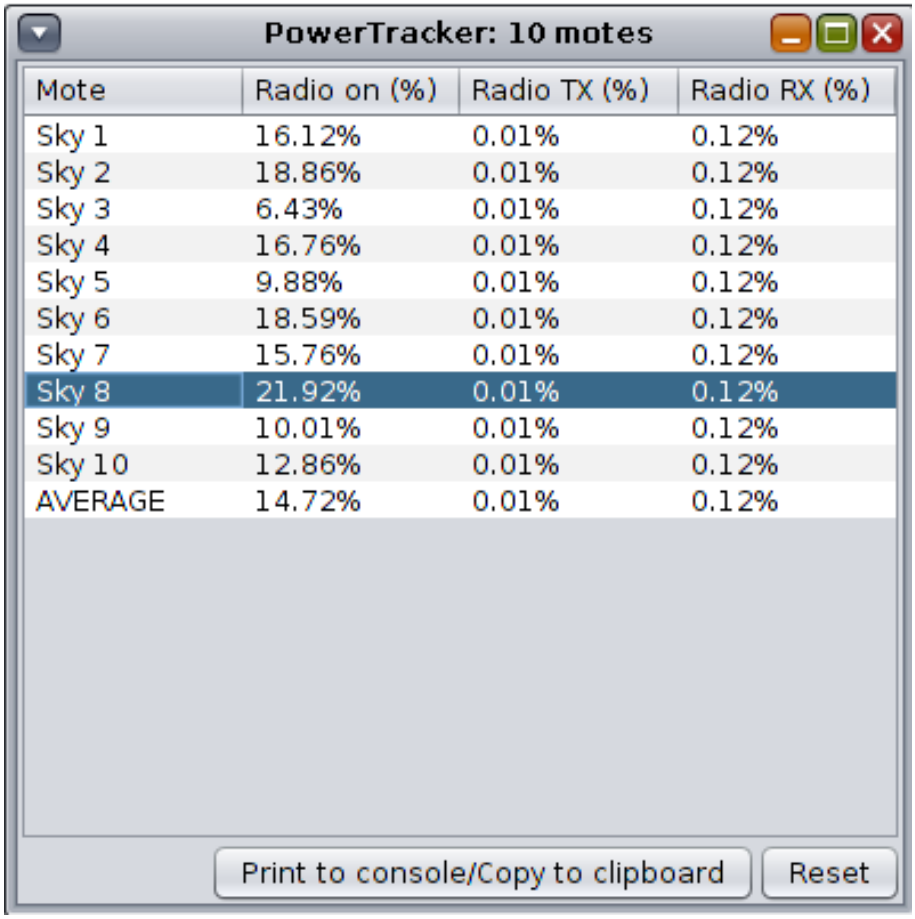


Figura 4.3: Tempo di utilizzo dell'infaccia radio dei singoli sensori.

Cooja dispone inoltre di un apposito *toolbox* che permette di valutare il tempo di utilizzo dell'interfaccia radio dei singoli nodi della rete (Figura 4.3).

Come risulta evidente dalla figura è possibile ricavare i tempi in cui le singole radio sono in stato di «acceso», «trasmissione» e «ricezione». Il tempo, oltre che in formato percentuale, è fornito in microsecondi e proprio questi valori sono stati utilizzati per valutare il consumo energetico (Capitolo 5).

4.3 Piattaforma hardware utilizzata: TelosB

Come piattaforma *hardware* si è optato, come già accennato, per la TelosB, al momento l'unica ad essere compatibile con l'implementazione CoAP di TinyOS . La piattaforma è

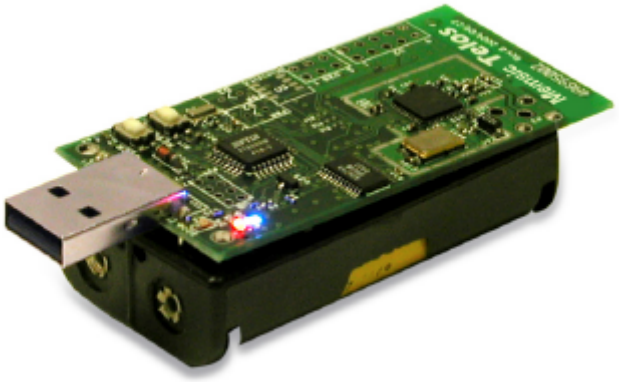


Figura 4.4: Un sensore MEMSIC TelosB TRP2424.

stata sviluppata dall'università di Berkeley ed i sensori sono prodotti dalla MEMSIC [30], un'azienda statunitense con sede nel Massachusetts. I modelli utilizzati sono i TPR2420.

Le principali caratteristiche *hardware* sono riassunte di seguito:

- Microcontrollore: il cuore della piattaforma TelosB è costituito da un microcontrollore MSP430F1611 prodotto da Texas Instruments. E' caratterizzato da un'architettura a 16 bit di tipo RISC ed una frequenza di *clock* pari ad 8 MHz. Le 5 diverse modalità di *sleep* consentono bassi assorbimenti di energia garantendo una lunga autonomia.

- Interfaccia radio: è costituita da un un circuito integrato Chipcon (ora Texas Instruments) CC2420 compatibile con lo standard 802.15.4 in grado di trasmettere ad un rate di 250 kb/s.

- Memoria: i TelosB sono dotati di 48 KB di memoria ROM e 10 KB di memoria RAM per l'esecuzione dei programmi.

- Sensori a bordo: la piattaforma può essere dotata di un sensore di luminosità Hamamatsu S1087 in grado di rilevare lunghezze d'onda nello spettro del visibile e dell'ultravioletto, e di un sensore di temperatura ed umidità prodotto da Sensirion.

- Antenna: integrata nel circuito stampato, consente una portata di 50 m in interni e di 125 m in esterni.

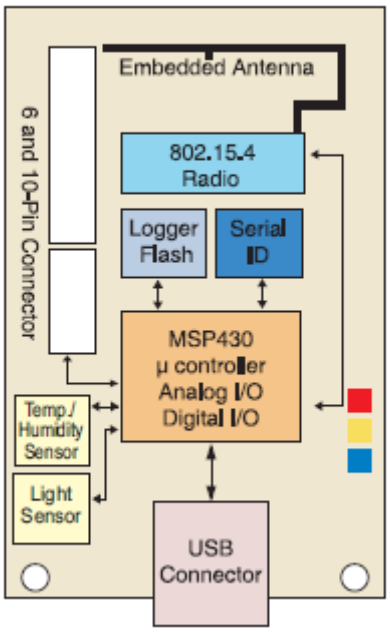


Figura 4.5: Diagramma a blocchi di un TelosB.

- Alimentazione: 5V DC tramite porta USB con la quale è possibile effettuare anche la programmazione del sensore; in alternativa possono essere utilizzate 2 batterie AA.

Il diagramma a blocchi di un TelosB è mostrato nella Figura 4.5.

Capitolo 5

Valutazione delle prestazioni

In questo capitolo vengono illustrati i risultati dei test eseguiti sfruttando le piattaforme *hardware* & *software* di cui al Capitolo 4. I risultati sono espressi in funzione di tempo necessario per servire una richiesta e consumo energetico dei vari nodi che compongono la rete. Questi parametri saranno valutati al variare della topologia e del ruolo svolto da alcuni nodi della rete (presenza o meno di un *proxy avanzato*).

5.1 Topologia della rete

La topologia della rete oggetto dei test è mostrata in Figura 5.1. Compongono la rete un Personal Computer (`nodo 0`) dotato di interfaccia 802.15.4 sul quale è in esecuzione un *client* CoAP; un nodo *root* (`nodo 1`) sul quale è in esecuzione sia un *client* che un *server* CoAP (è il nodo che può svolgere la funzione di *proxy-avanzato*) ed alcuni nodi foglia (`nodi 2-10`) sui quali è in esecuzione un *server* CoAP. Ogni nodo della rete si trova all'interno del raggio di trasmissione degli altri nodi sicché la comunicazione diretta fra i nodi è possibile. I nodi *server* mettono a disposizione dei nodi *client* una serie di risorse, come i valori di temperatura, umidità e luminosità misurati. A seconda delle applicazioni e delle piattaforme utilizzate naturalmente la tipologia di risorse offerte può variare.

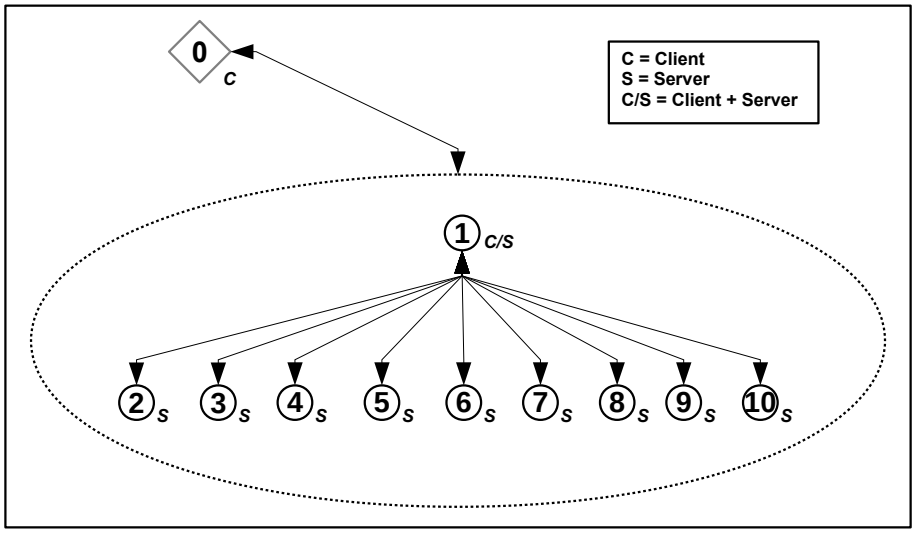


Figura 5.1: Topologia della rete.

5.2 Scenari applicativi

Si supponga che il «nodo 0» sia interessato ad accedere alle risorse messe a disposizione dai vari nodi *server*; al fine di misurare le prestazioni della rete quali il ritardo globale ed il consumo energetico dei vari sensori, vengono analizzati tre diversi scenari applicativi:

- UNICAST DIRETTO: il «nodo 0» interroga direttamente i nodi *server* per ottenere la risorsa d'interesse.
- UNICAST TRAMITE PROXY-AVANZATO: il «nodo 0» interroga il «nodo 1» (*root*) il quale provvede a recuperare le risorse dai nodi foglia (se non ne è già in possesso) contattandoli singolarmente.
- MULTICAST TRAMITE PROXY-AVANZATO: il «nodo 0» interroga il «nodo 1» (*root*) il quale provvede a recuperare le risorse dai nodi foglia (se non ne è già in possesso) contattandoli tramite un messaggio *multicast*.

5.2.1 Unicast diretto

In questo scenario il «nodo 0» è interessato ad accedere alle risorse di tutti i nodi della rete direttamente sfruttando il *client* CoAP in esecuzione sulla macchina e l'interfaccia di rete

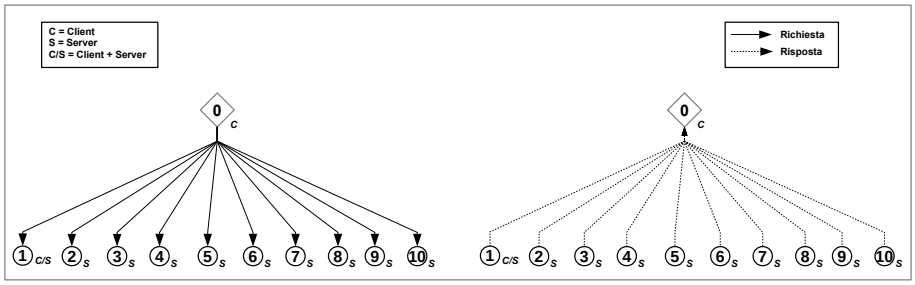


Figura 5.2: *Request/Response* nello scenario UNICAST DIRETTO.

802.15.4 di cui è predisposto. Provvede quindi ad inviare a ciascuno dei nodi *server* nella rete un messaggio *unicast* contenente una richiesta CoAP di tipo `GET` per la risorsa a cui è interessato. Ognuno dei server CoAP risponde tramite messaggio *unicast* fornendo il dato relativo alla risorsa richiesta. Lo scambio dei messaggi fra i vari nodi è mostrato in Figura 5.2.

E' stato quindi calcolato il ritardo globale, ovvero l'intervallo di tempo che intercorre fra l'invio della prima richiesta `GET` e la ricezione del dato relativo alla risorsa richiesta da parte dell'ultimo nodo interrogato. I risultati sono mostrati in Figura 5.3.

Il ritardo è direttamente proporzionale al numero di nodi presenti in rete in quanto si deve attendere una risposta dai singoli *server* che vengono contattati in sequenza.

Viene ora analizzato l'impatto sul consumo energetico dei vari nodi della rete dovuto allo scambio dei succitati pacchetti. I dati relativi al consumo energetico sono espressi in µWh e sono stati determinati rilevando il tempo effettivo di trasmissione essendo noto l'assorbimento di potenza durante la fase di trasmissione [31]:

- Trasmissione: 54 mW (@Ptx = -5 dBm)

I risultati sono mostrati in Figura 5.4.

I valori di consumo energetico dei singoli sensori in rete, incluso il PC facente da *client*, viene mostrato in funzione del numero dei sensori *server* presenti in rete ai quali esso effettua una richiesta CoAP di tipo `GET`. Il nodo per il quale si verifica il maggior consumo energetico è lo `«Sky_0»` ovvero il *client*. Esso infatti deve inviare una richiesta CoAP ad ognuno dei sensori di cui vuole conoscere il dato di risorsa, pertanto il consumo risulta essere direttamente proporzionale al numero di sensori da contattare, come meglio si evince dalla Figura

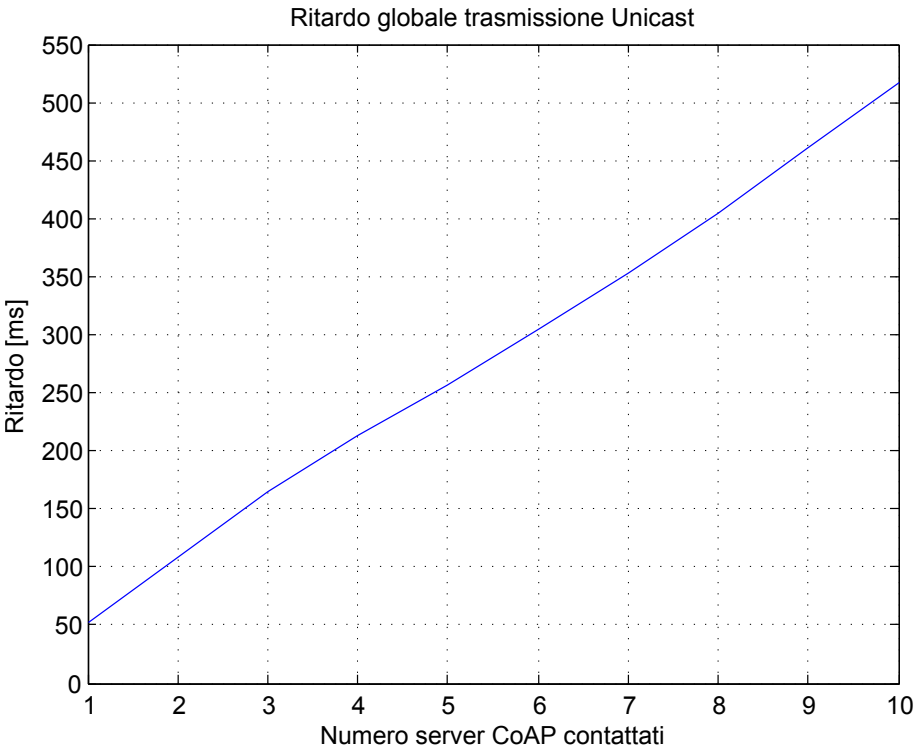


Figura 5.3: Ritardo globale nello scenario UNICAST DIRETTO.

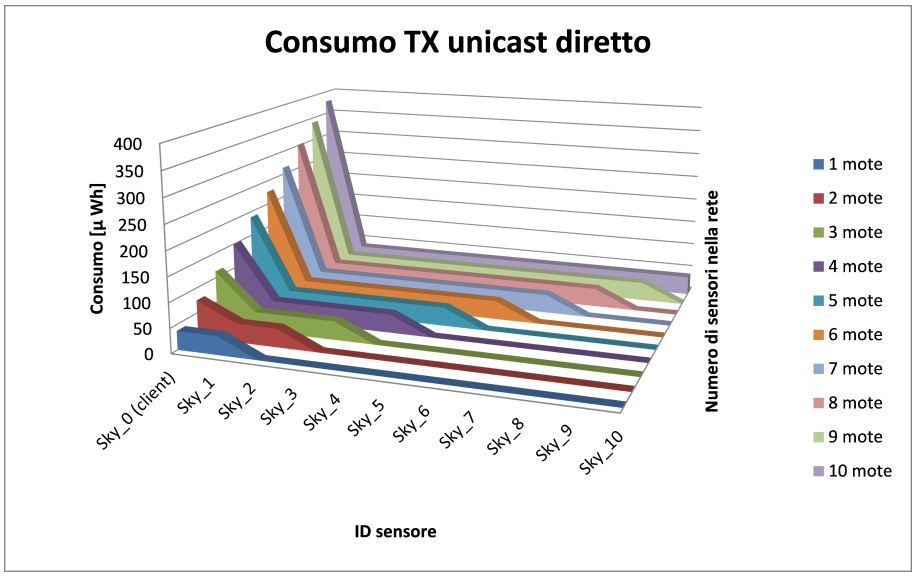


Figura 5.4: Consumo di energia in trasmissione.

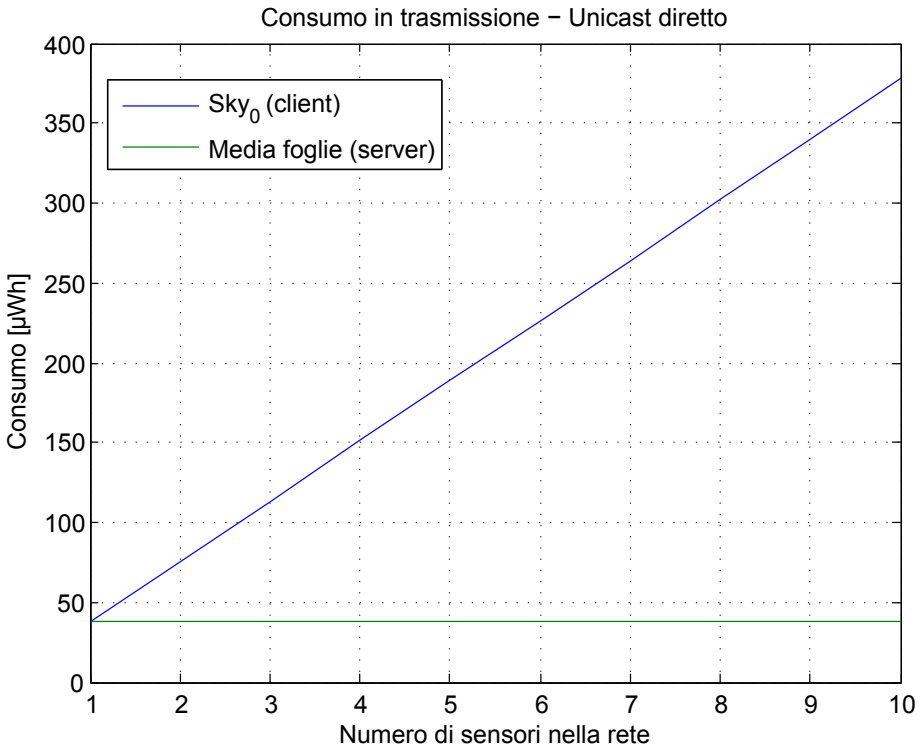


Figura 5.5: Confronto fra energia consumata dal nodo *client* e dai nodi *server*.

5.5. I nodi *server* d'altro canto dovendo rispondere ad una sola richiesta CoAP ciascuno, all'aumentare del numero di nodi in rete non vedono aumentare il loro consumo di energia relativo alla trasmissione radio, che si mantiene quindi costante.

5.2.2 Unicast tramite proxy avanzato

Anziché contattare direttamente i vari *server* CoAP, il nodo «`Sky_0`» può affidarsi ad un nodo intermedio «`Sky_1`» che svolga la funzione di *proxy* occupandosi di recuperare le informazioni relative alla risorsa di interesse per conto del nodo «`Sky_0`». Per far ciò è stata predisposta un'apposita risorsa nel nodo «`Sky_1`» che, quando richiesta, fa svolgere al nodo in oggetto la funzione di *proxy avanzato*: esso si occuperà quindi di richiedere la risorsa d'interesse ai server specificati, e di elaborare i dati ricevuti per fornirli al *client* che ha originato la richiesta inziale, come descritto più in dettaglio nei Paragrafi 3.1 e 3.2. A tale scopo il nodo *proxy* deve essere esso stesso oltre che un *server* anche un *client* CoAP . Si

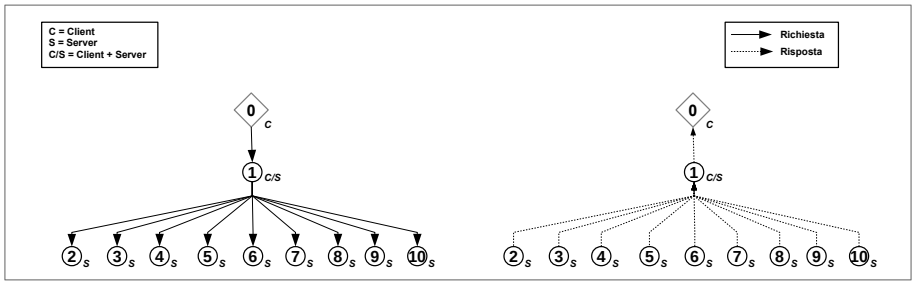


Figura 5.6: *Request/Response* nello scenario *unicast* tramite *proxy* privo di *cache*.

presentano due possibilità:

- PROXY PRIVO DI CACHE: Il nodo proxy «Sky_1» non è in possesso dei dati dei nodi server relativi alla risorsa d'interesse.
- PROXY CON CACHE: Il nodo proxy «Sky_1» possiede dati aggiornati relativi alla risorsa d'interesse.

Di seguito vengono analizzate le prestazioni di ritardo e consumo energetico per i due casi citati.

5.2.2.1 Proxy privo di cache

Nel caso qui preso in esame il nodo «Sky_0» invia una richiesta CoAP di tipo GET al *proxy* «Sky_1» richiedendo, ad esempio, la risorsa relativa al valore di temperatura media misurata da tutti i sensori della rete. Il *proxy* procede quindi tramite il client CoAP in esecuzione sul sensore a contattare tramite *request unicast* di tipo GET tutti i nodi *server* presenti in rete richiedendo la risorsa di temperatura locale, come mostrato in Figura 5.6. Dopo aver ricevuto la risposta contenente il dato di temperatura locale da tutti i nodi contattati, il *proxy* procede all'elaborazione dei dati, quindi ad inoltrare i risultati al client richiedente «Sky_0». Il consumo energetico in trasmissione dei vari nodi al variare del numero di sensori in rete è mostrato in Figura 5.7.

Analogamente al caso precedente i nodi foglia nella rete rivelano un consumo costante ed indipendente dal numero dei nodi in rete. Il maggior carico di lavoro è questa volta svolto dal nodo *proxy* che vede il suo consumo energetico crescere in maniera direttamente

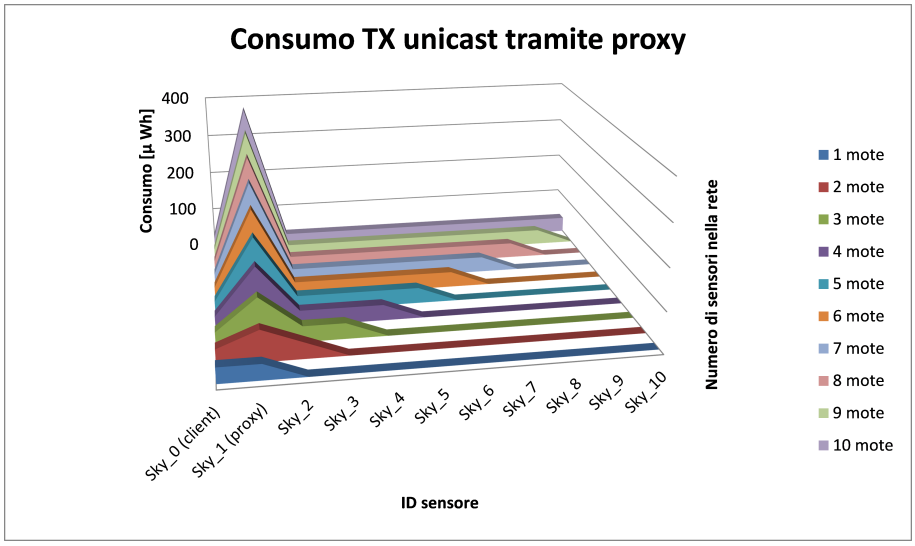


Figura 5.7: Consumo di energia in trasmissione con uso del proxy.

proporzionale ai *server* da interrogare. E' invece alleggerito l'utilizzo di energia del client «Sky_0»: all'aumentare dei nodi in rete il numero di messaggi da esso scambiato non varia così come il suo dispendio d'energia (fig. 5.8).

Per quanto concerne il ritardo globale esso rimane invariato rispetto al caso analizzato al Paragrafo 5.2.1 essendo il numero dei pacchetti scambiati nella rete il medesimo.

5.2.2.2 Proxy con cache

Volendo migliorare le prestazioni di ritardo e renderle indipendenti dal numero di *server* nella rete è possibile munire il *proxy* CoAP di funzionalità *cache*. Il *proxy* provvede periodicamente ad aggiornare la propria *cache* con i dati dei nodi foglia relativi alle risorse di interesse in modo tale da poterli fornire non appena richiesti dal *client*. La validità nel tempo di una data risorsa è espressa nell'apposita opzione «Max-Age» come descritto nel Capitolo 2.10. I benefici in termini di ritardo offerti da questo scenario sono mostrati in Figura 5.9.

Come è evidente il tempo necessario per ottenere il dato relativo alla risorsa globale è equivalente al tempo necessario per contattare un singolo *server* e non dipende quindi dalla popolazione di sensori presenti in rete. Il *proxy* infatti, nell'ipotesi che sia in possesso di tutte le risorse aggiornate, non ha necessità di contattare i vari *server* ma scambia messaggi

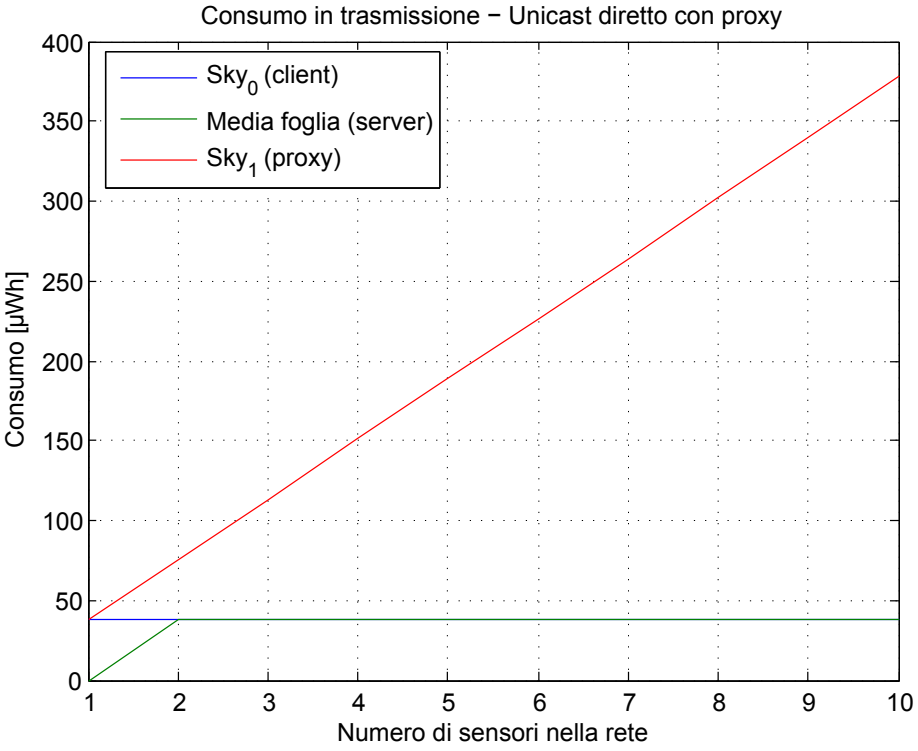


Figura 5.8: Confronto consumo energetico dei vari nodi con uso del *proxy*.

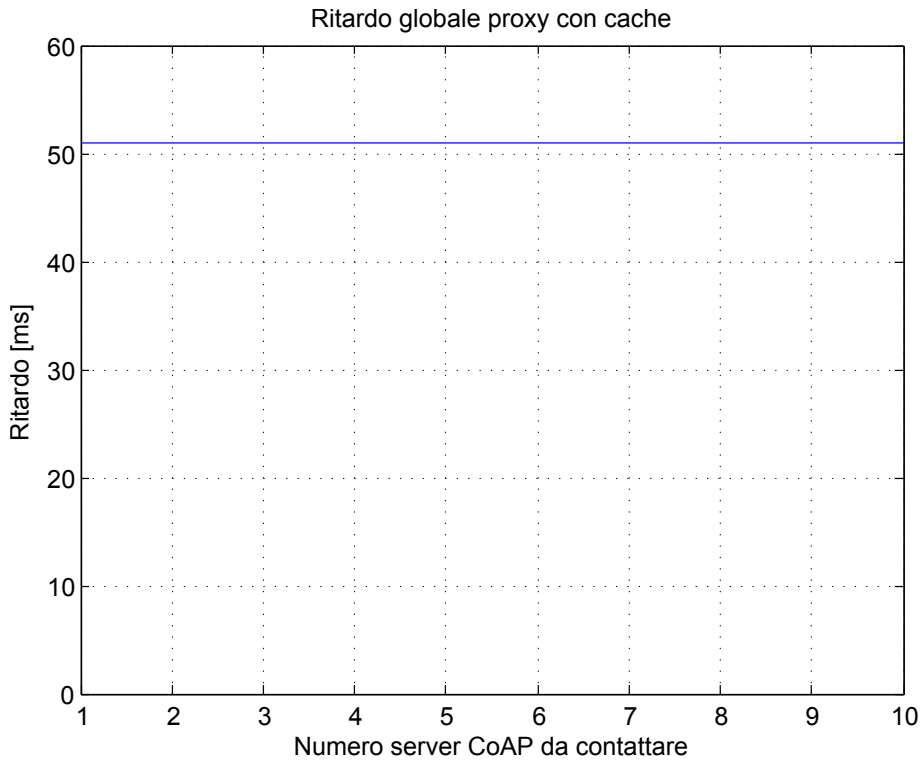


Figura 5.9: Tempo di servizio utilizzando *proxy* con *cache*.

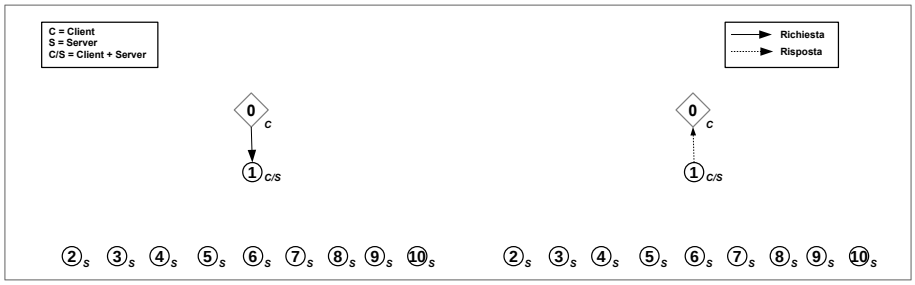


Figura 5.10: *Request/response* nello scenario *unicast* tramite *proxy* fornito di *cache*.

di *Request/Response* esclusivamente con il client richiedente (Figura 5.10).

Per contro se il periodo di aggiornamento dei dati in *cache* non è adeguatamente configurato - ad esempio frequenza di aggiornamento troppo elevata rispetto alle effettive necessità - l'utilizzo di questa soluzione porterebbe ad un inutile dispendio di energia dei sensori coinvolti. Un meccanismo per ovviare a questo problema è analizzato nel Paragrafo 3.1.

5.2.3 Multicast tramite proxy

Nei casi finora esaminati il *client* CoAP è sempre stato interessato ad ottenere le risorse da tutti i nodi *server* presenti in rete. Con questo presupposto l'utilizzo di messaggi *unicast* non è efficiente né dal punto di vista del consumo energetico né del ritardo globale. Le richieste CoAP inviate dal *client* infatti contengono sostanzialmente il medesimo *payload*! Diviene quindi più conveniente inviare un unico messaggio sfruttando l'indirizzo `«link-local multicast» ff02::1` che specifica che il pacchetto è destinato a tutti i nodi presenti nel segmento di rete locale. In alternativa sarebbe possibile utilizzare l'indirizzo `«All CoAP nodes»` ma ciò presuppone che i vari *server* siano entrati a far parte di tale gruppo. Lo scambio dei messaggi è mostrato in Figura 5.11. Il beneficio di specificare come destinatario un indirizzo *multicast* è appunto duplice: da un lato il *proxy* può inviare solamente un pacchetto di richiesta con un risparmio energetico tanto maggiore quanti più sono i nodi in rete da contattare, dall'altro il ritardo si riduce in quanto tutti i nodi *server* ricevono il messaggio di richiesta nel medesimo istante e possono immediatamente procedere alla risposta contendendosi il canale. I dettagli del consumo energetico e del ritardo sono mostrati nelle figure 5.12 e 5.13.

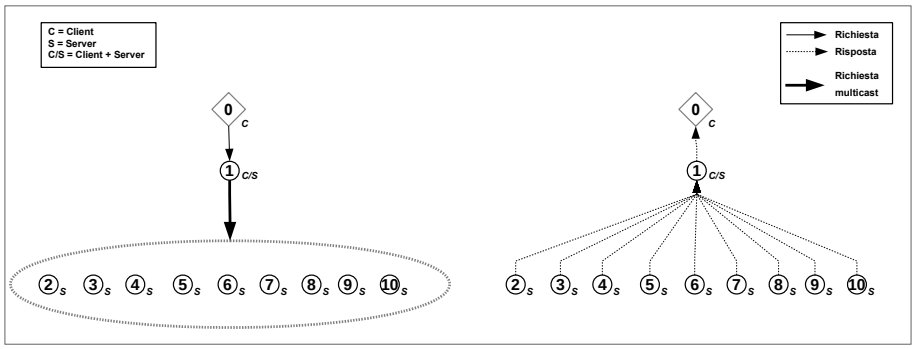


Figura 5.11: *Request/Response* nello scenario *multicast* tramite *proxy*.

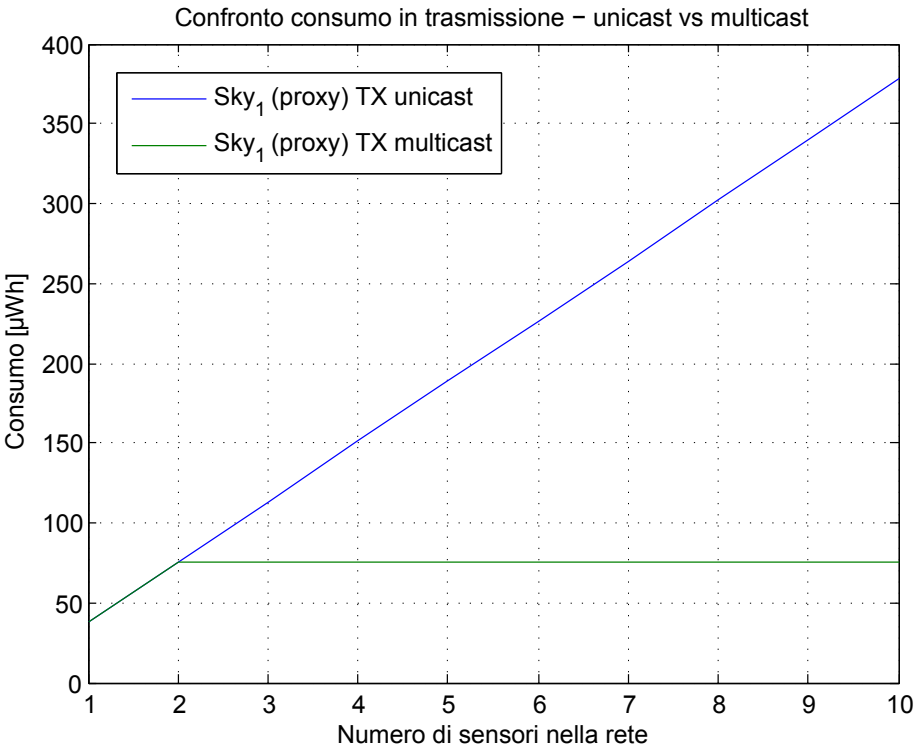


Figura 5.12: Confronto del consumo energetico fra trasmissione *unicast* e *multicast*.

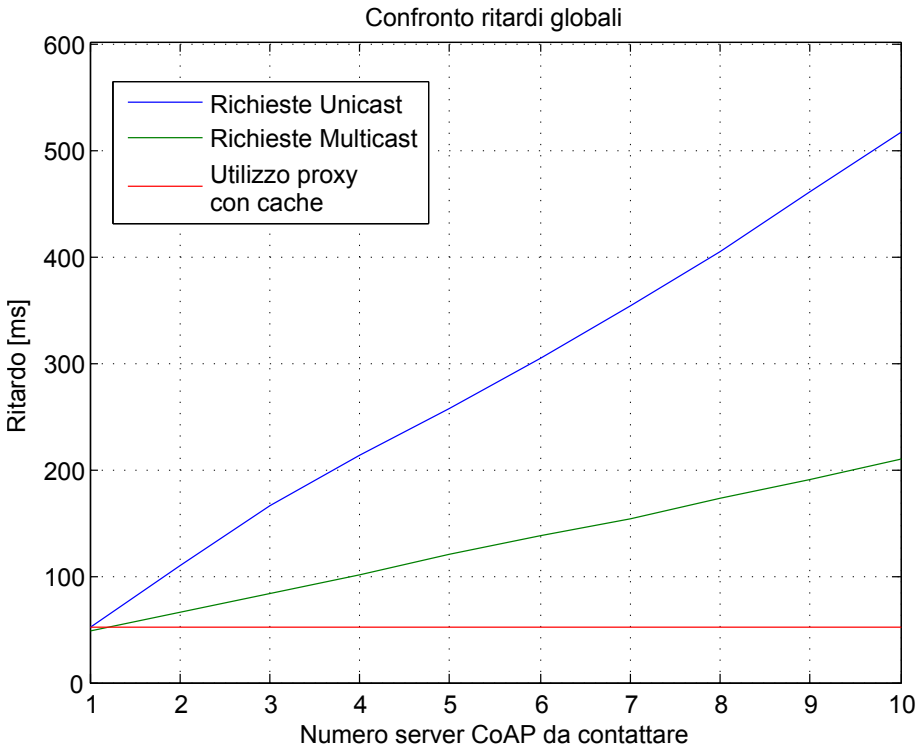


Figura 5.13: Ritardo globale con trasmissione *multicast*.

Il grafico di ritardo pone a confronto i tre casi analizzati; grazie all'invio di un unico pacchetto di richiesta il tempo impiegato per ottenere i dati, così come il consumo di energia si riduce notevolmente ed il risparmio è tanto più grande quanti più sono i *server* CoAP da contattare.

5.2.4 Unicast selettivo

Non sempre si è interessati ad ottenere i dati relativi ad una determinata risorsa da tutti i nodi presenti in rete. Volendosi servire di un *proxy* che si occupi di collezionare le risorse messe a disposizione dai *server* della rete si rende necessario l'utilizzo di un metodo per fornire al *proxy* stesso la lista di indirizzi dei server CoAP da contattare. A tale scopo è stata sviluppata una risorsa che, messa a disposizione dal *proxy* e tramite l'utilizzo del metodo PUT, permette di fornire una lista di indirizzi relativi ai *server* CoAP dei quali si vuole ottenere una determinata risorsa. La ricezione di questa *request* PUT da parte del *proxy* attiva l'invio di una serie di richieste CoAP di tipo GET, necessariamente *unicast*, relative alla risorsa d'interesse e con destinazione i nodi specificati. Al termine dell'operazione, i dati raccolti - eventualmente elaborati - vengono inviati al *client* CoAP richiedente. Il *client* CoAP può inoltre ottenere una lista dei server presenti in rete fra i quali operare una selezione, richiedendo al nodo *proxy* - che è anche root del DODAG - i dati relativi alla propria tabella di *routing* per mezzo della risorsa `«Routing Table»` (`URI: /rt`) che esso mette a disposizione.

5.3 Load Balancing

Ad esclusione del primo caso preso in esame (par. 5.2.1) in cui è direttamente il client CoAP interessato (Sky_0) ad effettuare l'invio delle richieste, si è visto come il nodo di rete sottoposto al maggior carico di lavoro e quindi al maggior consumo di energia è il nodo *proxy* (Sky_1). La Figura 5.14 riassume l'energia consumata da questo sensore nei tre casi analizzati in precedenza:

1. messaggi *unicast* diretti (in cui il nodo Sky_1 svolge la semplice funzione di server come gli altri sensori in rete)

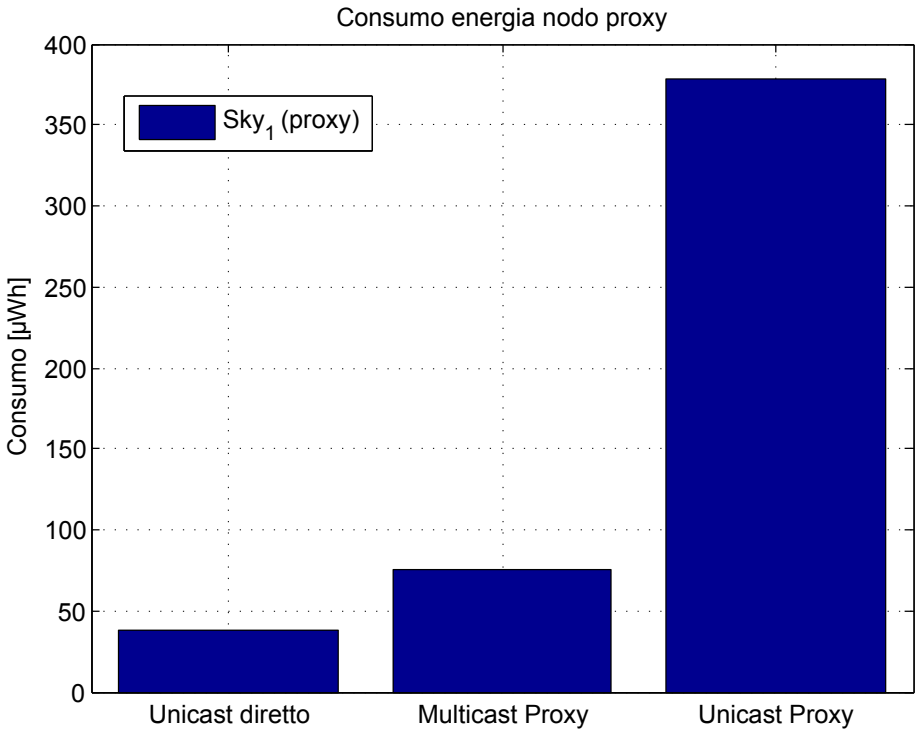


Figura 5.14: Consumo di energia del nodo *proxy* nei tre scenari analizzati.

2. messaggio *multicast* servendosi del nodo Sky_1 come *proxy*

3. messaggi *unicast* servendosi del nodo Sky_1 come *proxy*

Al fine di non sovraccaricare un unico nodo della rete ed uniformare il consumo energetico fra i vari sensori, si può fare in modo che il ruolo di *proxy* non sia svolto esclusivamente da un singolo sensore ma che periodicamente venga scelto un nodo differente basandosi sul livello di carica residua dei singoli sensori. Per far ciò il nodo designato come *proxy* effettua una richiesta CoAP ai vari nodi in rete accedendo alla risorsa `«Voltage»` (`URI: /sv`) che fornisce buona approssimazione del livello di carica residua del sensore. Il nodo *proxy*, dopo aver analizzato i dati ottenuti «elegge» il nuovo nodo *proxy* ovvero il sensore con il valore di tensione più elevato.

Requisito fondamentale per l'utilizzo di questo sistema di *load balancing* è che i sensori possano svolgere sia la funzione di *client* che di *server* CoAP; andranno quindi programmati appositamente. Fortunatamente l'esecuzione di un client CoAP non richiede un utilizzo di

	Funzionalità	ROM	RAM
DODAG root	Server	34.2 kB	6.2 kB
	Client & Server	34.8 kB	6.2 kB

Tabella 5.1: Risorse di memoria richieste da *client* e *server*.

risorse hardware aggiuntive di molto superiore rispetto all'esecuzione del solo server come mostrato dalla Tabella 5.1.

L'utilizzo di memoria RAM è il medesimo mentre la quantità di memoria ROM necessaria aumenta del solo 2%.

I nodi *proxy* inoltre non devono necessariamente essere anche *root* del DODAG ma possono ottenere la lista completa dei nodi presenti in rete accedendo alla risorsa `«Routing Table»` messa a disposizione dal nodo *root*. E' tuttavia possibile, grazie ad apposite funzioni, rimuovere dall'incarico di *DODAG root* un determinato nodo e fornire di tale funzionalità ad un altro nodo della rete (si veda il Paragrafo 3.2).

5.4 Multihop

I casi esaminati finora hanno preso in considerazione delle topologie in cui i vari nodi della rete si trovano tutti ad un *hop* di distanza l'uno dall'altro. Le reti WSN però raramente presentano un'organizzazione di questo tipo; esse sono invece caratterizzate da collegamenti *multi-hop*. Si è quindi voluto valutare l'impatto della distanza fra i nodi sul tempo necessario per ottenere la risorsa d'interesse. Il test è stato eseguito su sensori che utilizzano BLIP, e pertanto TinyRPL come protocollo di *routing*. Ogni nodo della rete svolge anche il ruolo di *router* ed è quindi in grado di inoltrare i pacchetti agli altri sensori. La topologia utilizzata è mostrata in Figura 5.15.

In rete è presente un *client* (Nodo 1) e tre *server* (Nodi 2-3-4). I casi analizzati sono:

- Il *client* 1 invia una richiesta al *server* 2; la comunicazione avviene direttamente (1 *hop*);
- Il *client* 1 invia una richiesta al *server* 3; la comunicazione avviene tramite il nodo 2 che svolge la funzione di *router* (2 *hop*);

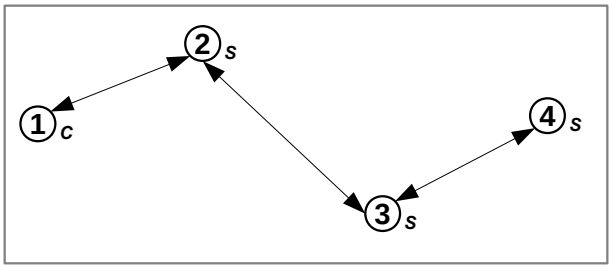


Figura 5.15: Topologia di rete *multi-hop*.

- Il *client* 1 invia una richiesta al *server* 4; la comunicazione avviene tramite i nodi 2 e 3 che svolgono la funzione di *router* (3 *hop*).

Il ritardo misurato in funzione del numero degli *hop* è mostrato in Figura 5.16. Esso include sia il tempo di trasmissione della richiesta che il tempo di ricezione della risposta o dell'*acknowledgement* (*round-trip-time*).

Nel caso di rete non sovraccarica il ritardo cresce all'incirca linearmente con il numero nodi che il pacchetto deve attraversare per giungere a destinazione, come visibile in figura; nel caso in cui vi sia molto traffico in rete vi è però da aspettarsi un aumento della varianza del ritardo. Per ottenere migliori prestazioni anche in questo caso è conveniente affidarsi ad un nodo *proxy* prossimo al client che possa procurarsi in modo pro-attivo le risorse necessarie. In questo caso il numero di *hop* necessari per raggiungere i nodi *server* diviene ininfluente sul tempo necessario per servire la richiesta.

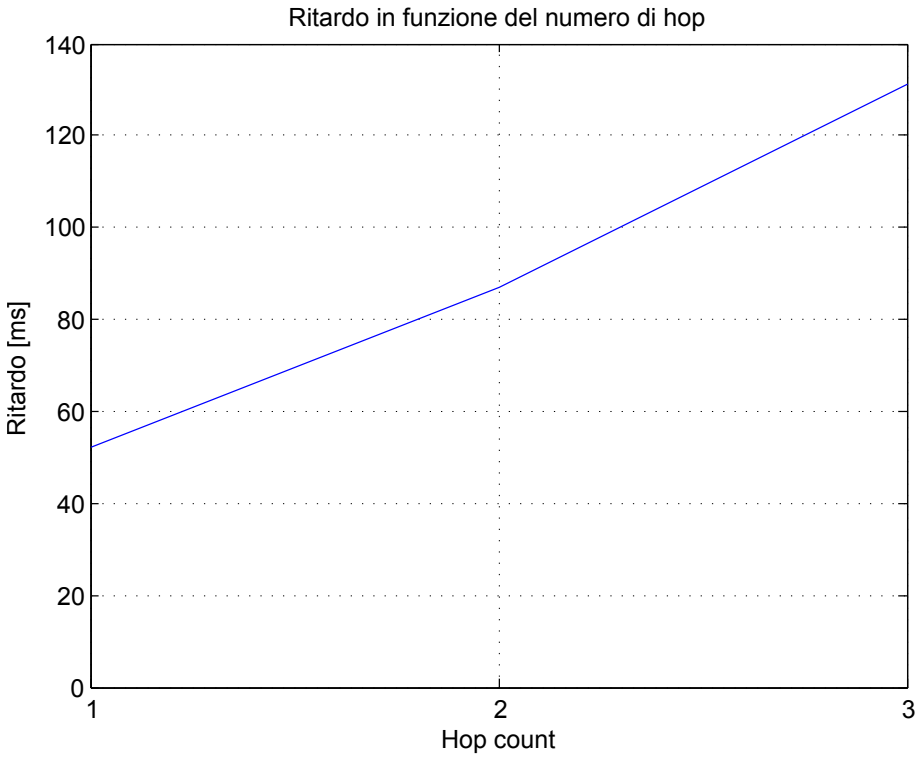


Figura 5.16: Tempo completamento richiesta in funzione del numero di *hop*.

Capitolo 6

Conclusioni

Il presente capitolo conclude il lavoro di tesi riportando alcune considerazioni relative al *proxy avanzato* qui progettato e agli eventuali sviluppi futuri.

Una rete di sensori (o *Wireless Sensor Network*, WSN) è un insieme di piccoli dispositivi elettronici in grado di ricevere dati dall'ambiente circostante e comunicare tra loro. Grande importanza riveste l'interoperabilità delle reti di sensori con la rete internet, affinché i dati rilevati dai *mote* possano essere raggiunti ovunque e da qualsiasi dispositivo. Per ottenere questo scopo, come si è visto, tutto lo *stack* di rete utilizza protocolli specifici che, oltre a limitare il consumo energetico e rendere le reti auto-organizzanti, sono atti ad ottimizzare questa interoperabilità. A livello applicazione si sta affermando lo standard CoAP sviluppato dalla IETF che, grazie alle sue analogie con HTTP, facilita lo scambio di informazioni fra dispositivi così eterogenei come lo sono *Personal Computer* e *nodi sensore*. Fra le funzionalità offerte da CoAP vi è il ruolo di *proxy*; esso può essere svolto da uno o più nodi di rete. Tuttavia per come concepiti dallo standard, in alcune applicazioni essi non portano benefici, ma causano anzi un decadimento delle prestazioni, come si è avuto modo di vedere nella topologia presa in esame in questa tesi. Essi procurano un aumento del ritardo e del consumo di energia globale della rete. A causa di queste limitazioni si è sviluppato per il sistema operativo TinyOS un *proxy* CoAP avanzato che si prefigge lo scopo di alleviare il carico di lavoro del *client* e ridurre i tempi necessari per l'ottenimento delle risorse. Lo standard CoAP

prevede che un nodo *proxy* possa fungere da intermediario per il servizio delle richieste: esso si occupa di inoltrare le singole richieste provenienti dal *client* e le eventuali singole risposte provenienti dai *server*. In alcuni scenari questo meccanismo non è affatto efficiente: nel caso la risorsa d'interesse sia la medesima per tutti i *server*, diviene inefficiente - in termini di consumo energetico e di tempo necessario per servire le richieste - inviare singole *request* CoAP. Il primo passo è stato pertanto quello di sviluppare un meccanismo che permetta di fornire al *proxy*, tramite un'unica *request*, una lista di indirizzi relativi ai server che si intendono contattare. Il *proxy* provvederà quindi, tramite richieste *unicast* piuttosto che *multicast*, a contattare i server e procurare le singole risorse d'interesse per conto del *client*. Con questo sistema il carico di lavoro del *client* si è parzialmente alleggerito in quanto il numero di *request* che esso deve emettere è pari ad uno, e diviene quindi indipendente dal numero di *server* da contattare. Il secondo *step* nella modifica del *proxy* standard parte dalla considerazione che non sempre il nodo client è interessato ai dati *grezzi*, ovvero ai dati così come forniti dai singoli sensori; esso può anzi effettuare un'elaborazione su tali dati per ottenere l'informazione di effettivo interesse. Partendo da questa ipotesi, è possibile ridurre ulteriormente il consumo energetico del *client* spostando la fase di elaborazione - e quindi il carico di lavoro - lato *proxy*. I dati elaborati, inoltre, hanno solitamente dimensione inferiore rispetto ai dati utilizzati per svolgere l'elaborazione, pertanto anche il consumo energetico dovuto alla trasmissione e ricezione dei pacchetti subirà un decremento. Il tempo di servizio delle richieste tuttavia non subisce una riduzione in quanto dalla partenza della *request* da parte del *client* intercorre un lasso di tempo durante il quale il *proxy* deve procurarsi le risorse desiderate ed elaborarle. Il terzo ed ultimo passo nello sviluppo del *proxy* avanzato è stato quindi quello di dotare lo stesso di funzionalità *cache* in cui possa memorizzare i dati elaborati per renderli immediatamente disponibili al *client* che ne faccia richiesta. Sono stati altresì studiati meccanismi affinché i dati elaborati siano sempre «freschi», minimizzando comunque il consumo di energia dei nodi coinvolti.

Il sistema sviluppato fa uso di risorse appositamente predisposte, le quali, per mezzo del metodo `PUT`, permettono di specificare al *proxy* quali siano le `URI` d'interesse e le liste di *server* da contattare. Per rendere il sistema più flessibile, un possibile sviluppo futuro del presente lavoro può riguardare l'aggiunta del supporto al *CoRE Link Format*, non ancora

pienamente supportato dall'implementazione CoAP di TinyOS, e la sua integrazione con il *proxy avanzato*. Anziché creare apposite risorse da rendere disponibili presso il *proxy*, sfruttando le opzioni `uri-query` caratterizzate da una struttura `key=value` è possibile specificare la lista dei *server* da contattare ed il tipo di elaborazioni richieste sulla risorsa oggetto della `URI` specificata.

Bibliografia

[1] Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422, 2002.

[2] Reinhard Bischoff, Jonas Meyer, and Glauco Feltrin. Wireless sensor network platforms. *Encyclopedia of structural health monitoring*, 2009.

[3] Tareq Alhmiedat, Anas Abu Taleb, and Mohammad Bsoul. A study on threats detection and tracking systems for military applications using wsns. *International Journal of Computer Applications*, 40(15):12–18, 2012.

[4] Sang Hyuk Lee, Soobin Lee, Heecheol Song, and Hwang Soo Lee. Wireless sensor network design for tactical military applications: remote large-scale environments. In *Military Communications Conference, 2009. MILCOM 2009. IEEE*, pages 1–7. IEEE, 2009.

[5] Ilker Bekmezci and Fatih Alagöz. Energy efficient, delay sensitive, fault tolerant wireless sensor network for military monitoring. *International Journal of Distributed Sensor Networks*, 5(6):729–747, 2009.

[6] Liyang Yu, Neng Wang, and Xiaoqiao Meng. Real-time forest fire detection with wireless sensor networks. In *Wireless Communications, Networking and Mobile Computing, 2005. Proceedings. 2005 International Conference on*, volume 2, pages 1214–1217. IEEE, 2005.

[7] Jaime Lloret, Miguel Garcia, Diana Bri, and Sandra Sendra. A wireless sensor network deployment for rural and forest fire detection and verification. *sensors*, 9(11):8722–8747, 2009.

[8] Scott L Collins, Luís MA Bettencourt, Aric Hagberg, Renee F Brown, Douglas I Moore, Greg Bonito, Kevin A Delin, Shannon P Jackson, David W Johnson, Scott C Burleigh, et al. New opportunities in ecological sensing using wireless sensor networks. *Frontiers in Ecology and the Environment*, 4(8):402–407, 2006.

[9] Peter Corke, Tim Wark, Raja Jurdak, Wen Hu, Philip Valencia, and Darren Moore. Environmental wireless sensor networks. *Proceedings of the IEEE*, 98(11):1903–1917, 2010.

[10] Robert Szewczyk, Eric Osterweil, Joseph Polastre, Michael Hamilton, Alan Mainwaring, and Deborah Estrin. Habitat monitoring with sensor networks. *Communications of the ACM*, 47(6):34–40, 2004.

[11] G Virone, A Wood, L Selavo, Q Cao, L Fang, T Doan, Z He, and J Stankovic. An advanced wireless sensor network for health monitoring. In *Transdisciplinary Conference on Distributed Diagnosis and Home Healthcare (D2H2)*, pages 2–4, 2006.

[12] David Malan, Thaddeus Fulford-Jones, Matt Welsh, and Steve Moulton. Codeblue: An ad hoc sensor network infrastructure for emergency medical care. In *International workshop on wearable and implantable body sensor networks*, volume 5, 2004.

[13] Carles Gomez and Josep Paradells. Wireless home automation networks: A survey of architectures and technologies. *IEEE Communications Magazine*, 48(6):92–101, 2010.

[14] Antimo Barbato, Luca Borsani, Antonio Capone, and Stefano Melzi. Home energy saving through a user profiling system based on wireless sensors. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 49–54. ACM, 2009.

[15] Hui Song, Sencun Zhu, and Guohong Cao. Svats: A sensor-network-based vehicle anti-theft system. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE, 2008.

[16] Ian F Akyildiz and Mehmet Can Vuran. *Wireless sensor networks*, volume 4. John Wiley & Sons, 2010.

[17] Ieee standard for information technology - telecommunications and information exchange between systems - local and metropolitan area networks specific requirements part 15.4: Wireless medium access control (mac) and physical layer (phy) specifications for low-rate wireless personal area networks (lr-wpans). *IEEE Std 802.15.4-2003*, 2003.

[18] Alliance, ZigBee. Zigbee specification, 2006.

[19] Nandakishore Kushalnagar, Gabriel Montenegro, C Schumacher, et al. Ipv6 over low-power wireless personal area networks (6lowpans): overview, assumptions, problem statement, and goals. *RFC4919, August*, 10, 2007.

[20] Z Shelby, S Chakrabarti, E Nordmark, and C Bormann. Neighbor discovery optimization for ipv6 over low-power wireless personal area networks (6lowpans). *Standard Track*, 6775, 2012.

[21] Tim Winter. Rpl: Ipv6 routing protocol for low-power and lossy networks. 2012.

[22] Zach Shelby, Klaus Hartke, Carsten Bormann, and B Frank. Rfc 7252: The constrained application protocol (coap). *Internet Engineering Task Force*, 2014.

[23] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[24] TinyOs Alliance. Sito web ufficiale tinyos `http://www.tinyos.net/`.

[25] David Gay, Philip Levis, David Culler, and Eric Brewer. nesc 1.3 language reference manual, 2009.

[26] Philip Levis and Nelson Lee. Tossim: A simulator for tinyos networks. *UC Berkeley, September*, 24, 2003.

[27] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with cooja. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 641–648. IEEE, 2006.

[28] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón. Cooja/mspsim: interoperability testing for wireless sensor networks. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, page 27. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009.

[29] Hugo Bitencourt, Adriano da Cunha, and Diogenes da Silva. Simulation domains for networked embedded systems.

[30] MEMSIC Inc. Sito web ufficiale memsic `http://www.memsic.com/`.

[31] Giacomo De Meulenaer, François Gosset, O-X Standaert, and Olivier Pereira. On the energy cost of communication and cryptography in wireless sensor networks. In *Networking and Communications, 2008. WIMOB'08. IEEE International Conference on Wireless and Mobile Computing,*, pages 580–585. IEEE, 2008.