



UNIVERSITÀ DEGLI STUDI DI UDINE

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Elettronica

Dipartimento di Ingegneria Elettrica, Gestionale e Meccanica

Tesi di Laurea

**Studio ed implementazione di un
proxy HTTP/CoAP per dispositivi
di elettronica di consumo**

Relatore

Chiar.mo Prof.
Roberto Rinaldo

Laureando

Cristiano Urban

Correlatore

Ing.
Roberto Cesco Fabbro

Anno Accademico 2012/13

STUDIO ED IMPLEMENTAZIONE DI UN
PROXY HTTP/CoAP PER DISPOSITIVI
DI ELETTRONICA DI CONSUMO

Cristiano Urban

Ai miei genitori

Indice

Sommario	1
Introduzione	2
1 Internet of Things	5
1.1 Ambiti di utilizzo e applicazioni	6
1.2 Criticità	6
1.3 Gli oggetti intelligenti	6
1.4 Wireless Sensor Networks (WSNs)	7
1.5 RFID Sensor Networks (RSNs)	7
2 Reti LoWPAN	9
2.1 Lo standard IEEE 802.15.4	9
2.1.1 Livello fisico	10
2.1.2 Livello MAC	10
2.2 6LoWPAN	11
2.2.1 Architettura	12
2.3 Altre implementazioni dello standard	13
3 Il protocollo CoAP	15
3.1 Caratteristiche principali (in sintesi)	16
3.2 Tipi di messaggi	16
3.3 Modello di richieste e risposte	17
3.4 Formato del messaggio	17
3.4.1 Header	18
3.4.2 Opzioni	18
3.4.3 Token value	19
3.4.4 Payload	20
3.5 Modalità di trasmissione dei messaggi	20
3.5.1 Messaggi trasmessi in maniera affidabile	20
3.5.2 Messaggi trasmessi in maniera non affidabile	21
3.6 Semantica di richieste e risposte	22
3.7 Metodi	22
3.8 Opzioni	23
3.8.1 URI-Host, URI-Port, URI-Path e URI-Query	24
3.8.2 Proxy-URI e Proxy-Scheme	24
3.8.3 Content-Format	24
3.8.4 Accept	24
3.8.5 Max-Age	25

3.8.6	ETag	25
3.8.7	Location-Path e Location-Query	25
3.8.8	Conditional Request	25
3.9	Formato degli URI	25
3.10	Blockwise transfers	26
3.10.1	Struttura di una Block Option	26
3.11	Gestione delle risorse	26
3.12	Tenere traccia delle risorse	28
3.12.1	L'opzione Observe	28
3.13	Stabilire connessioni sicure	28
3.13.1	Utilizzo del protocollo DTLS	28
4	API relative ai protocolli HTTP e CoAP	31
4.1	HTTP APIs	31
4.1.1	I messaggi HTTP	32
4.1.2	Request-Line e Status-Line	32
4.1.3	Header	33
4.1.4	Entity	34
4.1.5	L'interfaccia HttpResponseMessage	35
4.1.6	Parametri di connessione	37
4.1.7	Connessioni server-side	37
4.1.8	API lato client	39
4.2	CoAP APIs	40
4.2.1	Opzioni	41
4.2.2	La classe Message	42
4.2.3	Registri CoAP	44
4.2.4	La classe HttpTranslator	44
4.2.5	I layer e le loro funzioni	45
4.2.6	File di configurazione	46
5	Il proxy HTTP/CoAP	47
5.1	Hardware utilizzato	47
5.2	Implementazione	48
5.3	Analisi del codice sorgente	48
5.3.1	Proxy HTTP/CoAP	49
5.3.2	Client HTTP di test	52
5.4	Risultati dei test effettuati	53
5.4.1	Esempi di richieste di tipo GET	54
5.4.2	Esempi di richieste di tipo PUT	55
5.4.3	Esempio di richiesta di tipo POST	56
5.4.4	Esempio di richiesta di tipo DELETE	57
5.4.5	Esempio di richiesta di tipo OPTIONS	57
6	Conclusioni	59
	Ringraziamenti	60
A	Listato del proxy HTTP/CoAP	61
B	Listato del client HTTP	67

Bibliografia

74

Elenco delle tabelle

3.1	Valori di default di alcuni dei parametri di trasmissione previsti dal CoAP [11]	21
5.1	Costanti definite nel codice sorgente del proxy HTTP/CoAP	50

Elenco delle figure

1.1	Schema qualitativo di una Wireless Sensor Network [3]	7
2.1	Struttura dei canali per lo standard IEEE 802.15.4 [10]	11
2.2	Architettura 6LoWPAN [1]	12
3.1	I due livelli di astrazione che caratterizzano il CoAP [11]	16
3.2	Formato delle opzioni CoAP [11]	19
3.3	Formato completo dei messaggi CoAP [11]	20
3.4	Rappresentazione dei codici delle risposte contenute nei messaggi CoAP [11]	22
3.5	Rappresentazione della bit mask per i numeri delle opzioni CoAP [11]	24
3.6	Architettura Resource Directory [14]	27
3.7	Collocazione del layer DTLS [11]	29
4.1	L'interfaccia RequestLine e la sua implementazione	32
4.2	L'interfaccia StatusLine e la sua implementazione	33
4.3	L'interfaccia Header e la classe BasicHeader	34
4.4	L'interfaccia HttpEntity e le sue implementazioni	34
4.5	L'interfaccia HttpResponseMessage e le sue implementazioni	35
4.6	L'interfaccia HttpParams e le sue implementazioni	37
4.7	L'interfaccia HttpConnection e alcune delle sue implementazioni	38
4.8	L'interfaccia HttpClient e una delle sue implementazioni	39
4.9	Richieste HTTP utilizzate dal client	40
4.10	La classe Option e la sua sottoclasse BlockOption	41
4.11	La classe Message e due delle sue sottoclassi	42
4.12	La classe Request e le sue principali sottoclassi	43
5.1	Il sensore CoAP (board blu) e il border router (board rossa)	48
5.2	Schema qualitativo della rete LoWPAN	49

Elenco delle figure

Sommario

Questa tesi riguarda principalmente il CoAP, un protocollo di livello applicazione ampiamente utilizzato nell'ambito dell'Internet of Things e l'implementazione di un cross-protocol proxy HTTP/CoAP, ovvero un software in grado di stabilire una comunicazione fra client che utilizzano il protocollo HTTP e sensori con modeste capacità di elaborazione che utilizzano il CoAP.

In questa tesi vengono brevemente introdotte le caratteristiche principali e le tecnologie abilitanti che contraddistinguono il paradigma Internet of Things. Ampio spazio viene dedicato alla descrizione della struttura e dei meccanismi interni relativi al protocollo CoAP, come anche all'analisi dell'implementazione del proxy e delle librerie software utilizzate per lo sviluppo di quest'ultimo.

Introduzione

Nell'ultimo decennio il numero di dispositivi in grado di connettersi alla rete Internet è aumentato in maniera considerevole, sia grazie alla diffusione sempre più capillare delle connessioni a banda larga, sia grazie ai continui progressi della tecnologia, i quali hanno portato da un lato alla miniaturizzazione sempre più spinta dei dispositivi elettronici e dall'altro a notevoli miglioramenti nello sviluppo dei protocolli di comunicazione e nell'integrazione di moduli rice-trasmittenti all'interno di tali dispositivi.

In concomitanza allo sviluppo della rete Internet nella sua accezione più comune, cioè intesa come un insieme di router, server e personal computer, sta maturando una nuova visione di Internet in cui una moltitudine di dispositivi embedded, spesso chiamati oggetti intelligenti o *smart objects*, è in grado di connettersi alla rete utilizzando vari tipi di protocolli. Questi oggetti sono quindi parte integrante di Internet e hanno la capacità di interagire fra di loro e con l'ambiente fisico circostante.

Questo tipo di modello viene spesso identificato con il nome di Internet delle cose o Internet of Things (IoT). Esso apre a più visioni fra le quali la più emergente è quella di Web of Things (WoT) in cui è possibile astrarre questi oggetti in servizi Web che si appoggiano agli standard tutt'ora esistenti.

IoT è l'estensione di Internet al mondo degli oggetti e costituisce a tutti gli effetti un'evoluzione della rete Internet tradizionale.

Questo fenomeno, assieme a quello dell'ubiquitous computing, sta allargando la concezione di computer ad unità di calcolo sempre più presente negli ambiti più disparati della vita quotidiana. L'esatta dimensione di IoT è difficile da stimare in quanto, al contrario della rete tradizionale, essa non dipende dal numero di utenti che utilizzano regolarmente Internet. Tuttavia è possibile ipotizzare che il numero di nodi che compongono IoT supererà presto quello della rete attuale, continuando a crescere in maniera molto veloce [1].

Nella prima parte di questa tesi verrà presentata un'introduzione generale sull'Internet of Things, assieme alle applicazioni e alle criticità ad essa associate (cap. 1).

Successivamente si passerà a descrivere quelle che sono le cosiddette tecnologie abilitanti di questo paradigma (cap. 2).

In seguito l'attenzione verrà focalizzata prima sul protocollo CoAP (cap. 3) e poi sulla descrizione delle principali funzionalità delle librerie software che implementano i protocolli HTTP e CoAP (cap. 4).

Infine, l'ultimo capitolo sarà dedicato alla descrizione e all'analisi dell'implementazione di un proxy HTTP/CoAP. Verranno quindi commentati i risultati ottenuti da tale implementazione e dai test software effettuati (cap. 5).

Capitolo 1

Internet of Things

Il paradigma IoT prevede un crescente numero di dispositivi embedded di tutti i tipi in grado di comunicare e di condividere dati attraverso Internet, aprendo quindi nuovi scenari e, allo stesso tempo, nuove sfide.

Fra le varie visioni di IoT quella che oggi sembra prevalere di gran lunga è la cosiddetta visione Internet-oriented, detta anche Web of Things (WoT) [2]. Questa visione prevede che si possano utilizzare gli standard Web esistenti per far colloquiare i dispositivi fra di loro e con la rete.

In questo modo si è in grado di effettuare un'operazione di astrazione dei dispositivi, i quali vengono di fatto integrati nel Web sotto forma di servizi che possono essere scoperti¹, composti² ed eseguiti.

Le tecnologie Web applicate ai sistemi embedded consentono di offrire interfacce standard e indipendenti dalla piattaforma di utilizzo. Conseguentemente, da un lato gli utenti non devono installare software specifico, dall'altro gli sviluppatori non sono costretti a scrivere software per diverse piattaforme.

Tutt'oggi esiste la possibilità di integrare un server Web all'interno di dispositivi dotati di modeste quantità di memoria. In questo modo gli oggetti intelligenti possono, di fatto, agire come server Web, allocando risorse, scambiando dati e fornendo informazioni sul proprio stato.

Quindi, in sostanza, i punti chiave previsti dalla visione WoT sono essenzialmente due:

- integrare oggetti fisici nel Web
- fare in modo che tali oggetti forniscano servizi Web componibili ed interoperabili.

Per quanto concerne il primo punto, l'integrazione diretta dei dispositivi nel Web pone due requisiti: il supporto del protocollo IP da parte dei dispositivi e l'interoperabilità dei servizi a livello applicazione.

Riguardo al secondo punto, il W3C³ definisce due paradigmi Web: l'architettura WS-* e l'architettura REST [2].

L'architettura WS-* o *Web Services Architecture*, prevede l'utilizzo del protocollo SOAP (Simple Object Access Protocol) per lo scambio di messaggi e di un protocollo di trasporto HTTP-based. È ampiamente usata nelle applicazioni machine-to-machine (M2M) in ambito enterprise.

L'architettura REST o *REpresentational State Transfer Architecture*, viene considerata la “vera architettura del Web”. Il concetto alla base di questa architettura è quello di “risorsa”. Ogni entità fonte di informazione viene modellata come una risorsa ed è accessibile mediante un URI⁴.

¹discovery delle risorse, il concetto verrà chiarito con un esempio nella sez. 3.11

² riferito a servizi che utilizzano altri servizi

³World Wide Web Consortium

⁴Uniform Resource Identifier

1.1 Ambiti di utilizzo e applicazioni

Il paradigma IoT si presta ad essere utilizzato negli ambiti più disparati e sta portando ad una vera e propria rivoluzione di molti degli aspetti che influenzano la vita quotidiana.

Le applicazioni di maggiore rilievo si possono ricercare negli ambiti di:

domotica: basti pensare alla possibilità di programmare elettrodomestici e dispositivi in modo che comunichino e interagiscano fra di loro, con la possibilità per l'utente di controllare questi ultimi da remoto

monitoraggio medico/sanitario: questo ambito offre la possibilità di monitorare in modo continuativo le condizioni fisiche di soggetti a rischio tramite, ad esempio, l'acquisizione di dati relativi a ritmo cardiaco e pressione sanguigna

gestione intelligente dei consumi energetici: le *smart grid* sono delle reti di informazione affiancate alle linee di distribuzione elettrica per gestire in maniera ottimale l'erogazione di potenza, evitando sprechi energetici e sbalzi di tensione.

Altri ambiti di utilizzo sono, ad esempio, il monitoraggio ambientale, l'automazione industriale e la logistica.

1.2 Criticità

Dal punto di vista tecnico le principali criticità che affliggono IoT sono:

- **il numero enorme di dispositivi da gestire:** questo fatto comporta un numero elevatissimo di indirizzi IP da assegnare e, di conseguenza, il passaggio forzato a IPv6 che è in grado di gestire circa $3.4 \cdot 10^{38}$ indirizzi
- **l'autonomia e i consumi:** dato il numero enorme di dispositivi da gestire è problematico l'intervento umano ai fini di manutenzione (es. sostituzione della batteria), in particolar modo se questi sono posizionati in luoghi di difficile accessibilità. Inoltre la necessità di avere consumi ridotti pone delle notevoli restrizioni sulle prestazioni hardware e software
- **standardizzazione ed interoperabilità:** c'è la necessità di arrivare a definire degli standard comuni anche per rendere possibile la comunicazione fra dispositivi di produttori diversi
- **l'enorme mole di dati da gestire:** un numero molto alto di dispositivi connessi alla rete comporta un numero elevato di dati da analizzare e memorizzare.

1.3 Gli oggetti intelligenti

Fin qui si è parlato di Internet of Things, Web of Things e oggetti intelligenti senza entrare nel merito dell'anatomia di tali oggetti.

In generale possiamo individuare due possibili definizioni per queste entità, una di tipo tecnico e una di tipo comportamentale.

Dal punto di vista tecnico, è possibile definire uno smart object come un elemento equipaggiato con:

- un modulo sensore/attuatore
- un piccolo microprocessore/microcontrollore

- un dispositivo di comunicazione
- una sorgente di alimentazione.

Dal punto di vista comportamentale risulta più complicato definire in modo sintetico uno smart object, in quanto il comportamento di tale oggetto dipende pesantemente dall'ambito in cui esso viene utilizzato.

Tuttavia è possibile individuare due proprietà comuni a tutti gli oggetti intelligenti, quali:

- l'interazione con il mondo fisico
- la capacità di comunicare.

1.4 Wireless Sensor Networks (WSNs)

I recenti progressi tecnologici nell'ambito dei circuiti integrati a bassi consumi e in quello delle comunicazioni wireless, hanno consentito di poter disporre di sensori di dimensioni sempre più ridotte e in grado di trasmettere i dati da essi rilevati.

Questo ha fatto sì che si potessero creare delle vere e proprie reti composte da un elevato numero di sensori wireless, allo scopo di effettuare misurazioni su larga scala.

La struttura di questo tipo di reti è illustrata in Figura 1.1. I sensori trasmettono le proprie informazioni ad una stazione base la quale gestisce il traffico di dati da e verso l'esterno della rete. Per i sensori che si trovano nelle immediate vicinanze della stazione base, la trasmissione

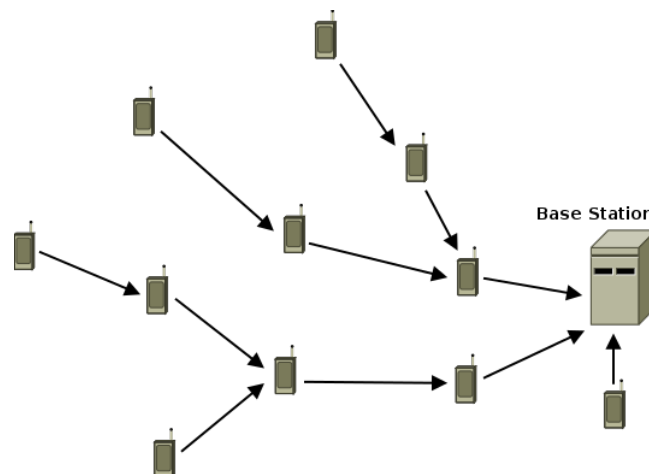


Figura 1.1: Schema qualitativo di una Wireless Sensor Network [3]

dei dati avviene in maniera diretta, mentre per quelli situati più lontano, la trasmissione avviene in maniera indiretta, cioè il percorso dei dati attraversa uno o più sensori prima di raggiungere la stazione base [3].

1.5 RFID Sensor Networks (RSNs)

Sono delle reti formate da dispositivi che utilizzano la tecnologia RFID⁵.

Il termine RFID viene spesso utilizzato per descrivere un sistema in grado di trasmettere, utilizzando onde radio, l'identità di un oggetto sotto forma di un numero seriale. Tale numero

⁵Radio Frequency IDentification

identifica l'oggetto in maniera univoca.

Le reti RFID sono simili alle Wireless Sensor Networks (sez. 1.4), ma spesso i dispositivi che le compongono utilizzano protocolli di comunicazione proprietari.

Questo tipo di tecnologia prevede la presenza di due tipi di dispositivi [4]:

tag: è un dispositivo costituito da un'antenna e da un circuito integrato dotato di memoria

lettore: è un dispositivo in grado di leggere e/o aggiornare lo stato di un tag.

Il lettore effettua un'interrogazione del tag il quale risponde inviandogli il proprio codice identificativo.

Il dispositivo tag può essere di due tipi:

- attivo: possiede una propria sorgente di alimentazione
- passivo: lavora impiegando la potenza del segnale inviatogli dal lettore.

Dal punto di vista strutturale i tag di tipo attivo sono molto simili ai nodi di una Wireless Sensor Network e differiscono da questi ultimi per minori capacità computazionali e di memorizzazione [5].

Capitolo 2

Reti LoWPAN

Con il termine LoWPAN, acronimo di Low-power Wireless Personal Area Network, si intende una rete di comunicazione senza fili a basso costo utilizzata in applicazioni caratterizzate da prestazioni e consumi limitati.

Una rete LoWPAN è generalmente composta da un insieme di dispositivi (ad esempio sensori e attuatori) che interagiscono fra di loro allo scopo di connettere l'ambiente fisico a specifiche applicazioni che utilizzano la rete Internet.

Alcune delle principali caratteristiche delle reti LoWPAN sono [6]:

- dimensioni ridotte del singolo pacchetto trasmesso a livello fisico
- larghezza di banda ridotta
- consumi ridotti (dispositivi tipicamente alimentati a batteria)
- supporto delle topologie di rete a stella e mesh.

Tuttavia vi sono anche alcuni fattori che possono rendere queste reti potenzialmente inaffidabili, quali, ad esempio, la connettività radio incerta, l'inevitabile scarica della batteria che alimenta i dispositivi, nonché il possibile blocco di uno di questi ultimi.

Inoltre, in molti ambienti, i dispositivi connessi ad una rete LoWPAN potrebbero rimanere in uno stato di "riposo" (*sleep*) per periodi più o meno prolungati al fine di risparmiare energia e quindi, durante tali periodi, essi non sono in grado di comunicare.

2.1 Lo standard IEEE 802.15.4

Spesso le reti LoWPAN si basano sullo standard IEEE 802.15.4.

Tale standard è stato concepito nell'ottica di favorire l'utilizzo di dispositivi ricetrasmittenti poco complessi e a bassi consumi di energia.

Esso è caratterizzato da una trasmissione del segnale a corto raggio (10 - 20 metri).

La dimensione massima del singolo pacchetto trasmesso è pari a 127 byte: questo in quanto il protocollo è stato pensato per reti a basse velocità di trasmissione.

Supporta tre tipologie di reti: a stella, mesh e cluster tree.

Data la notevole diffusione di dispositivi che supportano questo standard, vi sono molti stack radio a basso consumo basati su di esso, quali ad esempio ZigBee [7], WirelessHART [8] e ISA100.11a [9].

Le reti che aderiscono allo standard IEEE 802.15.4 sono suddivise in sottoreti LoWPAN le quali sono composte da due tipi di dispositivi:

- Full Function Devices (FFDs)

- Reduced Function Devices (RFDs).

I dispositivi FFD sono dotati di maggiore capacità di elaborazione rispetto agli RFDs e assumono il ruolo di “coordinatori” della rete LoWPAN di cui fanno parte. Essi possono comunicare sia con dispositivi RFD che con altri FFDs.

I dispositivi RFD, al contrario, sono dispositivi più semplici, caratterizzati da bassa capacità di elaborazione e bassi costi. Possono comunicare solamente con gli FFDs.

Sempre per quanto riguarda i dispositivi, lo standard prevede due tipi di indirizzamento, di cui uno a 64 bit e uno più corto (*short*) a 16 bit. All'interno di una LoWPAN gli indirizzi di tipo short vengono tipicamente assegnati da un dispositivo FFD e sono validi solamente all'interno di tale rete.

Lo standard IEEE 802.15.4 definisce due livelli¹:

- livello fisico
- livello MAC.

2.1.1 Livello fisico

Il livello fisico descrive le modalità attraverso le quali i messaggi vengono trasmessi e ricevuti via radio.

In particolare esso specifica le bande di frequenza, il numero di canali radio e i tipi di modulazione utilizzati [10].

Bande di frequenza utilizzate:

- Europa: 868.0 - 868.6 MHz
- Stati Uniti: 902.0 - 928.0 MHz
- In tutto il mondo: 2400.0 - 2483.5 MHz.

Canali radio utilizzati: Lo standard specifica 26 canali. Il canale 0 è definito solamente in Europa e occupa la banda 868 MHz, i canali dall'1 al 10 sono definiti solo negli Stati Uniti e occupano la banda 902-928 MHz mentre i canali dall'11 al 26 occupano la banda dei 2.4 GHz e, al contrario degli altri, sono disponibili ovunque (Fig. 2.1)

Tipi di modulazione: Vengono utilizzati due diversi tipi di modulazione, a seconda delle frequenze. I canali da 0 a 11 utilizzano una modulazione BPSK (binary phase-shift keying) mentre i canali da 11 a 26 utilizzano una modulazione QPSK (quadrature phase-shift keying).

2.1.2 Livello MAC

Il livello MAC (Media Access Control) specifica il modo in cui vengono gestiti i messaggi provenienti dal livello fisico.

Esso prevede le seguenti funzionalità [3]:

Gestione degli accessi ai canali: prima della trasmissione di un pacchetto, il livello MAC chiede al livello fisico di controllare se c'è qualche altro dispositivo che sta trasmettendo. In caso affermativo il livello MAC può attendere un tempo specifico prima di spedire il proprio pacchetto oppure può segnalare un “transmission failure” nel caso in cui abbia ritentato più volte la spedizione del pacchetto senza successo

¹riferito al modello ISO/OSI

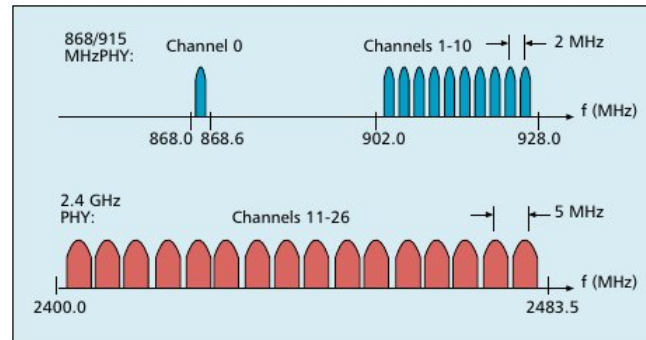


Figura 2.1: Struttura dei canali per lo standard IEEE 802.15.4 [10]

Validazione dei pacchetti in entrata: viene effettuato un controllo su eventuali errori di trasmissione tramite CRC² a 16 bit

Riconoscimento automatico dei pacchetti ricevuti: avviene solamente se l'indirizzo di destinazione del pacchetto in entrata è il medesimo del dispositivo ricevente e se la CRC sul pacchetto è valida.

2.2 6LoWPAN

6LoWPAN [1], acronimo di IPv6 over Low-power Wireless Personal Area Network, è un insieme di standard creati dall'IETF (Internet Engineering Task Force) allo scopo di rendere utilizzabile lo stack IP all'interno di reti wireless composte da dispositivi a basse prestazioni.

Questo insieme di standard definisce i meccanismi di incapsulazione e di compressione dell'header del messaggio, i quali consentono la trasmissione di pacchetti IPv6 in applicazioni "constrained". Sebbene 6LoWPAN supporti sia TCP/IP che UDP/IP, il protocollo di trasporto UDP è quello più comunemente utilizzato.

A differenza del TCP, esso è caratterizzato da un minor numero di controlli sui pacchetti che lo rende più veloce, anche se meno affidabile. Inoltre UDP è un protocollo di tipo "stateless", cioè non tiene traccia dello stato della connessione e dunque memorizza un minor numero di informazioni rispetto al protocollo TCP. Queste peculiarità lo rendono preferibile al TCP in applicazioni caratterizzate da un elevato numero di dispositivi a basse prestazioni collegati in rete, quali, ad esempio, le wireless sensor networks (sez. 1.4).

6LoWPAN in genere si usa quando [1]:

- i dispositivi necessitano di comunicare con servizi Internet-based
- varie tipologie di reti a basso consumo devono essere collegate assieme
- è necessaria la scalabilità e la rete deve essere aperta e riutilizzabile.

Il principale elemento che caratterizza 6LoWPAN è l'introduzione di un livello di adattamento (*adaptation layer*) che consente ai pacchetti IPv6 di adeguarsi allo standard IEEE 802.15.4.

Esso ha le seguenti funzioni [2]:

Compressione dell'header: è di fondamentale importanza per riuscire a trasmettere efficacemente il messaggio utile utilizzando lo standard IEEE 802.15.4. Senza compressione, infatti, ciò non sarebbe possibile

²Cyclic Redundancy Check

Frammentazione e riassemblaggio dei pacchetti: lo standard IEEE 802.15.4 supporta una MTU³ di dimensioni pari a 127 byte mentre un pacchetto IPv6 ha dimensioni pari a 1280 byte. Di conseguenza questa differenza viene gestita dall'adaptation layer

Edge routing: viene data la possibilità di collegare più reti LoWPAN fra di loro tramite degli edge routers la cui funzione principale è quella di instradare i pacchetti IPv6 di una rete da e verso l'esterno della rete stessa.

2.2.1 Architettura

L'architettura 6LoWPAN (Fig. 2.2) si ottiene collegando fra loro più reti LoWPAN. Queste ultime sono delle reti "stub" IPv6, cioè non fanno da tramite per la trasmissione di pacchetti da e verso altre reti. Una rete LoWPAN è costituita da un insieme di nodi ciascuno dei quali può

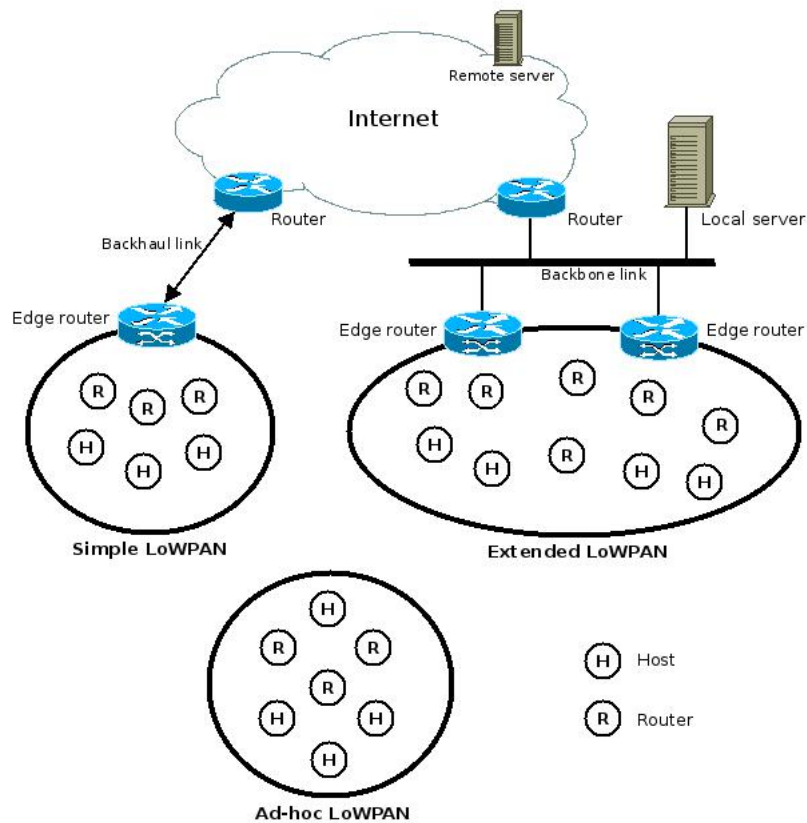


Figura 2.2: Architettura 6LoWPAN [1]

essere un host o un router ed è in grado di comunicare con l'esterno attraverso uno o più edge routers che ne gestiscono il traffico in entrata e in uscita. Tipicamente, i nodi che compongono la rete utilizzano il protocollo UDP per il trasporto dei pacchetti. Fondamentalmente esistono tre diverse tipologie di reti LoWPAN [1]:

Rete semplice (Simple LoWPAN): è dotata di un unico edge router per collegarsi ad un'altra rete IP tramite un collegamento point-to-point (*backhaul link*) oppure tramite un collegamento condiviso ad una dorsale

³Maximum Transmission Unit

Rete estesa (Extended LoWPAN): differisce da una rete semplice per la presenza di più edge routers, i quali condividono un collegamento comune ad una dorsale

Rete Ad-hoc (Ad-hoc LoWPAN): è una rete non connessa a Internet e che riesce ad operare senza la necessità di un'infrastruttura.

2.3 Altre implementazioni dello standard

Come già esposto nella sezione 2.1, l'ampia diffusione dello standard IEEE 802.15.4 ha posto le basi per la nascita di vari tipi di protocolli basati su di esso.

Fra i più importanti vanno citati:

- **ZigBee:** è il nome di una specifica proprietaria rilasciata dalla ZigBee Alliance nel 2004 per un insieme di protocolli che vengono utilizzati su dei dispositivi, dotati di piccole antenne radio, allo scopo di realizzare delle reti LoWPAN. Ad oggi le applicazioni di questa specifica si trovano in molti ambiti quali domotica, monitoraggio medico, gestione intelligente dell'energia nelle abitazioni, controllo remoto avanzato, controllo della luminosità di dispositivi a LED
- **WirelessHART:** è una specifica proprietaria che prevede l'utilizzo di reti mesh wireless ed è basata sul protocollo HART (Highway Addressable Remote Transducer). Tale specifica viene utilizzata in ambiti quali monitoraggio ambientale e di processo, sistemi di diagnostica avanzata, sistemi di controllo a circuito chiuso
- **ISA100.11a:** è uno standard proprietario sviluppato dall'International Society of Automation ed è utilizzato prevalentemente in applicazioni relative ad automazione e controllo di processo industriale.

Capitolo 3

Il protocollo CoAP

Il CoAP (Constrained Application Protocol) è un protocollo software che permette l'integrazione nella rete Internet di dispositivi elettronici caratterizzati da basse prestazioni e bassi consumi quali, ad esempio, i sensori di una wireless sensor network (sez. 1.4).

Il principale obiettivo di questo protocollo è quello di creare un insieme di interfacce REST (cap. 1) comuni con il protocollo HTTP e, allo stesso, tempo, ottimizzate per le applicazioni machine-to-machine quali, ad esempio, la gestione intelligente dell'energia con le smart grids o, ancora, l'automazione domestica attraverso le smart homes.

Nel modello di comunicazione previsto da queste applicazioni, la presenza dell'uomo è pressochè inesistente e i dispositivi interagiscono fra di loro in modo autonomo acquisendo dati, elaborando questi ultimi e generando eventi a seguito dell'elaborazione. Ad esempio, il rilevamento di una temperatura da parte di un sensore può generare un segnale di allarme se la temperatura rilevata eccede una certa soglia.

Le applicazioni machine-to-machine, solitamente, prevedono l'impiego di reti LoWPAN (cap. 2). Il CoAP nasce con l'intento di rendere possibili le interazioni di tipo REST all'interno delle reti constrained e quindi, di fatto, soddisfa i principali requisiti imposti dalle applicazioni M2M, implementando le caratteristiche che contraddistinguono le reti LoWPAN già discusse nel capitolo 2 di questa tesi.

Il modello di interazione alla base del CoAP è molto simile al modello client/server classico che caratterizza il protocollo HTTP. Tuttavia, nelle applicazioni M2M, i dispositivi in gioco, a differenza di quanto accade con il protocollo HTTP, possono assumere sia il ruolo di client che quello di server a seconda della situazione in cui si trovano ad operare.

Da un punto di vista puramente logico, il CoAP può essere pensato come un protocollo che opera su due diversi livelli di astrazione [11]:

- livello di messaggistica (*messaging layer*): il suo compito è quello di gestire la comunicazione con il protocollo UDP, fornendo inoltre delle funzionalità aggiuntive quali l'invio di messaggi in maniera affidabile e il rilevamento della duplicazione di questi ultimi
- livello di richieste/risposte (*requests/responses layer*): consente l'utilizzo di interazioni basate sull'invio di richieste contenenti dei metodi (GET, PUT, POST, DELETE) e sulla ricezione di messaggi contenenti i rispettivi codici di risposta.

Il CoAP si appoggia prevalentemente al protocollo di trasporto UDP, tuttavia può essere utilizzato anche attraverso altri protocolli, quali SMS¹, TCP o SCTP². Esso prevede anche la

¹Short Message Service

²Stream Control Transmission Protocol: protocollo unicast con caratteristiche simili al TCP

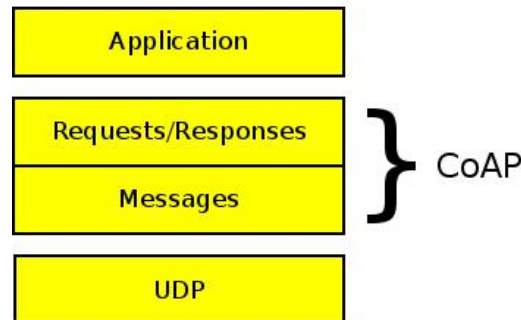


Figura 3.1: I due livelli di astrazione che caratterizzano il CoAP [11]

possibilità di stabilire connessioni sicure attraverso il protocollo DTLS³ (sez. 3.13.1). Il CoAP prevede, inoltre, le seguenti funzionalità [12]:

- osservazione delle risorse: un'estensione del protocollo consente a un client CoAP di ottenere l'aggiornamento dello stato di una o più risorse quando queste subiscono delle variazioni nella loro rappresentazione
- blockwise transfers: nel caso in cui vi sia la necessità di scambiare messaggi la cui dimensione eccede quella di un datagramma, il protocollo prevede l'utilizzo di apposite opzioni che consentono la trasmissione di dati in più blocchi
- comunicazione di gruppo: è prevista la possibilità di inviare dati simultaneamente a più nodi attraverso una trasmissione di tipo IP multicast
- discovery delle risorse: per la discovery delle risorse vengono utilizzati degli URI che includono il path speciale ".well-known/core" (a partire dal quale è possibile accedere alla lista delle risorse disponibili) e degli attributi che descrivono le risorse.

3.1 Caratteristiche principali (in sintesi)

Il protocollo CoAP:

- soddisfa i requisiti imposti dalle applicazioni M2M
- utilizza principalmente il protocollo di trasporto UDP
- prevede lo scambio di messaggi in maniera asincrona
- è in grado di gestire richieste unicast e multicast
- può stabilire connessioni sicure grazie al protocollo DTLS.

3.2 Tipi di messaggi

Il modello dei messaggi definito dal CoAP è basato sullo scambio di messaggi fra gli endpoint⁴ attraverso il protocollo di trasporto UDP.

Il CoAP definisce quattro tipi di messaggi [11]:

³Datagram Transport Layer Security

⁴le entità che partecipano alla comunicazione attraverso il protocollo CoAP

Confirmable (CON): messaggio trasmesso in maniera affidabile

Non-confirmable (NON): messaggio trasmesso in maniera non affidabile

Acknowledgement (ACK): serve per confermare la ricezione di un messaggio di tipo CON

Reset (RST): conferma la ricezione di un messaggio, ma indica l'impossibilità di processarlo.

3.3 Modello di richieste e risposte

La semantica di una richiesta o di una risposta viene trasportata all'interno dei messaggi, i quali includono rispettivamente un *method code* per definire il tipo di azione associata ad una richiesta e un *response code* che definisce il tipo di risposta da inviare per una data richiesta.

Per abbinare le risposte alle rispettive richieste viene utilizzato un *token*.

Tali codici verranno descritti in maniera più esauriente nella sezione 3.4.

Una richiesta può essere trasportata sia tramite un messaggio Confirmable che tramite un messaggio Non-confirmable.

Nel caso in cui una richiesta venga trasportata tramite un messaggio CON, le modalità di risposta possono essere di due tipi [11]:

- **risposta “piggy-backed”:** la risposta è immediatamente disponibile e viene trasportata all'interno di un messaggio ACK
- **risposta separata:** quando il server non è in grado di rispondere immediatamente ad una richiesta, esso risponde semplicemente con un messaggio ACK vuoto in modo tale che il client smetta di ritrasmettere la richiesta. Quando la risposta è pronta, il server la spedisce all'interno di un messaggio CON. La risposta deve poi essere riconosciuta dal client tramite l'invio di un messaggio ACK al server.

Se invece una richiesta viene spedita tramite un messaggio NON, la rispettiva risposta viene spedita separatamente, solitamente tramite un messaggio NON, sebbene vi sia la possibilità che il server la spedisca tramite un messaggio CON.

3.4 Formato del messaggio

I messaggi che utilizzano il protocollo CoAP prevedono un formato binario composto dai seguenti campi [11]:

- **header** di lunghezza fissa pari a 4 byte
- **token value** (o semplicemente *token*) di lunghezza variabile fra 0 e 8 byte
- sequenza di zero o più **opzioni** in formato TLV⁵
- **payload** (opzionale) che occupa il resto del datagramma.

⁵Type-Length-Value

3.4.1 Header

L'header, o intestazione, è un campo che contiene varie informazioni sul messaggio ed è posizionato all'inizio del datagramma.

Nel protocollo CoAP l'header del messaggio è composto dai seguenti campi [11]:

Version (Ver): intero a 2 bit senza segno. Indica la versione del CoAP e va settato a 1. Valori diversi da 1 sono riservati per le future versioni

Type (T): intero a 2 bit senza segno. Indica il tipo di messaggio: CON (0), NON (1), ACK (2) o RST (3)

Token Length (TKL): intero a 4 bit senza segno. Indica la lunghezza (variabile) del campo *token*

Code: intero a 8 bit senza segno. Indica se il messaggio contiene una richiesta (valori da 1 a 31), una risposta (valori da 64 a 191) o se è vuoto (0). In caso di richiesta indica il metodo associato alla richiesta (*method code*) mentre in caso di risposta indica il codice della risposta (*response code*)

Message ID: intero a 16 bit senza segno. È principalmente utilizzato per individuare la duplicazione dei messaggi. Viene generato dall'endpoint mittente di un messaggio.

3.4.2 Opzioni

Il CoAP definisce un certo numero di opzioni che possono essere incluse nei messaggi.

Un messaggio può contenere più opzioni e ogni istanza di un'opzione specifica:

- il numero dell'opzione CoAP definita
- la lunghezza del valore dell'opzione (Option Length)
- il valore dell'opzione (Option Value).

Il numero dell'opzione CoAP non viene specificato in modo diretto. Per ogni istanza di un'opzione, il numero associato a quest'ultima viene calcolato come la somma di un delta e del numero che rappresenta l'istanza dell'opzione precedente (cioè come la somma di tutti i delta delle istanze precedenti).

Le istanze devono apparire in ordine rispetto ai loro numeri di opzione. Per la prima istanza in un messaggio, si assume che essa sia preceduta da un'istanza con numero di opzione pari a zero. Multiple istanze di una stessa opzione possono essere incluse utilizzando un delta pari a zero.

I campi di un'opzione sono definiti come segue:

Option Delta: intero a 4 bit senza segno. Un valore da 0 a 12 indica il delta dell'opzione

Option Length: intero a 4 bit senza segno. Un valore da 0 a 12 indica la lunghezza (in byte) dell'*option value*

Option Value: una sequenza di byte di lunghezza pari a *option length*, la cui lunghezza e il cui formato dipendono dalla rispettiva opzione.

Per l'*option delta* e l'*option length* i valori 13, 14 e 15 sono riservati per usi speciali.

L'*option value*, a sua volta, può assumere i seguenti valori:

- empty: sequenza di byte di lunghezza nulla

- opaque: sequenza di byte opaca (non viene processata)
- uint: un intero non negativo formato da un numero di byte specificato dal campo *option length* e rappresentato in network byte order⁶
- string: una stringa Unicode codificata usando UTF-8⁷ nella forma Net-Unicode.

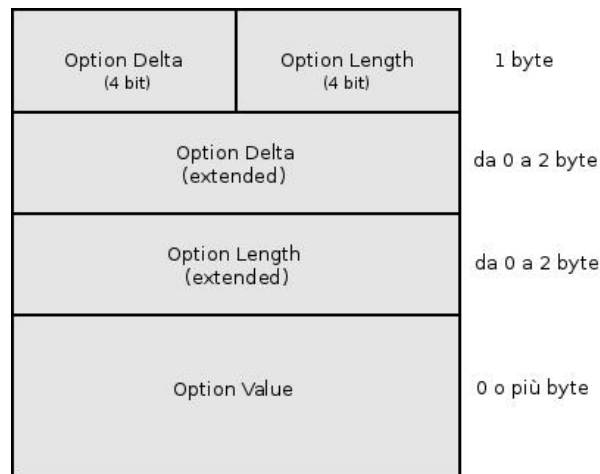


Figura 3.2: Formato delle opzioni CoAP [11]

3.4.3 Token value

Una risposta viene abbinata alla rispettiva richiesta per mezzo di un *token* che viene incluso dal client nella richiesta.

Il token è una sequenza di byte variabile da 0 a 8. Esso è utilizzato in modo “locale” per differenziare richieste concorrenti. Può essere inteso come un identificatore di una richiesta.

Ogni messaggio trasporta un token, il quale può anche essere di lunghezza nulla.

Il client dovrebbe generare i token in maniera tale che quelli attualmente in uso, per una data associazione di endpoint, siano unici.

Regole per l’abbinamento (matching rules)

- L’endpoint sorgente della risposta deve coincidere con l’endpoint destinazione della richiesta originaria
- l’incorrere di un messaggio inaspettato contenente una risposta comporta il rifiuto della stessa
- per quanto concerne una risposta piggy-backed, i *Message ID* della richiesta CON e dell’ACK devono combaciare fra loro, cosa che deve valere anche per i token della risposta e della richiesta originaria
- per quanto riguarda una risposta separata, solamente il token della risposta e quello della richiesta originaria devono combaciare.

⁶l’ordine big-endian

⁷Unicode Transformation Format, 8 bit

3.4.4 Payload

Il *payload* rappresenta la parte utile del messaggio.

Se presente e di lunghezza non nulla, è preceduto da un *payload marker* di lunghezza fissa pari a 1 byte e di valore pari a $0xFF^8$. Il payload marker indica la fine delle opzioni e l'inizio del payload.

L'assenza del payload marker denota un payload di lunghezza nulla mentre la presenza dello stesso, seguito da un payload di lunghezza nulla, va processata come “formato errato del messaggio”.

Un messaggio vuoto ha il campo *Code* settato a zero. In questo caso il campo *TKL* deve essere settato a zero e non ci devono essere dei byte dopo il campo *Message ID*, altrimenti essi devono essere processati come formato errato del messaggio.

I dati del payload si estendono fino alla fine del datagramma UDP e la lunghezza del payload viene calcolata in base all'intera dimensione del datagramma.

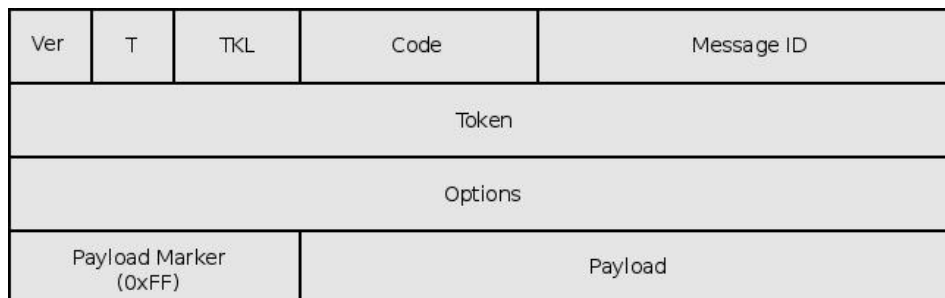


Figura 3.3: Formato completo dei messaggi CoAP [11]

3.5 Modalità di trasmissione dei messaggi

I messaggi CoAP vengono scambiati in maniera asincrona fra gli endpoint.

Il CoAP è spesso abbinato a protocolli di trasporto non affidabili come, ad esempio, l'UDP.

Esso implementa un meccanismo di affidabilità leggero tentando di ricreare un insieme completo di caratteristiche simile a quello che contraddistingue il protocollo TCP. Tale meccanismo prevede le due seguenti caratteristiche [11]:

- trasmissione affidabile con exponential back-off per i messaggi CON
- individuazione della duplicazione dei messaggi sia per i messaggi CON che per i messaggi NON.

3.5.1 Messaggi trasmessi in maniera affidabile

La trasmissione affidabile di un messaggio inizia marcando il messaggio come CON nell'header.

Un messaggio CON trasporta sempre una richiesta o una risposta e non deve essere vuoto a meno che non venga usato per suscitare un messaggio di Reset (RST).

Il destinatario (*recipient*) deve riconoscere il messaggio inviando un messaggio ACK oppure, nel caso in cui non riesca a processare correttamente il messaggio (es. message format error), deve rifiutarlo.

⁸ $0xFF = (FF)_{16} = (11111111)_2$

Il messaggio ACK deve avere lo stesso Message ID del messaggio CON e deve trasportare una risposta o essere vuoto.

Per rifiutare un messaggio CON, il destinatario, può spedire un messaggio RST oppure semplicemente ignorare il messaggio in entrata.

Il mittente (*sender*) ritrasmette il messaggio CON a intervalli di tempo esponenziali crescenti fino a che non riceve un ACK o un RST, o esaurisce il numero massimo di tentativi di ritrasmissione.

La ritrasmissione è controllata da due fattori, di cui un endpoint deve tenere traccia per ogni messaggio CON che invia, mentre attende un ACK o un RST [11]:

- un timeout: per un nuovo messaggio CON il timeout iniziale è settato ad un numero casuale compreso fra i valori ACK_TIMEOUT e (ACK_TIMEOUT · ACK_RANDOM_FACTOR)
- un contatore di ritrasmissione (*retransmission counter*): per un nuovo messaggio CON è settato a zero.

Exponential back-off mechanism

Quando il timeout viene raggiunto e il retransmission counter è minore del valore MAX_RETRANSMIT, il messaggio viene ritrasmesso, il retransmission counter incrementato e il timeout raddoppiato.

Se il retransmission counter raggiunge il valore MAX_RETRANSMIT in un timeout o se l'endpoint riceve un RST, il tentativo di trasmettere il messaggio è cancellato e il processo viene informato del failure.

D'altra parte, se invece l'endpoint riceve un ACK in tempo, la trasmissione si considera effettuata con successo.

3.5.2 Messaggi trasmessi in maniera non affidabile

In molti casi i messaggi non richiedono un ACK. Basti pensare, ad esempio, alle letture effettuate ripetutamente da un sensore.

Un messaggio può essere trasmesso in maniera non affidabile marcandolo nell'header come NON. Un messaggio NON non deve ottenere un ACK dal recipient. Se il recipient non riesce a processare correttamente il messaggio, deve rifiutarlo. Anche qui, per rifiutare il messaggio, esso può spedire un RST oppure semplicemente ignorarlo.

A livello CoAP non c'è modo per il sender di sapere se un NON è stato ricevuto o meno.

Il sender potrebbe decidere di trasmettere copie multiple di un messaggio NON entro un tempo pari a MAX_TRANSMIT_SPAN. Per permettere al receiver di agire su di un'unica copia del messaggio, i messaggi Non-confirmable specificano un Message ID.

Parametro	Valore
ACK_TIMEOUT	2 secondi
ACK_RANDOM_FACTOR	1.5
MAX_RETRANSMIT	4
MAX_TRANSMIT_SPAN	45 secondi

Tabella 3.1: Valori di default di alcuni dei parametri di trasmissione previsti dal CoAP [11]

3.6 Semantica di richieste e risposte

Il CoAP opera secondo un modello di richieste e risposte simile a quello utilizzato dal protocollo HTTP.

Diversamente da quest'ultimo, però, le richieste e le risposte non vengono spedite tramite una connessione stabilita precedentemente, ma vengono scambiate in modo asincrono attraverso i messaggi CoAP.

Il CoAP supporta alcuni metodi di base che sono facilmente mappati dal protocollo HTTP, quali [11]:

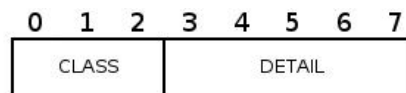
- GET
- POST
- PUT
- DELETE.

Una richiesta viene inizializzata settando il campo *Code* nell'header di un messaggio con il codice relativo al metodo e includendo le informazioni della richiesta.

Dopo aver ricevuto ed interpretato una richiesta, il server invia una risposta, la quale è abbinata alla rispettiva richiesta per mezzo di un token generato dal client (sez. 3.4.3).

Una risposta è identificata dal campo *Code* nell'header del messaggio, settando tale campo con il codice della risposta.

L'elenco completo dei codici di risposta che devono essere settati nel campo *Code* dell'header, si trova nel "CoAP Response Code Registry" [11]. Per quanto concerne i codici delle risposte,



Esempio: Response Code 4.03 (Forbidden)

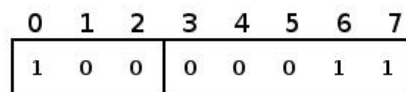


Figura 3.4: Rappresentazione dei codici delle risposte contenute nei messaggi CoAP [11]

sono previsti tre tipi di classi:

- 2 (Success): richiesta ricevuta, interpretata ed accettata correttamente
- 4 (Client Error): la richiesta contiene una sintassi errata oppure non può essere soddisfatta
- 5 (Server Error): il server fallisce l'adempimento ad una richiesta apparentemente valida.

3.7 Metodi

È possibile definire ciascun metodo CoAP in base al tipo di comportamento da esso adottato:

GET: recupera le informazioni relative alla rappresentazione di una determinata risorsa, la quale è identificata dall'URI della richiesta.

Il metodo GET è sicuro⁹ ed idempotente¹⁰

POST: richiede che una rappresentazione, racchiusa in una richiesta, venga processata. La funzione eseguita dal metodo è determinata dal server che mette a disposizione la risorsa (*origin server*) e dipende dalla "risorsa target". La funzione può consistere nella creazione di una nuova risorsa o nell'aggiornamento della risorsa target esistente.

Il metodo POST non è nè sicuro, nè idempotente

PUT: richiede che una risorsa identificata da un URI di richiesta, sia aggiornata o creata con la rappresentazione inclusa nella richiesta. Il formato di rappresentazione è specificato nell'opzione "Content-Format".

Il metodo PUT non è sicuro ma è idempotente

DELETE: richiede che una risorsa, identificata da un URI di richiesta, venga cancellata. Il metodo DELETE non è sicuro ma è idempotente.

3.8 Opzioni

Sia le richieste che le risposte possono includere una lista di una o più opzioni. Il CoAP definisce un singolo insieme di opzioni valido sia per le richieste che per le risposte.

Classificazione

Le opzioni si possono suddividere in due tipi di classi, a seconda di come un'opzione, non riconosciuta, viene gestita da un endpoint [11]:

- **critical:** le opzioni non riconosciute della classe "critical", presenti in un messaggio di tipo NON, causano il rifiuto di tale messaggio. Quelle presenti all'interno di una risposta di tipo CON o piggy-backed in un ACK, causano il rifiuto di tale risposta, mentre quelle che si verificano in una richiesta di tipo CON, devono causare una risposta di tipo "Bad Option". Tale risposta dovrebbe includere un payload di diagnostica che descrive le opzioni non riconosciute
- **elective:** al momento della ricezione, le opzioni non riconosciute della classe "elective", devono essere silenziosamente ignorate.

Riferendosi all'utilizzo di un proxy, è possibile effettuare un'ulteriore distinzione per le opzioni [11]:

- **unsafe:** un'opzione che necessita di essere interpretata dal proxy che riceve il messaggio, prima di poter inoltrare tale messaggio
- **safe:** un'opzione che può essere inoltrata dal proxy senza bisogno di essere interpretata.

⁹non modifica in alcun modo la rappresentazione di una risorsa

¹⁰l'effetto di più richieste identiche è lo stesso di quello di una sola richiesta

I numeri delle opzioni (option numbers)

Un'opzione è identificata da un valore numerico, il quale fornisce anche alcune informazioni aggiuntive.

Per esempio, i numeri dispari indicano un'opzione "critical" mentre i numeri pari un'opzione "elective".

Un option number viene costruito tramite una bit mask, illustrata in Figura 3.5. Se il bit 7 è a

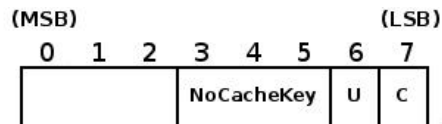


Figura 3.5: Rappresentazione della bit mask per i numeri delle opzioni CoAP [11]

1, l'opzione è "critical" mentre, se è a 0, è "elective".

Se il bit 6 è a 1, l'opzione è "unsafe" mentre, se è a 0, è "safe".

Quando il bit 6 è a 0 non c'è una cache-key, se e solo se, i bit da 3 a 5 sono settati a 1, altrimenti c'è una cache-key.

Qui di seguito verranno brevemente descritti i tipi di opzioni previsti dal protocollo CoAP.

3.8.1 URI-Host, URI-Port, URI-Path e URI-Query

Vengono utilizzate per specificare la risorsa target di una richiesta ad un origin server:

- URI-Host: specifica l'host Internet della risorsa richiesta
- URI-Port: specifica il numero della porta
- URI-Path: specifica un segmento del path assoluto della richiesta
- URI-Query: specifica un argomento, parametrizzando la risorsa.

3.8.2 Proxy-URI e Proxy-Scheme

L'opzione Proxy-URI viene utilizzata per effettuare una richiesta ad un "forward proxy". Un forward proxy potrebbe inoltrare la richiesta ad un altro proxy o direttamente al server specificato dall'URI assoluto. Tale opzione, se inserita nel messaggio, ha la precedenza sulle opzioni URI-Host, URI-Port, URI-Path e URI-Query.

L'opzione Proxy-Scheme permette di costruire l'URI assoluto a partire dalle opzioni URI*.

3.8.3 Content-Format

L'opzione Content-Format indica il formato di rappresentazione del payload. La rappresentazione del formato è fornita tramite un identificatore numerico definito nel "CoAP Content-Format Registry" [11].

3.8.4 Accept

Può essere utilizzata per indicare quale Content-Format è accettabile dal client. La rappresentazione del formato è data tramite un identificatore numerico definito nel CoAP Content-Format Registry.

Se tale opzione non viene inclusa nel messaggio, il client non esprime alcuna preferenza.

3.8.5 Max-Age

Indica il tempo massimo che una risposta può rimanere in cache prima che essa venga considerata “scaduta”.

L’*option value* è un numero compreso fra 0 e $(2^{32} - 1)$. Il valore di default è pari a 60 secondi.

3.8.6 ETag

Identifica risorse locali al fine di distinguere fra rappresentazioni della stessa risorsa che variano nel tempo. È generata dal server che fornisce la risorsa.

Se presente in una risposta fornisce il valore corrente della entity-tag per la rappresentazione della risorsa. Se inclusa in una richiesta di tipo GET, viene utilizzata da un client per interrogare il server in modo da capire se una data risposta, precedentemente archiviata nella cache del client, è valida o se invece deve subire un “refresh”.

3.8.7 Location-Path e Location-Query

L’opzione Location-Path specifica una parte del path assoluto della risorsa, mentre l’opzione Location-Query specifica un argomento che parametrizza la risorsa.

Le due opzioni, messe assieme, indicano un URI relativo che consta di un path assoluto, una query string o entrambi.

3.8.8 Conditional Request

Un’opzione Conditional Request consente al client di chiedere al server di eseguire una richiesta solo se certe condizioni specificate dall’opzione sono soddisfatte.

Le opzioni Conditional Request possono essere di due tipi:

- If-Match: viene utilizzata per effettuare una richiesta condizionata all’esistenza di un’ETag o al valore assunto da quest’ultima, per una o più rappresentazioni della risorsa target
- If-None-Match: viene utilizzata per effettuare una richiesta condizionata alla non esistenza di una risorsa target.

3.9 Formato degli URI

Il formato di un URI di tipo CoAP si presenta come segue [11]:

```
"coap(s)://" host [ ":" port ] path [ "?" query ]
```

coap(s): la sintassi “coap://” si utilizza per una connessione standard, mentre la sintassi “coaps://” viene utilizzata per connessioni sicure che utilizzano il protocollo DTLS

host: è l’host Internet al quale il server è raggiungibile. Può essere un nome o, più comunemente, un indirizzo IP

port: indica la porta UDP alla quale il server è in ascolto. La porta di default è la 5683

path: identifica una risorsa. È costituito da un insieme di segmenti separati dal carattere / (slash)

query: parametrizza una risorsa. È formata da una sequenza di argomenti separati dal carattere & (ampersand). La sintassi degli argomenti è del tipo “key = value”.

3.10 Blockwise transfers

Il modello base dei messaggi previsto dal CoAP funziona correttamente per payload di dimensioni ridotte come, ad esempio, quelli contenuti all'interno di messaggi che rappresentano le letture di un sensore.

Tuttavia in alcuni casi vi è la necessità di scambiare messaggi caratterizzati da payload di dimensioni più elevate, ad esempio, quando si deve effettuare l'aggiornamento di un firmware. Per gestire questo tipo di situazione il CoAP prevede due opzioni, chiamate rispettivamente Block1 e Block2 [13], che consentono la trasmissione dei dati in più blocchi. L'opzione Block1 si riferisce al payload di una richiesta mentre l'opzione Block2 si riferisce al payload di una risposta.

L'introduzione di queste opzioni ha come principale obiettivo quello di limitare le dimensioni del singolo datagramma cercando di evitare il più possibile la frammentazione.

3.10.1 Struttura di una Block Option

Le block options hanno una struttura di lunghezza variabile fra 0 e 3 byte e ognuna di esse è caratterizzata da tre parametri [13]:

- **SZX**: intero a 3 bit senza segno. Questo parametro viene utilizzato per codificare la dimensione (in byte) del blocco utilizzando l'espressione 2^{SZX+4} . Assume valori compresi fra 0 e 6. Il valore 7 è riservato
- **M**: flag a 1 bit. Se settato a 1 indica la presenza di altri blocchi che seguono quello corrente, altrimenti indica che il blocco corrente è l'ultimo che deve essere trasferito
- **NUM**: è il numero che identifica il blocco richiesto, presente all'interno di una sequenza di blocchi di una data dimensione. Il valore 0 indica il primo blocco dell'intero corpo del messaggio.

Le block options vengono sostanzialmente utilizzate in due modalità [13]:

- **uso descrittivo**: quando vi è la presenza di un'opzione Block1 in una richiesta o di un'opzione Block2 in una risposta. Tale modalità descrive il modo in cui la porzione di payload presente nel messaggio fa parte dell'intero corpo del messaggio
- **controllo di una richiesta o di una risposta**: si ha quando è presente un'opzione Block1 in una risposta o un'opzione Block2 in una richiesta. Questo tipo di utilizzo ha lo scopo di fornire controlli aggiuntivi sulla porzione di messaggio presente in un determinato blocco.

3.11 Gestione delle risorse

In molte delle applicazioni machine-to-machine la procedura di discovery delle risorse in maniera diretta è difficilmente praticabile a causa di vari problemi. Uno di questi problemi è senz'altro quello relativo al fatto che i nodi di una rete constrained, in molti casi, possono entrare in uno stato di riposo al fine ottimizzare i consumi (cap. 2).

Per risolvere queste problematiche viene utilizzata un'entità Web chiamata Resource Directory [14] che è in grado di memorizzare le descrizioni delle risorse presenti su altri server e consente ad un client di effettuare la ricerca di tali risorse.

In questo contesto gli endpoint sono dei server Web che effettuano la registrazione delle proprie risorse all'interno della Resource Directory.

L'interazione fra gli endpoint e la Resource Directory avviene per mezzo di una serie di interfacce REST (cap. 1) che generalmente implementano le operazioni di discovery, registrazione, aggiornamento, rimozione e, opzionalmente, di validazione delle risorse.

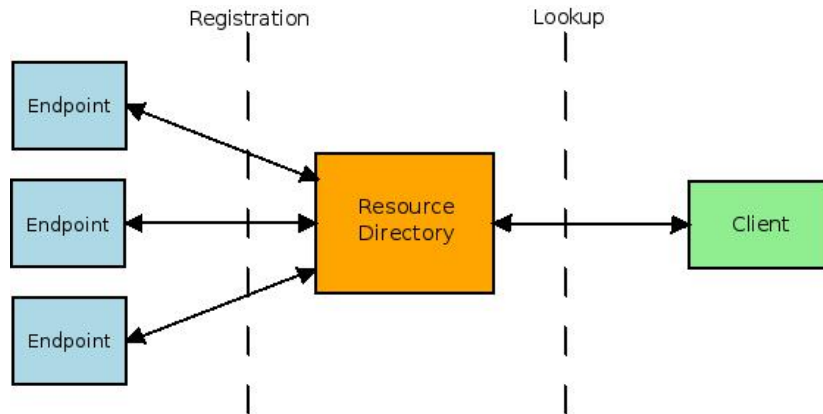


Figura 3.6: Architettura Resource Directory [14]

Un esempio pratico di discovery

Consideriamo la seguente richiesta CoAP:

```
GET coap://vs0.inf.ethz.ch:5683/.well-known/core?rt=block
```

Utilizzando Californium¹¹ per effettuare la richiesta, si otterrà una risposta di questo tipo:

```
==[ CoAP RESPONSE ]=====
Address: 129.132.15.80:5683
MID    : 10027
Token  : 13
Type   : ACK
Code   : 2.05 Content
Options: 1
  * Content-Type: application/link-format (1 Bytes)
Payload: 233 Bytes
-----
</large>;rt="block";sz=1280;title="Large resource",</large-create>;
rt="block";title="Large resource that can be created using POST
method",
</large-update>;rt="block";sz=1280;title="Large resource that can be
updated using PUT method"
=====
```

Tale richiesta effettua una discovery selettiva di tutte le risorse di tipo “block”. Il path “.well-known/core” è un path speciale che indica l’entry point per richiedere la lista dei link delle risorse presenti sul server, mentre la query “rt=block” indica il tipo di risorsa (*resource type*) a cui il client che effettua la richiesta è interessato.

Osservando il contenuto del payload della risposta CoAP, si può notare come ogni risorsa sia

¹¹<http://people.inf.ethz.ch/mkovatsc/californium.php>

caratterizzata da alcuni attributi. Gli attributi che solitamente vengono utilizzati per descrivere una risorsa sono: *resource type* (“rt”), *interface usage* (“if”), *content format* (“ct”) e *maximum expected size* (“sz”).

3.12 Tenere traccia delle risorse

Esiste un'estensione del protocollo che consente a un client CoAP di ottenere l'aggiornamento dello stato di una o più risorse ogni volta che queste subiscono delle variazioni nella loro rappresentazione [15].

Il client (*observer*) effettua la registrazione presso la risorsa (*subject*) di cui vuole tenere traccia. La risorsa è in grado di amministrare autonomamente la lista dei client che si sono registrati.

Se un client è interessato all'osservazione di più risorse è obbligato a effettuare la registrazione presso ognuna di queste. La registrazione avviene tramite una richiesta GET “estesa” che consente al client di registrarsi presso la risorsa target e, allo stesso tempo, di ottenere la rappresentazione della stessa.

Una volta effettuata la registrazione, ogniqualvolta vi è una variazione nello stato della risorsa, il server invia una notifica al client.

Il server, oltre alla gestione dell'invio delle notifiche, si occupa anche di decidere se un client può rimanere o meno all'interno della lista degli observers di una determinata risorsa.

Se un client continua a inviare al server un ACK ogni volta che riceve una notifica, significa che esso è ancora “interessato” alla risorsa. Nel momento in cui il client inizia a rifiutare attivamente le notifiche, il server lo cancella dalla lista degli observers.

3.12.1 L'opzione Observe

L'opzione *Observe* consente la realizzazione pratica della procedura di osservazione delle risorse appena descritta.

Se presente in una richiesta di tipo GET, essa estende tale richiesta consentendo al client di registrarsi presso la risorsa target.

Se presente in una risposta, invece, l'opzione *Observe* serve a identificare tale risposta come una notifica.

Nel caso in cui venga effettuata una richiesta GET estesa e il server non sia in grado di interpretare correttamente l'opzione, la richiesta non viene rifiutata, ma viene trattata come una richiesta GET standard.

3.13 Stabilire connessioni sicure

Il protocollo DTLS viene utilizzato per garantire sicurezza e privacy nelle comunicazioni basate su protocolli di trasmissione non affidabili. Esso è basato sul TLS¹², un protocollo crittografico ampiamente utilizzato nelle reti TCP/IP.

L'obiettivo principale del protocollo DTLS è quello di prevenire la decodifica dei pacchetti e preservare l'integrità della comunicazione da eventuali manomissioni e falsificazioni dei messaggi.

Il CoAP prevede la possibilità di stabilire connessioni sicure utilizzando il layer DTLS attraverso il protocollo di trasporto UDP.

3.13.1 Utilizzo del protocollo DTLS

Nel CoAP sono previste quattro modalità di utilizzo del protocollo DTLS [11]:

¹²Transport Layer Security

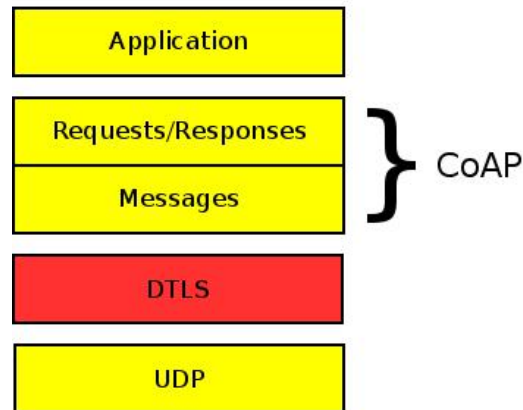


Figura 3.7: Collocazione del layer DTLS [11]

NoSec: il protocollo DTLS è disabilitato e la comunicazione avviene in maniera standard

PreSharedKey (PSK): il protocollo DTLS è abilitato, è presente una lista di chiavi pre-condivise e ognuna di queste chiavi include una lista di nodi con cui può essere utilizzata per comunicare

RawPublicKey: il protocollo DTLS è abilitato e ciascun dispositivo possiede una coppia di chiavi asimmetriche senza certificato che viene validata utilizzando un meccanismo chiamato “out-of-band”¹³. Il dispositivo possiede inoltre un’identità calcolata a partire dalla chiave pubblica e da una lista di identità dei nodi con cui esso può comunicare

Certificate: il protocollo DTLS è abilitato e il dispositivo possiede una coppia di chiavi asimmetriche con un certificato X.509 che lo lega al proprio Authority Name¹⁴. Il certificato serve ad attestare l’associazione univoca fra la chiave pubblica utilizzata dal dispositivo ed il dispositivo stesso e viene verificato ogniqualvolta viene stabilita una nuova connessione.

¹³tale meccanismo si riferisce all’utilizzo simultaneo di due distinti canali di comunicazione fra due dispositivi al fine di consentire una procedura di autenticazione

¹⁴il nome utilizzato nel campo HOST di un URI

Capitolo 4

API relative ai protocolli HTTP e CoAP

In questo capitolo verranno descritte le API¹ utilizzate per la stesura del software che implementa il proxy HTTP/CoAP (sez. 5.3.1) e il client HTTP (sez. 5.3.2) utilizzato a scopo di test. La descrizione si articola in due parti, di cui la prima riguarda il protocollo HTTP mentre la seconda il protocollo CoAP.

Al fine di rendere più agevole la lettura vengono inclusi alcuni semplici diagrammi UML² che evidenziano le relazioni che intercorrono fra interfacce e classi.

4.1 HTTP APIs

L'Apache Software Foundation (ASF), attraverso il progetto Apache Http ComponentsTM, mette a disposizione delle API Java che implementano le principali caratteristiche del protocollo HTTP. Queste librerie consentono allo sviluppatore di creare delle applicazioni lato client, proxy e server in maniera relativamente semplice e garantendo, allo stesso tempo, un elevato grado di efficienza. Il progetto è strutturato essenzialmente in tre insiemi di API [16]:

HttpCore: è un insieme di componenti che implementa la maggior parte degli aspetti fondamentali relativi al protocollo. Supporta due modelli di I/O: il modello di I/O classico chiamato “blocking I/O” e il modello “non-blocking I/O” (NIO). Il primo viene prevalentemente utilizzato in applicazioni caratterizzate da uno scambio intensivo di dati e da bassi tempi di latenza mentre il secondo utilizza un approccio event-driven ed è più adatto ad applicazioni in cui è determinante poter gestire in maniera efficiente migliaia di connessioni HTTP simultanee

HttpClient: è basato su HttpCore e costituisce un insieme di elementi che implementano varie funzionalità dal lato client

Asynch HttpClient: è un modulo complementare all'HttpClient basato su entrambi i precedenti set di API. È pensato per applicazioni in cui prevale l'esigenza di gestire un ampio numero di connessioni simultanee a scapito del throughput³.

Per la stesura del software sono state utilizzate interfacce e classi appartenenti ai primi due set di API appena descritti.

¹Application Programming Interface

²Unified Modeling Language

³quantità di dati trasmessi nell'unità di tempo

4.1.1 I messaggi HTTP

Il modello di interazione del protocollo HTTP prevede lo scambio di messaggi fra client e server. Un nodo che agisce come client invierà quindi una richiesta al server il quale, una volta ricevuta ed elaborata tale richiesta, spedisce al mittente un messaggio di risposta.

In generale, un messaggio HTTP è composto dai seguenti campi [17]:

- una **start-line** che viene denominata **request-line** nel caso di una richiesta e **status-line** nel caso di una risposta
- una sequenza di zero o più **header**
- un **message-body** contenente un **entity-body** (opzionale).

4.1.2 Request-Line e Status-Line

Come appena accennato, il primo campo di una richiesta HTTP viene chiamato request-line, mentre il primo campo di una risposta viene chiamato status-line.

La request-line è una linea formata da tre elementi: il metodo della richiesta, l'URI della risorsa e la versione del protocollo.

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

Questi tre elementi sono separati fra loro da uno spazio (SP); inoltre la linea termina con la sequenza di caratteri CRLF⁴.

La status-line è anch'essa formata da tre elementi: la versione del protocollo, un codice di stato numerico e una frase testuale associata a tale codice.

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Anche in questo caso i tre elementi sono separati fra loro da uno spazio e la linea termina con i caratteri CRLF.

L'interfaccia RequestLine

La struttura di una request-line viene implementata dalla classe *BasicRequestLine* (Fig. 4.1). Per un oggetto di tipo *BasicRequestLine* la sintassi del costruttore è la seguente:

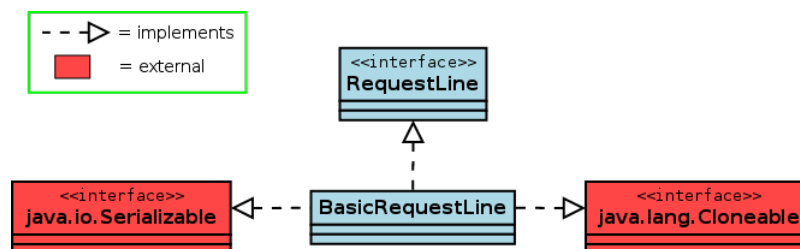


Figura 4.1: L'interfaccia RequestLine e la sua implementazione

```
BasicRequestLine(String method, String uri, ProtocolVersion version)
```

I principali metodi forniti dalla classe sono:

⁴CR + LF, i caratteri ASCII standard carriage return (0xD) e line feed (0xA)

```
public String getMethod()
public ProtocolVersion getProtocolVersion()
public String getUri()
```

I tre metodi forniscono rispettivamente il metodo della richiesta, la versione del protocollo utilizzata e l'URI di destinazione.

L'interfaccia `StatusLine`

La struttura di una status-line viene implementata dalla classe `BasicStatusLine` (Fig. 4.2). Per

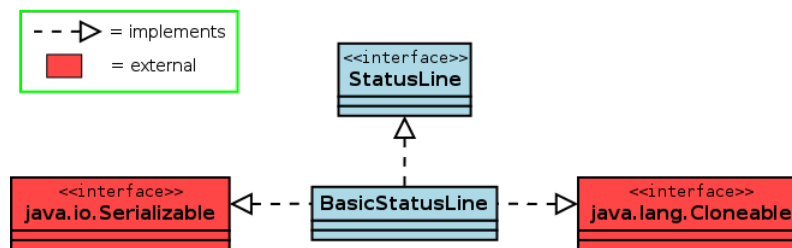


Figura 4.2: L'interfaccia `StatusLine` e la sua implementazione

un oggetto di tipo `BasicStatusLine` la sintassi del costruttore è la seguente:

```
BasicStatusLine(ProtocolVersion version, int statusCode,
                String reasonPhrase)
```

I principali metodi forniti dalla classe sono:

```
public int getStatusCode()
public ProtocolVersion getProtocolVersion()
public String getReasonPhrase()
```

I tre metodi forniscono rispettivamente lo Status Code, la versione del protocollo utilizzata e la Reason Phrase.

4.1.3 Header

Gli header comprendono i seguenti campi [17]:

general-header: sono degli header di utilità generica, applicabili sia alle richieste che alle risposte, ma non all'entity-body del messaggio

request-header: consentono al client di inviare al server ulteriori informazioni sulla richiesta e sul client stesso

response-header: consentono al server di inviare al client informazioni aggiuntive sulla risposta

entity-header: danno informazioni riguardanti l'entity-body, se presente; in caso contrario forniscono informazioni sulla risorsa identificata dalla richiesta.

Gli header sono rappresentati da delle stringhe composte da un nome e dal rispettivo valore attraverso il seguente formato:

```
Header = field-name ":" [ field-value ]
```

Volendo fare un'analogia si può affermare che gli header HTTP svolgono le stesse funzioni delle opzioni CoAP.

L'interfaccia Header

Le funzionalità relative agli header vengono implementate dalla classe *BasicHeader* (Fig. 4.3). Per un oggetto di questa classe il costruttore ha la seguente sintassi:

```
BasicHeader(String name, String value)
```

I due principali metodi d'interesse per questa classe sono:

```
String getName()
String getValue()
```

Tali metodi forniscono rispettivamente il nome e il relativo valore di un determinato campo header.

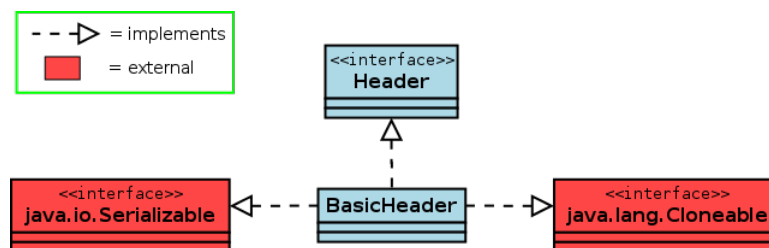


Figura 4.3: L'interfaccia Header e la classe BasicHeader

4.1.4 Entity

In modo analogo al campo *payload* nel CoAP, l'entity-body, o più semplicemente *entity*, rappresenta la parte utile di un messaggio HTTP.

L'interfaccia HttpEntity

Le principali funzionalità relative alle entity HTTP vengono implementate dalle classi *BasicHttpEntity* e *StringEntity* (Fig. 4.4). La classe *BasicHttpEntity* consente la creazione di un oggetto

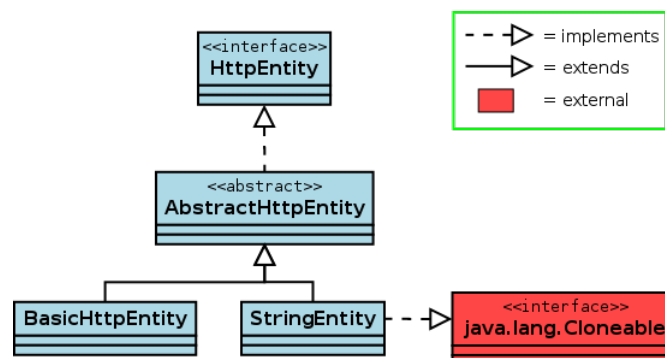


Figura 4.4: L'interfaccia HttpEntity e le sue implementazioni

attraverso il seguente costruttore:

```
BasicHttpEntity()
```

I metodi di maggiore importanza sono i seguenti:

```
InputStream getContent()
long getContentLength()
void setContent(InputStream instream)
void setContentLength(long len)
```

I primi due metodi forniscono rispettivamente il contenuto dell'entity e la sua lunghezza (in byte). Gli altri due, invece, consentono di impostare il valore e la lunghezza del dato contenuto nell'entity.

Per quanto concerne la classe *StringEntity*, essa fornisce cinque costruttori. Il più semplice è quello che consente di inizializzare l'entity passando come argomento solamente una stringa:

```
StringEntity(String string)
```

Per questa classe i metodi di maggiore rilievo sono:

```
InputStream getContent()
long getContentLength()
```

Vi sono poi alcuni metodi comuni ad entrambe le classi, ereditati dalla classe astratta *AbstractHttpEntity*, che riguardano la codifica e il tipo dei dati contenuti nell'entity:

```
public Header getContentEncoding()
public Header getContentType()
public void setContentEncoding(Header contentEncoding)
public void setContentEncoding(String ceString)
public void setContentType(Header contentType)
public void setContentType(String ctString)
```

4.1.5 L'interfaccia `HttpMessage`

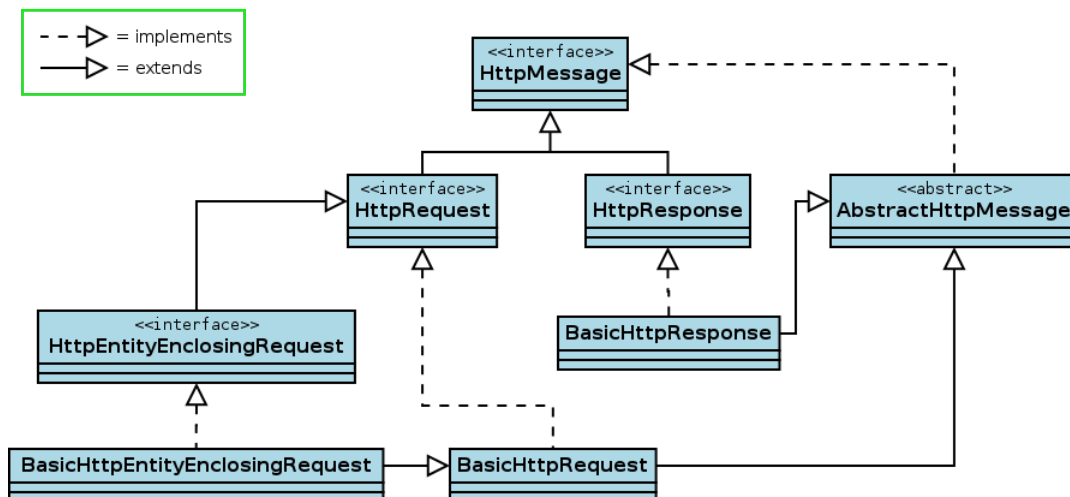


Figura 4.5: L'interfaccia `HttpMessage` e le sue implementazioni

La struttura di un messaggio HTTP viene modellata principalmente dalle classi *BasicHttpRequest*, *BasicHttpEntityEnclosingRequest* e *BasicHttpResponse*, le quali implementano, seppur in maniera indiretta, l'interfaccia *HttpMessage* (Fig. 4.5).

Le richieste HTTP

Le prime due classi descrivono rispettivamente il comportamento di una richiesta senza entity e di una richiesta con entity. Entrambe le classi consentono la creazione di oggetti attraverso tre diversi costruttori.

Un oggetto appartenente alla classe *BasicHttpRequest* può essere creato, per esempio, attraverso il seguente costruttore:

```
BasicHttpRequest(String method, String uri)
```

I due principali metodi della classe sono:

```
ProtocolVersion getProtocolVersion()  
RequestLine getRequestLine()
```

Essi restituiscono rispettivamente la versione del protocollo utilizzata e la request-line della richiesta.

Per quanto concerne la classe *BasicHttpEntityEnclosingRequest*, uno dei possibili costruttori è il seguente:

```
BasicHttpEntityEnclosingRequest(String method, String uri)
```

Mentre fra i metodi più importanti troviamo:

```
HttpEntity getEntity()  
void setEntity(HttpEntity entity)
```

Il primo metodo restituisce il valore dell'entity inclusa nella richiesta, mentre il secondo consente di impostare l'entity con un determinato valore.

Le risposte HTTP

Le risposte vengono gestite dalla classe *BasicHttpResponse*. Anch'essa è dotata di tre costruttori, uno dei quali è il seguente:

```
BasicHttpResponse(ProtocolVersion ver, int code, String reason)
```

Per quanto riguarda i metodi, quelli di maggiore rilevanza sono:

```
ProtocolVersion getProtocolVersion()  
protected String getReason(int code)  
StatusLine getStatusLine()  
HttpEntity getEntity()  
  
void setEntity(HttpEntity entity)  
void setReasonPhrase(String reason)  
void setStatusCode(int code)  
void setStatusLine(ProtocolVersion ver, int code)  
void setStatusLine(ProtocolVersion ver, int code, String reason)  
void setStatusLine(StatusLine statusline)
```

Altri metodi di rilievo ereditati dalla classe *AbstractHttpMessage* sono:

```

// Restituisce tutti gli header contenuti in un messaggio
public Header[] getAllHeaders()

// Restituisce tutti gli header con un nome specifico
public Header[] getHeaders(String name)

// Aggiunge un header al messaggio, posizionandolo a fine lista
public void addHeader(Header header)
public void addHeader(String name, String value)

// Sovrascrive un determinato header
public void setHeader(Header header)
public void setHeader(String name, String value)

// Rimuove un header dal messaggio
public void removeHeader(Header header)

// Rimuove tutti gli header con un nome specifico
public void removeHeaders(String name)

```

4.1.6 Parametri di connessione

L'interfaccia *HttpParams* (Fig. 4.6) serve per impostare i parametri della connessione HTTP. Una delle sue implementazioni è la classe *BasicHttpParams* il cui costruttore è il seguente:

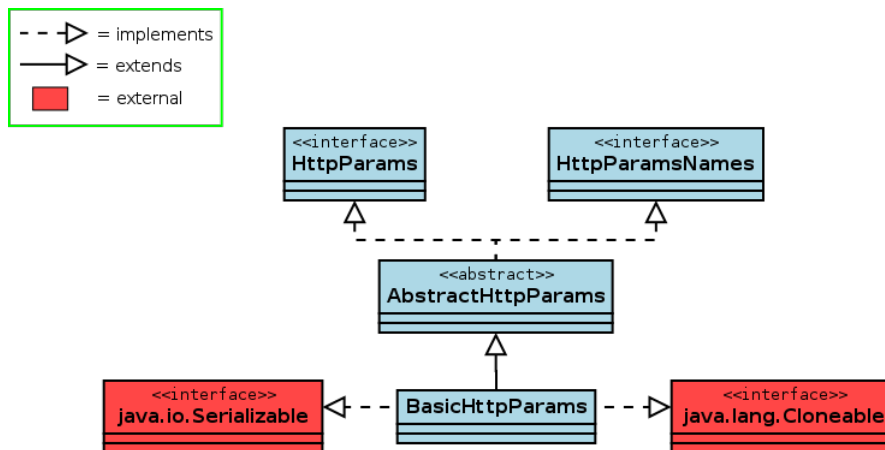


Figura 4.6: L'interfaccia *HttpParams* e le sue implementazioni

```
BasicHttpParams ()
```

Come sarà possibile verificare nel seguito, un oggetto appartenente a questa classe viene solitamente utilizzato per abbinare una nuova connessione HTTP ad un socket.

4.1.7 Connessioni server-side

La gestione della connessione è affidata all'interfaccia *HttpConnection* (Fig. 4.7). Per quanto riguarda la gestione della connessione HTTP del proxy, è di particolare interesse la classe *DefaultHttpServerConnection*. Quest'ultima viene utilizzata per aprire/chiedere la connessione

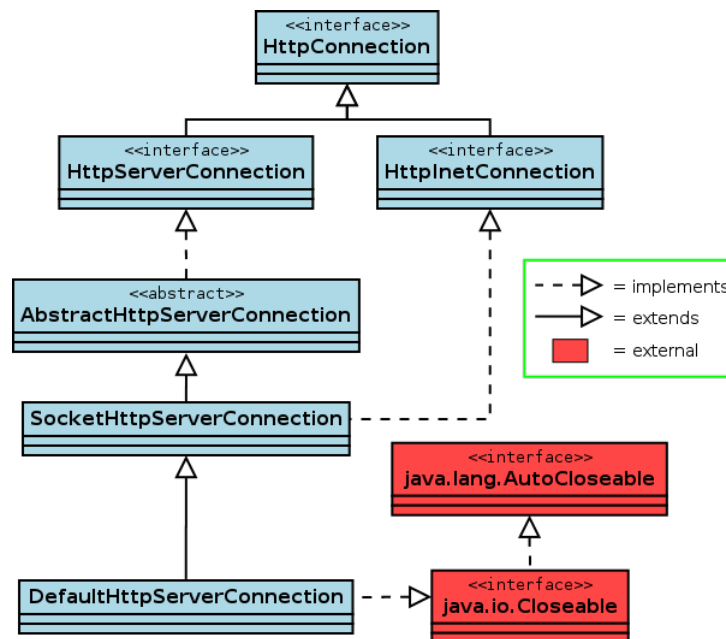


Figura 4.7: L'interfaccia HttpConnection e alcune delle sue implementazioni

e per ricevere/inviare gli header e l'entity dei messaggi.

Il costruttore di un oggetto appartenente a questa classe è il seguente:

```
DefaultHttpServerConnection()
```

Il metodo utilizzato per abbinare la connessione ad un socket è il seguente:

```
public void bind(Socket socket, HttpParams params)
    throws IOException
```

I metodi utilizzati per gestire la connessione vengono ereditati dalle classi *AbstractHttpServerConnection* e *SocketHttpServerConnection*. Essi sono:

```
// Riceve la RequestLine e tutti gli header disponibili
public HttpRequest receiveRequestHeader()
    throws HttpException,
    IOException
```

```
// Riceve l'entity e la allega ad una richiesta già esistente
public void receiveRequestEntity(HttpEntityEnclosingRequest request)
    throws HttpException,
    IOException
```

```
// Invia la StatusLine e gli header di una risposta tramite la connes-
// sione
public void sendResponseHeader(HttpResponse response)
    throws HttpException,
    IOException
```

```
// Invia l'entity della risposta tramite la connessione
```

```

public void sendResponseEntity(HttpResponse response)
    throws HttpException,
           IOException

// Chiude la connessione
void close()
    throws IOException

```

4.1.8 API lato client

Di seguito verranno illustrate le principali API utilizzate per l'implementazione del client HTTP.

L'interfaccia `HttpClient`

L'esecuzione delle richieste HTTP viene affidata alla classe `DefaultHttpClient`, un'implementazione dell'interfaccia `HttpClient` (Fig. 4.8) preconfigurata per i più comuni scenari di utilizzo. La classe presenta quattro tipi di costruttore. Il più immediato è il seguente:

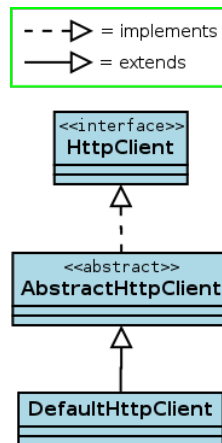


Figura 4.8: L'interfaccia `HttpClient` e una delle sue implementazioni

```
DefaultHttpClient()
```

Il principale metodo utilizzato è quello che consente di inviare una richiesta, ricevere la relativa risposta e memorizzarla:

```

public final HttpResponse execute(HttpUriRequest request)
    throws IOException,
           ClientProtocolException

```

Implementazione delle richieste

Le principali classi che implementano le richieste HTTP utilizzate dal client sono quelle visibili in Fig. 4.9.

Tutte queste classi (`HttpGet`, `HttpPut`, `HttpPost`, `HttpDelete`, `HttpOptions`) sono dotate di tre costruttori. Considerando, ad esempio, una richiesta di tipo GET, uno dei tre possibili costruttori è il seguente:

```
HttpGet(String uri)
```

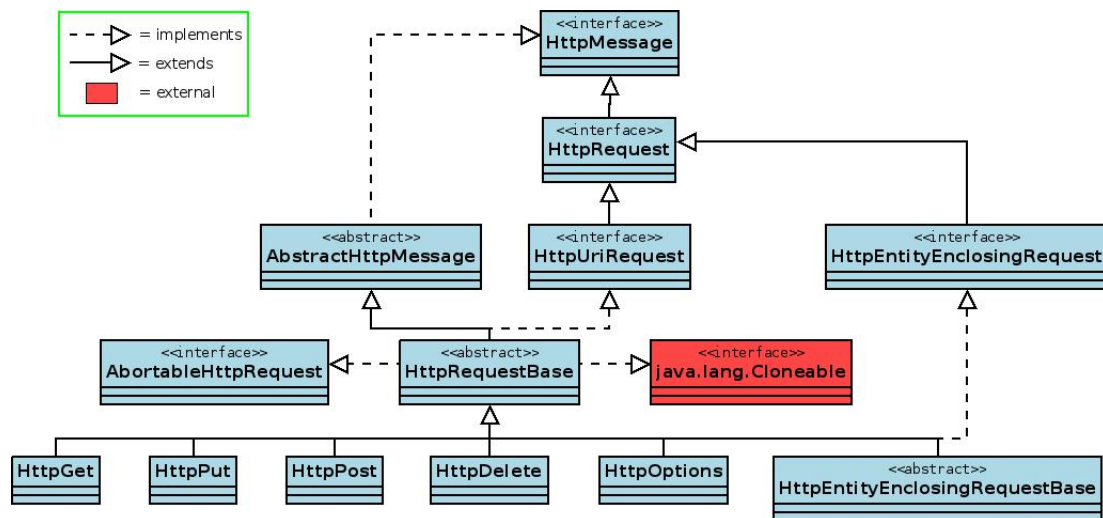


Figura 4.9: Richieste HTTP utilizzate dal client

Per ognuna di queste classi rimangono validi i metodi ereditati dalla classe *AbstractHttpRequest* già visti nella sottosezione 4.1.5.

Anche in questo caso va fatta una distinzione fra le richieste che includono un'entity e quelle che, invece, non ne includono una.

4.2 CoAP APIs

All'Institute for Pervasive Computing ETH di Zurigo⁵ è stata sviluppata un'implementazione in linguaggio Java del protocollo CoAP chiamata Californium.

Californium è un framework che ha come principale obiettivo quello di fornire un insieme di API che consenta allo sviluppatore di scrivere del codice nel modo più semplice possibile, senza la necessità di dover interagire in maniera diretta con i meccanismi interni del protocollo quali, ad esempio, la gestione della ritrasmissione dei messaggi, i trasferimenti di tipo blockwise o, ancora, l'osservazione delle risorse.

Le principali caratteristiche del framework sono:

- modularità ed estensibilità: Californium è basato su di un'architettura a più layer che estende l'approccio logico a due layer già discusso al capitolo 3 di questa tesi. Inoltre esso è altamente modulare, facilmente estensibile ed è personalizzabile tramite appositi file di configurazione
- retrocompatibilità: il team di sviluppo ha previsto di mantenere il supporto alle specifiche contenute nelle vecchie draft emanate dall'IETF.

Le interfacce e le classi che costituiscono Californium sono suddivise principalmente in quattro package [18]:

Coap: contiene le classi che definiscono la struttura e le funzionalità dei messaggi CoAP, nonché le proprietà definite dai vari registri. Definisce, inoltre, delle interfacce comuni per la comunicazione con gli altri package

⁵<http://www.pc.inf.ethz.ch/>

Endpoint: contiene tutte le classi che forniscono le funzionalità relative agli endpoint e alle risorse

Layer: contiene tutte le classi che forniscono le funzionalità necessarie all'implementazione dell'architettura a più layer che caratterizza il framework

Utils: è formato da classi che forniscono funzionalità di logging e di gestione delle preferenze. Queste ultime vengono lette e modificate attraverso appositi file di configurazione. Fornisce inoltre una classe che consente di effettuare la traduzione dei messaggi da HTTP a CoAP e viceversa.

4.2.1 Opzioni

Le opzioni CoAP vengono implementate dalla classe *Option* (Fig. 4.10). Questa classe è dotata

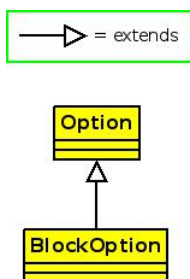


Figura 4.10: La classe *Option* e la sua sottoclasse *BlockOption*

di quattro costruttori. Il più immediato da utilizzare è il seguente:

```
Option(int val, int nr)
```

I metodi maggiormente utilizzati sono:

```
// Restituisce il valore contenuto in un'opzione
int getIntValue()
String getStringValue()
byte[] getRawValue()
```

```
// Imposta il dato contenuto all'interno di un'opzione
void setIntValue(int val)
void setStringValue(String str)
void setValue(byte[] value)
```

```
// Restituisce l'option number di un'opzione
int getOptionNumber()
```

```
// Imposta l'option number di un'opzione
void setOptionNumber(int nr)
```

La sottoclasse *BlockOption*

Viene utilizzata per codificare/decodificare i campi SZX, M e NUM di una block option (vedi sez. 3.10.1).

4.2.2 La classe Message

I messaggi CoAP vengono modellati dalle classi *Request* e *Response*, due delle principali sottoclassi della classe *Message* (Fig. 4.11). La classe *Message* è dotata di tre costruttori. Quello di

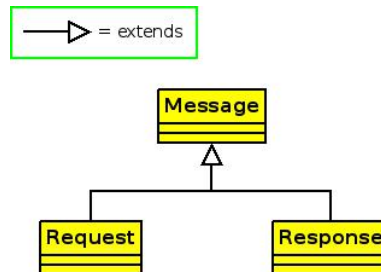


Figura 4.11: La classe Message e due delle sue sottoclassi

default è il seguente:

```
Message()
```

Fra i numerosi metodi forniti dalla classe troviamo i seguenti:

```
// Restituisce il numero della versione del protocollo utilizzata
int getVersion()
```

```
// Restituisce il tipo di messaggio (CON, NON, ACK, RST)
Message.messageType getType()
```

```
// Imposta il tipo di messaggio
void setType(Message.messageType msgType)
```

```
// Restituisce il token del messaggio
byte[] getToken()
String getTokenString()
```

```
// Imposta il token del messaggio
void setToken(byte[] token)
```

```
// Restituisce il campo code relativo al messaggio (method o status
// code)
int getCode()
```

```
// Imposta il campo code del messaggio (method o status code)
void setCode(int code)
```

```
// Restituisce il message ID (16 bit) del messaggio
int getMID()
```

```
// Imposta il message ID del messaggio
void setMID(int mid)
```

```

// Restituisce l'URI di un messaggio
URI getCompleteUri()

// Consente di impostare l'URI di un messaggio
boolean setURI(String uri)
void setURI(URI uri)

// Restituisce il payload incluso nel messaggio
byte[] getPayload()
String getPayloadString()

// Consente di impostare il payload incluso nel messaggio
void setPayload(byte[] payload)
void setPayload(String payload)
void setPayload(String payload, int mediaType)

// Restituisce la prima opzione con un determinato option number
Option getFirstOption(int optionNumber)

// Restituisce una lista ordinata delle opzioni incluse nel messaggio
List<Option> getOptions()

// Aggiunge un'opzione alla lista delle opzioni del messaggio
void addOption(Option option)

// Imposta un'opzione e sovrascrive tutte quelle con lo stesso option
// number
void setOption(Option option)

// Uguale al precedente, solo per più opzioni
void setOptions(List<Option> options)

// Rimuove dal messaggio tutte le opzioni con un determinato option
// number
void removeOptions(int optionNumber)

```

Le richieste CoAP

Le richieste CoAP vengono gestite dalla classe *Request* (Fig. 4.12). Essa è dotata di due tipi di

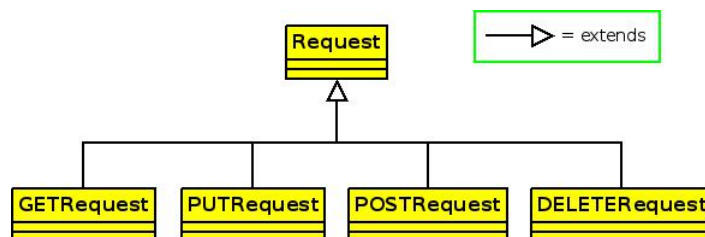


Figura 4.12: La classe *Request* e le sue principali sottoclassi

costruttore, dei quali il più generico è il seguente:

```
Request(int method)
```

I metodi utilizzati dal proxy HTTP/CoAP sono:

```
// Esegue la richiesta nell'endpoint specificato dall'URI
// del messaggio
void execute()
```

```
// Abilita la coda per la ricezione delle risposte
void enableResponseQueue(boolean enable)
```

```
// Restituisce la risposta per una data richiesta
Response receiveResponse()
```

Le risposte CoAP

Analogamente, le risposte CoAP vengono gestite dalla classe *Request*. Anch'essa è dotata di due tipi di costruttore dei quali il più generico è il seguente:

```
Response()
```

I metodi di maggiore rilievo sono:

```
// Restituisce la richiesta relativa ad una determinata risposta
Request getRequest()
```

```
// Restituisce il round trip time in millisecondi relativo alla
// risposta
double getRTT()
```

4.2.3 Registri CoAP

All'interno del package *coap.registries* troviamo le classi *CodeRegistry*, *OptionNumberRegistry* e *MediaTypeRegistry* che implementano i registri CoAP. I registri hanno le seguenti funzioni [11]:

CoAP Code Registry: contiene le costanti numeriche relative ai codici contenuti nelle richieste (method code) e nelle risposte (status code)

Option Number Registry: contiene le costanti numeriche indicanti il tipo di opzioni

Media-Type Registry: chiamato anche Content-Type Registry, contiene le costanti numeriche che identificano il tipo di codifica associata ai dati contenuti nel payload del messaggio.

4.2.4 La classe *HttpTranslator*

La classe *HttpTranslator* si è rivelata di fondamentale importanza per lo sviluppo del proxy HTTP/CoAP. Questa classe fa parte del package *utils* e contiene vari metodi che consentono l'estrazione, la manipolazione e la traduzione dei campi che compongono i messaggi di tipo HTTP e CoAP.

I metodi forniti dalla classe sono i seguenti:

```

// Restituisce il tipo di codifica CoAP associato al contenuto
// dell'entity di un messaggio HTTP
static int getCoapMediaType(HttpMessage httpMessage)

// Restituisce le opzioni CoAP a partire dagli header di un messaggio
// HTTP
static List<Option> getCoapOptions(Header[] headers)

// Convertete un'entity HTTP nel corrispondente payload CoAP
static byte[] getCoapPayload(HttpEntity httpEntity)

// Restituisce una richiesta CoAP a partire da una richiesta HTTP
static Request getCoapRequest(HttpRequest httpRequest,
                               String proxyResource,
                               boolean proxyingEnabled)

// Restituisce una risposta CoAP a partire da una risposta HTTP
static Response getCoapResponse(HttpResponse httpResponse,
                                 Request coapRequest)

// Restituisce un'entity HTTP a partire da una richiesta CoAP
static HttpEntity getHttpEntity(Message coapMessage)

// Restituisce gli header HTTP a partire da una lista di opzioni CoAP
static Header[] getHttpHeaders(List<Option> optionList)

// Restituisce una richiesta HTTP a partire da una richiesta CoAP
static HttpRequest getHttpRequest(Request coapRequest)

// Imposta i parametri di una risposta HTTP a partire da una
// risposta CoAP
static void getHttpResponse(HttpRequest httpRequest,
                             Response coapResponse,
                             HttpResponse httpResponse)

```

Per quanto concerne l'utilizzo del metodo *getCoapRequest* vanno fatte due importanti considerazioni:

1. l'argomento *proxyResource* è obbligatorio, tuttavia, come si vedrà in seguito nell'analisi del codice del proxy HTTP/CoAP, esso sarà costituito da una stringa costante che non avrà alcun ruolo nel processo di traduzione del messaggio. Tali considerazioni valgono anche per l'argomento *proxyingEnabled* al quale verrà assegnato il valore booleano 'False'
2. le richieste HTTP contenenti un metodo non supportato dal CoAP vengono trattate come eccezioni.

4.2.5 I layer e le loro funzioni

Come già accennato in precedenza, Californium è basato su di una struttura a più layer. Vediamo brevemente i compiti svolti da ciascun layer [18]:

Transfer layer: supporto per i trasferimenti di tipo blockwise

Transaction layer: abbinamento delle risposte con le relative richieste, trasmissione affidabile dei messaggi di tipo CON e gestione dei timeout delle transazioni

Message layer: trasmissione affidabile dei messaggi CON, abbinamento dei messaggi CON con i corrispondenti ACK o RST, rilevamento e cancellazione dei messaggi duplicati e ritrasmissione di messaggi ACK/RST in seguito alla ricezione di messaggi duplicati CON

UDP layer: invio e ricezione dei pacchetti UDP.

4.2.6 File di configurazione

Il file `~/Californium/californium/Californium.properties` consente di configurare diversi parametri di connessione. Nello stesso path si trova un'altro file molto importante: si tratta del file `ProxyMapping.properties` che contiene le traduzioni (mappings) da HTTP a CoAP, e viceversa, dei codici che caratterizzano i messaggi.

Capitolo 5

Il proxy HTTP/CoAP

In questo capitolo l'argomento centrale è l'implementazione del proxy HTTP/CoAP e di un semplice client HTTP utilizzato a scopo di test.

Nel seguito si passerà a descrivere brevemente l'hardware utilizzato nonché la struttura della rete che comprende i collegamenti fra i vari dispositivi.

Si procederà poi con la descrizione del principio di funzionamento alla base del proxy e del client e quindi all'analisi del codice Java che implementa questi ultimi.

Infine verranno discussi i risultati ottenuti dai vari test attraverso vari esempi esplicativi.

5.1 Hardware utilizzato

L'hardware utilizzato per testare il funzionamento del proxy è il seguente:

- un sensore in grado di comunicare tramite il protocollo CoAP (DiZiC MB851 Rev.D)
- un border router (o edge router) il cui compito è quello di instradare i pacchetti IPv6 da e verso l'esterno della rete LoWPAN costituita dall'unico sensore a disposizione (DiZiC MB954 Rev.C)
- un PC desktop con sistema operativo GNU/Linux Ubuntu 12.04 LTS utilizzato per consentire l'esecuzione del software che implementa il proxy e il client di test.

Il sensore e il border router sono delle piccole board che presentano le seguenti caratteristiche [19]:

- microprocessore ARM CortexTM-M3 a 32 bit con frequenza di clock fino a 24 MHz
- memoria flash da 64 a 256 kB
- memoria SRAM fino a 16 kB
- ricetrasmittitore con frequenza di lavoro a 2.4 GHz, aderente allo standard IEEE 802.15.4
- alimentazione: tramite bus USB oppure 2 batterie AAA.

Il sensore fornisce vari tipi di risorse, fra cui tre led (di cui uno a infrarossi), un sensore di temperatura e un accelerometro.

Il firmware presente all'interno delle due board è costituito da una versione personalizzata del sistema operativo Contiki¹, un OS open source ampiamente utilizzato nell'ambito di IoT.

¹<http://www.contiki-os.org/>

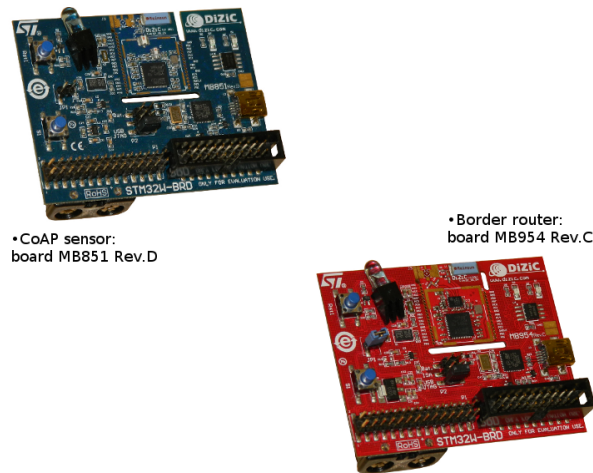


Figura 5.1: Il sensore CoAP (board blu) e il border router (board rossa)

5.2 Implementazione

Lo scenario in cui si colloca il proxy è quello mostrato in Figura 5.2. In realtà, come già anticipato in precedenza, la rete LoWPAN si riduce ad un unico sensore assieme al border router.

Inoltre, sempre osservando la medesima figura, si nota che in questo caso specifico il client HTTP di test viene utilizzato in locale. Tuttavia, nelle applicazioni reali, solitamente, il client interroga uno o più sensori appartenenti ad una rete LoWPAN remota.

Il border router è collegato via USB al personal computer su cui girano proxy e client. Tale collegamento serve sia a fornire l'alimentazione al dispositivo che a stabilire la connessione dati con l'esterno della rete LoWPAN.

La connessione avviene utilizzando il protocollo SLIP², il quale consente l'incapsulazione dei pacchetti IPv6 in maniera agevole.

Per stabilire la connessione fra il dispositivo ed il PC si è utilizzato il tool *tunslip6*³.

Una volta lanciato, il proxy rimane in attesa di ricevere una richiesta HTTP da parte del client. La comunicazione fra il client ed il sensore avviene tramite il proxy secondo i seguenti step:

- il client invia una richiesta HTTP al proxy
- il proxy riceve la richiesta HTTP, la traduce in una richiesta CoAP e la invia al sensore
- il sensore riceve la richiesta, la elabora e restituisce la rispettiva risposta al proxy
- il proxy riceve la risposta CoAP, la traduce in una risposta HTTP ed, infine, la invia al client.

5.3 Analisi del codice sorgente

Nelle prossime due sezioni verranno descritte nel dettaglio le principali porzioni di codice che compongono il proxy e il client.

Le parti di codice relative alla gestione delle eccezioni e agli output su video di importanza

²Serial Line Internet Protocol

³<https://github.com/contiki-os/contiki/tree/master/tools>

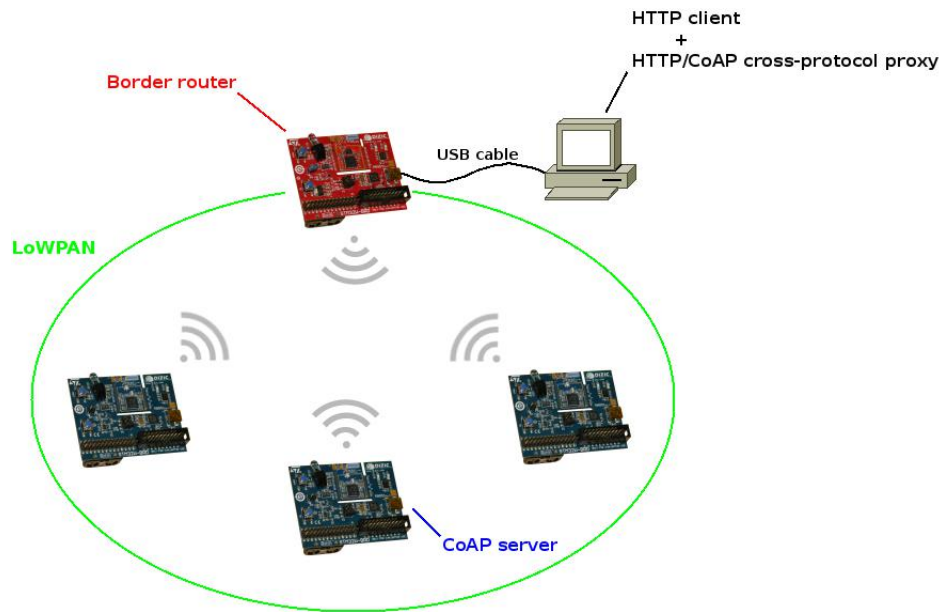


Figura 5.2: Schema qualitativo della rete LoWPAN

secondaria saranno volutamente tralasciate allo scopo di rendere più scorrevole la lettura. I listati completi dei programmi, con relativi commenti, sono visibili nelle appendici.

5.3.1 Proxy HTTP/CoAP

Il proxy HTTP/CoAP presentato di seguito è un cross-protocol proxy, ovvero un software in grado di far colloquiare fra di loro dispositivi che utilizzano due diversi protocolli a livello applicazione che, in questo caso specifico, sono l'HTTP e il CoAP.

Il programma che implementa il proxy consiste essenzialmente in un ciclo *while* infinito che contiene il codice necessario a compiere le azioni descritte dagli step elencati precedentemente. In sostanza, durante ogni iterazione del ciclo, il proxy deve attendere una richiesta HTTP da parte del client, una volta ricevuta tale richiesta la deve tradurre in una di tipo CoAP e la deve poi inviare al sensore.

Successivamente deve attendere la risposta CoAP da parte del sensore, tradurla in una risposta HTTP ed, infine, spedire quest'ultima al client.

All'inizio del programma, immediatamente prima del ciclo *while*, è possibile notare la definizione di alcune costanti, l'inizializzazione di alcune variabili e la creazione dell'oggetto *serverSocket*, il quale verrà utilizzato in seguito per aprire la connessione HTTP del proxy.

```
private final static int HTTP_PORT = 8080;
private final static String COAP_DEFAULT_PORT = "5683";
private final static String HTTP_PREFIX = "http";
private final static String COAP_PREFIX = "coap";
private final static String PROXY_HOST = "localhost";
private final static String PROXY_RESOURCE = "proxy";
```

```
URI uri = null;
HttpRequest httpRequest = null;
ServerSocket serverSocket = null;
```

```

Socket socket = null;
SocketAddress remoteSocketAddress = null;
String uristr = null;

serverSocket = new ServerSocket (HTTP_PORT);

```

Costante	Funzione
HTTP_PORT	Porta TCP per la ricezione e l'invio di messaggi HTTP
COAP_DEFAULT_PORT	Porta UDP per l'invio e la ricezione di messaggi CoAP
HTTP_PREFIX	Prefisso utilizzato per un'indirizzo HTTP
COAP_PREFIX	Prefisso utilizzato per un'indirizzo CoAP
PROXY_HOST	L'host a cui è raggiungibile il proxy
PROXY_RESOURCE	Una costante ausiliaria

Tabella 5.1: Costanti definite nel codice sorgente del proxy HTTP/CoAP

La funzione di ogni costante è brevemente descritta in Tab. 5.1.

Una volta dentro il ciclo, il programma si comporta come spiegato qui di seguito.

Per prima cosa vengono definiti i parametri della connessione HTTP attraverso la creazione dell'oggetto *httpParams*:

```
BasicHttpParams httpParams = new BasicHttpParams();
```

Viene quindi creata una nuova connessione HTTP tramite l'oggetto *httpServerConnection*

```
DefaultHttpClient httpClient = new DefaultHttpClient();
DefaultHttpClient httpServerConnection =
new DefaultHttpClient();
```

Il proxy rimane in attesa di una richiesta HTTP sulla porta 8080:

```
System.out.println("\n\n\nWaiting for an HTTP request...\n\n");
socket = serverSocket.accept();
```

La connessione HTTP viene “legata” al socket utilizzando i parametri di connessione definiti precedentemente:

```
httpServerConnection.bind(socket, httpParams);
```

Nel momento in cui viene ricevuta una richiesta HTTP:

- se la richiesta non contiene un'entity, nella variabile *httpRequest* vengono memorizzati solamente gli header
- in caso contrario, la variabile *httpRequest* viene convertita da un oggetto di tipo *BasicHttpRequest* ad uno di tipo *BasicHttpEntityEnclosingRequest*. In questo modo è possibile memorizzare sia gli header che l'entity.

```
httpRequest = httpServerConnection.receiveRequestHeader();
```

```
if(httpRequest instanceof HttpEntityEnclosingRequest)
httpServerConnection.
receiveRequestEntity((HttpEntityEnclosingRequest) httpRequest);
```

L'URI completo contenuto nella richiesta viene memorizzato nella variabile *uristr*. Il metodo *extractCoapAddress* estrae la porzione di URI relativa alla richiesta CoAP e la memorizza all'interno della medesima variabile. Viene quindi creato un nuovo oggetto di tipo *URI* contenente l'URI CoAP:

```
uristr = httpRequest.getRequestLine().getUri();
uristr = extractCoapAddress(uristr);
uri = new URI(uristr);
```

Successivamente viene creata una nuova richiesta CoAP vuota. La richiesta HTTP ricevuta e memorizzata precedentemente, viene tradotta in una richiesta CoAP e memorizzata nella variabile *coapRequest*. Infine, viene settato l'URI della richiesta:

```
Request coapRequest = null;
coapRequest =
HttpTranslator.getCoapRequest(httpRequest, PROXY_RESOURCE, false);
coapRequest.setURI(uristr);
```

Il metodo *enableResponseQueue* consente di creare una coda per memorizzare le risposte CoAP, mentre con il metodo *execute* si procede con l'invio della richiesta CoAP al sensore:

```
coapRequest.enableResponseQueue(true);
coapRequest.execute();
```

Viene creata una nuova risposta CoAP vuota. La risposta ricevuta dal sensore viene recuperata dalla coda e memorizzata nella variabile *coapResponse*:

```
Response coapResponse = null;
coapResponse = coapRequest.receiveResponse();
```

Viene quindi creata una nuova risposta HTTP (con Status Code 200 e Reason Phrase vuota). La risposta CoAP viene tradotta in una risposta HTTP e memorizzata nella variabile *httpResponse*:

```
HttpResponse httpResponse =
new BasicHttpResponse(HttpVersion.HTTP_1_1, 200, null);
HttpTranslator.getHttpResponse(httpRequest, coapResponse,
httpResponse);
```

La risposta HTTP viene spedita al client tramite i metodi *sendResponseHeader* e *sendResponseEntity*:

```
httpClient.sendResponseHeader(httpResponse);
httpClient.sendResponseEntity(httpResponse);
```

Alla fine del ciclo la connessione HTTP viene chiusa:

```
httpClient.close();
```

Il metodo *extractCoapAddress*

Viene utilizzato per estrarre da una richiesta HTTP la parte di indirizzo relativa alla richiesta da inviare al sensore CoAP. L'indirizzo completo contenuto all'interno della richiesta HTTP è nel formato:

```
"http:" "://" proxy_hostname [ ":" HTTP_PORT ] "/" foo1 "/" foo2 "/"
... "/" "coap" "/" IPv6_address [ ":" COAP_DEFAULT_PORT ] path [ "?"
query ]
```

Applicando il metodo ad una stringa contenente questo URI, l'output sarà del tipo:

```
"coap:" "://" IPv6_address [ ":" COAP_DEFAULT_PORT ] path [ "?" query ]
```

Se l'indirizzo CoAP è un indirizzo IPv6 in formato esadecimale, esso verrà riportato all'interno di due parentesi quadre. Volendo fare un esempio pratico, se l'input è il seguente,

```
http://localhost:8080/proxy/coap/aaaa::280:e103:0:7be3:5683/sen/temp
```

il metodo *extractCoapAddress* restituirà la seguente stringa:

```
coap://[aaaa::280:e103:0:7be3]:5683/sen/temp
```

Il metodo IPv6AddressIsLiteral

Questo metodo ha come input una stringa contenente l'URI di una richiesta HTTP e fornisce in output un valore booleano. Serve per determinare se l'indirizzo a cui è possibile raggiungere il sensore CoAP è un hostname oppure un indirizzo IPv6 a notazione esadecimale (literal). Nel primo caso è necessario l'utilizzo di un DNS.

Il metodo *IPv6AddressIsLiteral* viene richiamato all'interno del metodo *extractCoapAddress*.

5.3.2 Client HTTP di test

Il client HTTP è stato scritto allo scopo di poter disporre di un semplice strumento per testare il proxy.

Esistono, tuttavia, vari metodi per generare delle richieste di tipo HTTP. Ad esempio è possibile affidarsi a *cURL*⁴, un tool open source a linea di comando.

Il client consente di eseguire richieste di tipo GET, POST, PUT, DELETE e OPTIONS. I primi quattro tipi sono quelli supportati anche dal CoAP mentre la richiesta di tipo OPTIONS è stata introdotta per testare il comportamento del proxy nel caso in cui quest'ultimo riceva una richiesta che è supportata dal protocollo HTTP ma non dal CoAP.

Il client è costituito essenzialmente da un insieme di costrutti *if*, ciascuno dei quali gestisce un tipo di richiesta basandosi sugli argomenti passati alla linea di comando.

La sintassi utilizzata dal client per l'invio delle richieste HTTP prevede l'immissione dei seguenti parametri tramite linea di comando:

```
[REQUEST TYPE] [ENTITY (if any)] [Full HTTP URI for the target
resource]
```

Gli indirizzi IPv6 in formato esadecimale contenuti nell'URI non vanno racchiusi fra parentesi quadre.

All'inizio del programma troviamo l'inizializzazione di due variabili:

- viene creato l'oggetto *httpClient* il cui compito è quello di gestire l'invio e la ricezione dei messaggi HTTP
- viene creata una risposta HTTP vuota tramite la variabile *httpResponse*, che servirà per memorizzare le risposte ricevute, le quali verranno poi stampate a video.

⁴<http://curl.haxx.se/>

```
HttpClient httpClient = new DefaultHttpClient();
HttpResponse httpResponse = null;
```

A seguire troviamo i costrutti *if* accennati sopra, la cui struttura è praticamente identica per ogni tipo di richiesta da gestire. L'unica sostanziale differenza si ha nell'approccio utilizzato per gestire le richieste HTTP con un'entity e quelle senza.

Andiamo ad analizzare il codice relativo ai due casi appena esposti considerando, ad esempio, una richiesta di tipo PUT (che include un'entity) e una di tipo GET (senza entity).

Richiesta di tipo PUT

Prima di tutto viene creato un nuovo oggetto di tipo *HttpPut* che modella una richiesta di tipo PUT. Esso viene inizializzato settando l'URI passato come argomento alla linea di comando:

```
HttpPut httpPutRequest = new HttpPut(args[2]);
```

L'oggetto appena creato viene quindi convertito in un oggetto di tipo *HttpEntityEnclosingRequestBase*, consentendo di impostare un'entity a partire dal parametro *args[1]*. La richiesta viene quindi stampata sullo standard output:

```
((HttpEntityEnclosingRequestBase) httpPutRequest).
setEntity(new StringEntity(args[1]));
```

```
printHttpRequest(httpPutRequest);
```

Attraverso il metodo *execute* la richiesta HTTP viene spedita e la relativa risposta viene memorizzata nella variabile *httpResponse*:

```
httpResponse = httpClient.execute(httpPutRequest);
```

Richiesta di tipo GET

Come si può notare dai frammenti di codice riportati qui sotto, la gestione di una richiesta di tipo GET avviene in maniera simile a quanto appena visto per la richiesta di tipo PUT. L'unica differenza sta nel fatto che, in questo caso, la riga di codice relativa al settaggio dell'entity è assente:

```
HttpGet httpGetRequest = new HttpGet(args[1]);
printHttpRequest(httpGetRequest);
httpResponse = httpClient.execute(httpGetRequest);
```

Ritornando al programma, dopo i costrutti *if*, si ha infine il metodo *printHttpResponse*, il quale si occupa di stampare a video la risposta ricevuta dal sensore attraverso il proxy:

```
printHttpResponse(httpResponse);
```

5.4 Risultati dei test effettuati

I molteplici test effettuati sul proxy ne hanno confermato il corretto funzionamento. Attraverso alcuni esempi verranno brevemente illustrati i risultati ottenuti.

5.4.1 Esempi di richieste di tipo GET

Questo primo esempio mostra un richiesta di tipo GET tramite la quale è possibile ottenere il valore della temperatura rilevata dal sensore.

```
java ch.ethz.inf.vs.californium.HttpClientTest.HttpClientTest GET
http://localhost:8080/proxy/coap/aaaa::280:e103:0:7be3:5683/sen/temp
```

La risposta che si ottiene include un'entity contenente il valore della temperatura rilevata, anche se in realtà il valore ricevuto va "condizionato" in modo tale da essere facilmente interpretabile tramite un'opportuna scala di misurazione.

Notiamo che la risposta include anche lo Status Code 200 (OK) che corrisponde al codice di risposta CoAP 2.05 (Content).

```
2013-06-10 11:14:41 [util.Log] INFO - ==[ START-UP ]=====

SEND
==[ HTTP REQUEST ]=====
Request line: GET http://localhost:8080/proxy/coap/aaaa::280:e103:0:
7be3:5683/sen/temp HTTP/1.1
Headers: 0
Entity: 0 Bytes
=====

RCV
==[ HTTP RESPONSE ]=====
Status line: HTTP/1.1 200 OK
Headers: 3
  * Etag: #
  * cache-control: max-age=60
  * content-type: text/plain; charset=ISO-8859-1
Entity: 5 Bytes
-----
154.4
=====
```

Qui di seguito vediamo un'altro esempio di richiesta di tipo GET con la relativa risposta.

```
java ch.ethz.inf.vs.californium.HttpClientTest.HttpClientTest GET
http://localhost:8080/proxy/coap/aaaa::280:e103:0:7be3:5683/dev/mfg
```

```
2013-06-10 11:10:19 [util.Log] INFO - ==[ START-UP ]=====

SEND
==[ HTTP REQUEST ]=====
Request line: GET http://localhost:8080/proxy/coap/aaaa::280:e103:0:
7be3:5683/dev/mfg HTTP/1.1
Headers: 0
Entity: 0 Bytes
=====
```

```

RCV
==[ HTTP RESPONSE ]=====
Status line: HTTP/1.1 200 OK
Headers: 2
  * cache-control: max-age=60
  * content-type: text/plain; charset=ISO-8859-1
Entity: 18 Bytes
-----
STMicroelectronics
=====

```

5.4.2 Esempi di richieste di tipo PUT

Questo primo esempio mostra una richiesta HTTP, che è in grado di accendere il led arancione presente sulla board del sensore utilizzando il metodo PUT con un'entity contenente il valore "1".

In modo analogo è possibile spegnere il led inviando la medesima richiesta, ma con un'entity contenente il valore "0".

```

java ch.ethz.inf.vs.californium.HttpClientTest.HttpClientTest PUT 1
http://localhost:8080/proxy/coap/aaaa::280:e103:0:7be3:5683/lt/ledr/
on

```

Dopo aver inviato la richiesta, il led si accende e si ottiene una risposta HTTP contenente lo Status Code 204 (No content) che corrisponde al codice di risposta 2.04 (Changed) del CoAP.

```

2013-06-10 11:12:40 [util.Log] INFO - ==[ START-UP ]=====

SEND
==[ HTTP REQUEST ]=====
Request line: PUT http://localhost:8080/proxy/coap/aaaa::280:e103:0:
7be3:5683/lt/ledr/on HTTP/1.1
Headers: 0
Entity: 1 Bytes
-----
1
=====

```

```

RCV
==[ HTTP RESPONSE ]=====
Status line: HTTP/1.1 204 No Content
Headers: 1
  * cache-control: max-age=60
Entity: 0 Bytes
=====

```

Questo secondo esempio mostra una richiesta di tipo PUT non valida.

```
java ch.ethz.inf.vs.californium.HttpClientTest.HttpClientTest PUT 432
http://localhost:8080/proxy/coap/aaaa::280:e103:0:7be3:5683/lt/ledr/
on
```

Il contenuto dell'entity della richiesta HTTP non è valido per la risorsa “/lt/ledr/on”, pertanto, dato che il server CoAP non è in grado di soddisfare la richiesta, si otterrà una risposta HTTP con lo Status Code 400 (Bad Request).

```
2013-06-18 10:10:07 [util.Log] INFO - ==[ START-UP ]=====
```

SEND

```
==[ HTTP REQUEST ]=====
```

```
Request line: PUT http://localhost:8080/proxy/coap/aaaa::280:e103:0:
7be3:5683/lt/ledr/on HTTP/1.1
```

```
Headers: 0
```

```
Entity: 3 Bytes
```

```
-----
432
=====
```

RCV

```
==[ HTTP RESPONSE ]=====
```

```
Status line: HTTP/1.1 400 Bad Request
```

```
Headers: 1
```

```
  * cache-control: max-age=60
=====
```

5.4.3 Esempio di richiesta di tipo POST

```
java ch.ethz.inf.vs.californium.HttpClientTest.HttpClientTest POST
http://localhost:8080/proxy/coap/aaaa::280:e103:0:7be3:5683/sen/
motion/status
```

In questo caso si ottiene una risposta con Status Code 405 (Method Not Allowed), in quanto il metodo POST non è fra quelli accettabili dalla risorsa “/sen/motion/status”.

```
2013-06-18 10:13:19 [util.Log] INFO - ==[ START-UP ]=====
```

SEND

```
==[ HTTP REQUEST ]=====
```

```
Request line: POST http://localhost:8080/proxy/coap/aaaa::280:e103:0:
7be3:5683/sen/motion/status HTTP/1.1
```

```
Headers: 0
```

```
-----
RCV
```

```
==[ HTTP RESPONSE ]=====
```

```
Status line: HTTP/1.1 405 Method Not Allowed
```

```

Headers: 1
  * cache-control: max-age=60
=====

```

5.4.4 Esempio di richiesta di tipo DELETE

```

java ch.ethz.inf.vs.californium.HttpClientTest.HttpClientTest DELETE
http://localhost:8080/proxy/coap/aaaa::280:e103:0:7be3:5683/gpio/btn

```

Anche in questo caso, come per la richiesta di tipo POST dell'esempio precedente, si ottiene una risposta HTTP con lo Status Code 405 (Method Not Allowed).

```

2013-06-18 10:02:02 [util.Log] INFO - ==[ START-UP ]=====

```

```

SEND

```

```

==[ HTTP REQUEST ]=====
Request line: DELETE http://localhost:8080/proxy/coap/aaaa::280:e103:
0:7be3:5683/gpio/btn HTTP/1.1
Headers: 0
Entity: 0 Bytes
=====

```

```

RCV

```

```

==[ HTTP RESPONSE ]=====
Status line: HTTP/1.1 405 Method Not Allowed
Headers: 1
  * cache-control: max-age=60
=====

```

5.4.5 Esempio di richiesta di tipo OPTIONS

```

java ch.ethz.inf.vs.californium.HttpClientTest.HttpClientTest OPTIONS
http://localhost:8080/proxy/coap/aaaa::280:e103:0:7be3:5683/dev/md1

```

Il metodo OPTIONS non è supportato dal CoAP e quindi, in base alle direttive fornite dall'IETF [11] ci si aspetterebbe di ricevere una risposta HTTP contenente lo Status Code 501 (Not implemented).

Tuttavia, il codice di Californium tratta i metodi non supportati dal CoAP come delle eccezioni. Di conseguenza il proxy segnalerà che il metodo in questione non è supportato e chiuderà la connessione con il client.

Lato proxy:

```

2013-06-18 10:04:39 [util.Log] INFO - ==[ START-UP ]=====

```

```

Waiting for an HTTP request...

```

5 II proxy HTTP/CoAP

HTTP client: http://127.0.0.1:39270
Complete URI contained into the HTTP request: http://localhost:8080/
proxy/coap/aaaa::280:e103:0:7be3:5683/dev/mdl

Incoming HTTP request... [OK]

2013-06-18 10:05:46 [util.HttpTranslator] WARNING - options method
not supported
Unable to translate the request from HTTP to CoAP: options method
not supported

Lato client:

2013-06-18 10:05:46 [util.Log] INFO - ==[START-UP]=====

SEND

==[HTTP REQUEST]=====

Request line: OPTIONS http://localhost:8080/proxy/coap/aaaa::280:
e103:0:7be3:5683/dev/mdl HTTP/1.1

Headers: 0

Entity: 0 Bytes

=====

2013-06-18 10:05:46 [org.apache.http.impl.client.DefaultRequestDirec
tor] INFO - I/O exception (org.apache.http.NoHttpResponseException)
caught when processing request: The target server failed to respond

2013-06-18 10:05:46 [org.apache.http.impl.client.DefaultRequestDirec
tor]

INFO - Retrying request

Unable to receive the HTTP response: Connection to http://localhost:
8080 refused

Capitolo 6

Conclusioni

Gli argomenti centrali di questa tesi sono stati principalmente lo studio del protocollo CoAP e lo sviluppo di un proxy in grado di consentire la comunicazione fra un client HTTP e un server CoAP costituito da un sensore con ridotte capacità computazionali. In aggiunta è stato sviluppato anche un client HTTP allo scopo di collaudare il proxy. I test effettuati utilizzando il client hanno dimostrato il corretto funzionamento del software che implementa il proxy. Quest'ultimo, infatti, è in grado di tradurre correttamente i messaggi da un protocollo all'altro e di gestire la comunicazione fra il client e il sensore prevedendo anche il verificarsi dei casi particolari.

Il modello sul quale si basa lo sviluppo del proxy è sicuramente utilizzabile in applicazioni con un numero limitato di richieste ma non è molto adatto per gestire migliaia di connessioni simultanee. A tale proposito, per quanto concerne lo stile di programmazione, sarebbe preferibile utilizzare un approccio di tipo event-driven rispetto a quello tradizionale.

Nello sviluppo del proxy vi sono poi alcune funzionalità che potrebbero essere aggiunte. Una di queste è sicuramente l'osservazione delle risorse attraverso l'opzione CoAP Observe, che consentirebbe di ottenere l'aggiornamento in tempo reale dello stato delle risorse fornite da un server CoAP. Infine, a causa dell'incompatibilità del firmware del sensore con le versioni più recenti di Californium, non è stato possibile testare il protocollo DTLS per le connessioni sicure.

Ringraziamenti

Vorrei ringraziare il Prof. Roberto Rinaldo per avermi dato la possibilità di vivere questa esperienza e l'Ing. Roberto Cesco Fabbro per il tempo che mi ha dedicato durante lo svolgimento del lavoro relativo alla tesi.

Ringrazio con affetto la mia famiglia che mi ha sempre sostenuto durante questo percorso.

Appendice A

Listato del proxy HTTP/CoAP

Listing A.1: hcproxy.java

```
1 /*
2  hcproxy.java:
3  a basic implementation of an HTTP/CoAP cross-protocol proxy based on Apache
4     HttpComponents APIs and Californium (Cf) framework
5
6  author: Cristiano Urban
7  mail: cristiano.urban.slack@gmail.com
8 */
9 package ch.ethz.inf.vs.californium.hcproxy;
10
11
12 import java.io.*;
13 import java.net.URI;
14 import java.net.URISyntaxException;
15 import java.net.UnknownHostException;
16 import java.net.Socket;
17 import java.net.SocketAddress;
18 import java.net.ServerSocket;
19 import java.net.SocketException;
20 import java.util.logging.Logger;
21 import java.util.logging.Level;
22 import java.util.Locale;
23
24
25 import ch.ethz.inf.vs.californium.coap.*;
26 import ch.ethz.inf.vs.californium.coap.registries.*;
27 import ch.ethz.inf.vs.californium.endpoint.*;
28 import ch.ethz.inf.vs.californium.endpoint.resources.*;
29 import ch.ethz.inf.vs.californium.layers.*;
30 import ch.ethz.inf.vs.californium.util.*;
31 //import ch.ethz.inf.vs.californium.dtls.*;    Not implemented in Californium (Cf)
32     v0.8.4
33
34 import org.apache.http.HttpEntity;
35 import org.apache.http.HttpEntityEnclosingRequest;
36 import org.apache.http.HttpRequest;
37 import org.apache.http.impl.DefaultHttpClientConnection;
38 import org.apache.http.HttpResponse;
39 import org.apache.http.HttpVersion;
40 import org.apache.http.params.BasicHttpParams;
```

A Listato del proxy HTTP/CoAP

```
41 import org.apache.http.params.CoreProtocolPNames;
42 import org.apache.http.params.HttpProtocolParams;
43 import org.apache.http.HttpException;
44 import org.apache.http.entity.StringEntity;
45 import org.apache.http.message.BasicHttpEntityEnclosingRequest;
46 import org.apache.http.message.BasicHttpRequest;
47 import org.apache.http.message.BasicHttpResponse;
48
49
50 public class hcproxy {
51
52     // Log
53     private static Logger logger =
54     Logger.getLogger(hcproxy.class.getName());
55
56     // The TCP port used to receive the HTTP requests and send back the HTTP
57     // responses
58     private final static int HTTP_PORT = 8080;
59
60     // The UDP port used to send the CoAP requests and receive the respective
61     // responses
62     private final static String COAP_DEFAULT_PORT = "5683";
63
64     // The proxy doesn't support SSL for now, so we will use the "http://" syntax
65     private final static String HTTP_PREFIX = "http";
66
67     // In the 0.8.4 version of Californium DTLS is not supported, so we only use
68     // the "coap://" syntax
69     private final static String COAP_PREFIX = "coap";
70
71     // The host at which you can reach the proxy
72     private final static String PROXY_HOST = "localhost";
73
74     // Needs to be passed as argument of the method used for HTTP to CoAP message
75     // translation
76     private final static String PROXY_RESOURCE = "proxy";
77
78
79     public static void help() {
80         System.out.println();
81         System.out.println("USAGE:\n");
82         System.out.println("java [PKGNAME].hcproxy\n");
83         return;
84     }
85
86     public static boolean IPv6AddressIsLiteral (String Uri) {
87         // Omits all the whitespaces
88         Uri = Uri.trim();
89         // Returns the start index of the COAP_PREFIX string
90         int i = Uri.indexOf(COAP_PREFIX);
91         // Returns the start index of the colons situated before the
92         // COAP_DEFAULT_PORT string
93         int j = Uri.indexOf(COAP_DEFAULT_PORT) - 1;
94         // Returns 'false' if the first occurrence of the colons matches with the
95         // colons situated immediately before the COAP_DEFAULT_PORT string.
96         // Otherwise returns 'true'
97         return Uri.indexOf(":", i + COAP_PREFIX.length()) < j ? true : false;
98     }
99
100     public static String extractCoapAddress (String Uri) {
101         // Omits all the whitespaces
```

```

95     Uri = Uri.trim();
96     // Removes the / at the begin of the address
97     Uri = Uri.substring(1);
98
99     int i = Uri.indexOf(COAP_PREFIX);
100
101     if(IPv6AddressIsLiteral(Uri))
102     // Retrieves the CoAP IPv6 address and adds the square brackets
103     Uri = Uri.substring(i, i + COAP_PREFIX.length()) + "://[" + Uri.
        substring(i + COAP_PREFIX.length() + 1, Uri.lastIndexOf(":")) +
104     "]" + Uri.substring(Uri.lastIndexOf(":"));
105
106     // Simply retrieves the CoAP IPv6 address (not literal). In this case you
        probably need a DNS for host names translation
107     else Uri = Uri.substring(i, i + COAP_PREFIX.length()) + "://" + Uri.
        substring(i + COAP_PREFIX.length() + 1);
108     return Uri;
109 }
110
111 public static void main(String[] args) {
112     Log.init();
113     Log.setLevel(Level.CONFIG);
114
115     // A few controls...
116     if(args.length > 0) {
117         help();
118         System.exit(-1);
119     }
120
121     // Some initializations...
122     URI uri = null;
123     HttpRequest httpRequest = null;
124     ServerSocket serverSocket = null;
125     Socket socket = null;
126     SocketAddress remoteSocketAddress = null;
127     String uristr = null;
128
129     // Creates a new ServerSocket object
130     try {
131         serverSocket = new ServerSocket(HTTP_PORT);
132     } catch (IOException e) {
133         System.err.println("Unable to create the HTTP server socket: " + e.
            getMessage());
134         System.exit(-1);
135     }
136
137     // Main loop (Ctrl+C to kill the proxy)
138     while(true) {
139         // Default HTTP connection parameters
140         BasicHttpParams httpParams = new BasicHttpParams();
141
142         // Sets the charset to be used for encoding HTTP protocol content and
            elements
143         //httpParams.setParameter(CoreProtocolPNames.
144         //HTTP_CONTENT_CHARSET, "UTF-8");
145         //httpParams.setParameter(CoreProtocolPNames.
146         //HTTP_ELEMENT_CHARSET, "UTF-8");
147
148         // Creates a new default HTTP server connection
149         DefaultHttpClient httpServerConnection =
150         new DefaultHttpClient();

```

A Listato del proxy HTTP/CoAP

```
151
152     try {
153         // Listen to the port 8080 until you receive a request
154         System.out.println("\n\nWaiting for an HTTP request...\n\n");
155         socket = serverSocket.accept();
156
157         // Binds the HTTP server connection to the socket
158         httpServerConnection.bind(socket, httpParams);
159
160         // Receives the HTTP header for the incoming request
161         httpRequest = httpServerConnection.receiveRequestHeader();
162
163         // Typecasts the received request if it contains an HTTP entity
164         if(httpRequest instanceof HttpEntityEnclosingRequest)
165             httpServerConnection.
166                 receiveRequestEntity((HttpEntityEnclosingRequest) httpRequest);
167
168         // Returns the address of the remote HTTP sender
169         remoteSocketAddress =
170             socket.getRemoteSocketAddress();
171         System.out.println("HTTP client: " + HTTP_PREFIX + "://" +
172             remoteSocketAddress.toString().substring(1));
173
174         // Gets the URI contained into the HTTP request
175         uristr = httpRequest.getRequestLine().getUri();
176         System.out.println("Complete URI contained into the HTTP request: " +
177             HTTP_PREFIX + "://" + PROXY_HOST + ":" + HTTP_PORT + uristr);
178         uristr = extractCoapAddress(uristr);
179         //System.out.println("CoAP URI obtained from the request: "
180         //+ uristr);
181     } catch (IOException e) {
182         System.err.println("Unable to create the HTTP server socket: " + e.
183             getMessage());
184         System.exit(-1);
185     } catch (HttpException e) {
186         System.err.println("Unable to receive the HTTP request: " + e.
187             getMessage());
188         System.exit(-1);
189     }
190
191     System.out.println("\n\nIncoming HTTP request... [OK]\n\n\n");
192
193     try {
194         // Sets the URI to point to the desired resource according to the
195         // incoming HTTP request
196         uri = new URI(uristr);
197     } catch (URISyntaxException e) {
198         System.err.println("Invalid URI: " + e.getMessage());
199         System.exit(-1);
200     }
201
202     // Creates a new empty CoAP request object
203     Request coapRequest = null;
204
205     // Translates the request from HTTP to CoAP
206     try {
207         coapRequest = HttpTranslator.getCoapRequest(httpRequest,
208             PROXY_RESOURCE, false);
209         System.out.println("CoAP request: " + coapRequest.toString());
210     } catch (TranslationException e) {
```

```

207         System.err.println("Unable to translate the request from HTTP to CoAP:
           " + e.getMessage());
208         System.exit(-1);
209     }
210
211     // Sets the desired URI according to the HTTP request
212     coapRequest.setURI(uristr);
213
214     try {
215         System.out.println("CoAP request URI: " + coapRequest.
216             getCompleteUri());
217     } catch (URISyntaxException e) {
218         System.err.println("Unable to get the CoAP request URI: " + e.
219             getMessage());
220         System.exit(-1);
221     }
222
223     // Incoming requests are queued and can be retrieved using the
224     // synchronous "receiveResponse" method (see below...)
225     coapRequest.enableResponseQueue(true);
226
227     // Sends the CoAP request to the recipient
228     try {
229         coapRequest.execute();
230     } catch (IOException e) {
231         System.err.println("Unable to execute the request: " + e.
232             getMessage());
233         System.exit(-1);
234     }
235
236     // Creates a new empty CoAP response object
237     Response coapResponse = null;
238
239     // Receives the response from the sensor
240     try {
241         coapResponse = coapRequest.receiveResponse();
242     } catch (InterruptedException e) {
243         System.err.println("Receiving of response interrupted: " + e.
244             getMessage());
245         System.exit(-1);
246     }
247
248     try {
249         // Creates a new HTTP response
250         HttpResponse httpResponse =
251             new BasicHttpResponse(HttpVersion.HTTP_1_1, 200, null);
252
253         // Fills the HTTP response according to the CoAP response
254         HttpTranslator.getHttpResponse(httpRequest, coapResponse,
255             httpResponse);
256
257         // Sends the response back to the HTTP client
258         try {
259             httpServerConnection.sendResponseHeader(httpResponse);
260             httpServerConnection.sendResponseEntity(httpResponse);
261         } catch (HttpException e) {
262             System.err.println("Unable to send back the response to the HTTP
                client: " + e.getMessage());
263             System.exit(-1);
264         } catch (IOException e) {
265             System.err.println("Unable to send back the response to the HTTP

```

A Listato del proxy HTTP/CoAP

```

        client: " + e.getMessage());
263     System.exit(-1);
264     }
265     } catch (TranslationException e) {
266     System.err.println("Unable to translate the response from CoAP to HTTP
        : " + e.getMessage());
267     System.exit(-1);
268     }
269
270     System.out.println("\n\n\nOutgoing HTTP response... [OK]\n\n\n");
271     System.out.println("_____
272     _____\n");
273
274     // Closes the HTTP connection
275     try {
276     httpServerConnection.close();
277     } catch (IOException e) {
278     System.err.println("Unable to close the HTTP connection: " + e.
        getMessage());
279     System.exit(-1);
280     }
281     } // End of the while loop
282     }
283 }
```

Appendice B

Listato del client HTTP

Listing B.1: HttpClientTest.java

```
1 /*
2  HttpClientTest.java:
3  a simple HTTP client for testing purposes based on Apache HttpComponents APIs
4
5  author: Cristiano Urban
6  mail: cristiano.urban.slack@gmail.com
7 */
8
9 package ch.ethz.inf.vs.californium.HttpClientTest;
10
11 import java.io.*;
12 import java.net.URI;
13 import java.net.URISyntaxException;
14 import java.nio.charset.Charset;
15
16
17 import java.util.logging.Logger;
18 import java.util.logging.Level;
19
20
21 import ch.ethz.inf.vs.californium.util.*;
22
23
24 import org.apache.http.Header;
25 import org.apache.http.HttpEntity;
26 import org.apache.http.entity.StringEntity;
27 import org.apache.http.HttpEntityEnclosingRequest;
28 import org.apache.http.HttpRequest;
29 import org.apache.http.HttpResponse;
30 import org.apache.http.entity.StringEntity;
31 import org.apache.http.message.BasicHttpEntityEnclosingRequest;
32 import org.apache.http.message.BasicHttpRequest;
33 import org.apache.http.message.BasicHttpResponse;
34 import org.apache.http.client.HttpClient;
35 import org.apache.http.impl.client.DefaultHttpClient;
36 import org.apache.http.client.methods.HttpGet;
37 import org.apache.http.client.methods.HttpPut;
38 import org.apache.http.client.methods.HttpPost;
39 import org.apache.http.client.methods.HttpDelete;
40 import org.apache.http.client.methods.HttpOptions;
41 import org.apache.http.client.methods.HttpEntityEnclosingRequestBase;
42
43
```

B Listato del client HTTP

```
44 public class HttpClientTest {
45
46     // Log
47     private static Logger logger =
48     Logger.getLogger(HttpClientTest.class.getName());
49
50
51     public static void help() {
52         System.out.println();
53         System.out.println("USAGE:\n");
54         System.out.println("java [PKGNAME].HttpClientTest [REQUEST TYPE] [PAYLOAD (
55             if any)] [URI of target resource preceded by the proxy URI\n");
56         System.out.println("[REQUEST TYPE]: GET, PUT, POST, DELETE and OPTIONS are
57             available (for now)");
58         System.out.println("[URI]: IPv6 literal addresses MUST NOT include the
59             square brackets\n");
60         System.out.println();
61         return;
62     }
63
64     public static void printHttpRequest(HttpRequest httpRequest) {
65
66         // New empty HTTP entity
67         HttpEntity httpRequestEntity = null;
68
69         // A string used to print the entity value to the std output
70         String entitystr = null;
71
72         // Prints the HTTP request
73         System.out.println("\nSEND");
74         System.out.println("==[ HTTP REQUEST
75             ]=====");
76
77         // Request line...
78         System.out.println("Request line: " + httpRequest.getRequestLine());
79
80         // Headers...
81         if(httpRequest.getAllHeaders().length == 0) {
82             System.out.println("Headers: 0");
83         } else {
84             System.out.println("Headers: " + httpRequest.getAllHeaders().length);
85             for (Header httpRequestHeader : httpRequest.getAllHeaders()) {
86                 System.out.println(" * " + httpRequestHeader);
87             }
88         }
89
90         // Entity (if any)
91         if(httpRequest instanceof HttpEntityEnclosingRequest) {
92             httpRequestEntity = ((HttpEntityEnclosingRequest) httpRequest).
93             getEntity();
94
95             if(httpRequestEntity != null) {
96                 try {
97                     InputStream instream = httpRequestEntity.getContent();
98                     BufferedReader reader =
99                     new BufferedReader(new InputStreamReader(instream));
100                     entitystr = reader.readLine();
101                     System.out.println("Entity: " + entitystr.length() + " Bytes");
102                     System.out.println("-----
103                     -----");
104                     System.out.println(entitystr);
105                 }
106             }
107         }
108     }
109 }
```

```

101         instream.close();
102     } catch (IOException e) {
103         System.err.println("Unable to retrieve the HTTP request entity: " +
104             e.getMessage());
105         System.exit(-1);
106     } catch (RuntimeException e) {
107         System.err.println("Unable to retrieve the HTTP request entity: " +
108             e.getMessage());
109         System.exit(-1);
110     }
111 } else {
112     System.out.println("Entity: 0 Bytes");
113 }
114 System.out.println("=====");
115 System.out.println("=====\n");
116 return;
117 }
118
119 public static void printHttpResponse(HttpResponse httpResponse) {
120     // Gets the HTTP entity
121     HttpEntity httpResponseEntity = httpResponse.getEntity();
122
123     // A string used to print the entity value to the std output
124     String entitystr = null;
125
126     // Prints the HTTP response
127     System.out.println("\nRCV");
128     System.out.println("==[ HTTP RESPONSE
129         ]=====");
130
131     // Status line...
132     System.out.println("Status line: " + httpResponse.getStatusLine());
133
134     // Headers...
135     if (httpResponse.getAllHeaders().length == 0) {
136         System.out.println("Headers: 0");
137     } else {
138         System.out.println("Headers: " + httpResponse.getAllHeaders().length);
139         for (Header httpResponseHeader : httpResponse.getAllHeaders()) {
140             System.out.println(" * " + httpResponseHeader);
141         }
142     }
143
144     // Entity (if any)
145     if (httpResponseEntity != null) {
146         try {
147             InputStream instream = httpResponseEntity.getContent();
148             BufferedReader reader =
149                 new BufferedReader(new InputStreamReader(instream));
150             entitystr = reader.readLine();
151             System.out.println("Entity: " + entitystr.length() + " Bytes");
152             System.out.println("-----");
153             System.out.println(entitystr);
154             instream.close();
155         } catch (IOException e) {
156             try {
157                 // To handle the HTTP 400 Bad Request
158                 httpResponse.setEntity(new StringEntity(""));

```

B Listato del client HTTP

```
159         } catch (UnsupportedEncodingException ex) {
160             System.err.println("Unable to encode the HTTP response entity: "
161                 + e.getMessage());
162             System.exit(-1);
163         }
164     } catch (RuntimeException e) {
165         try {
166             // To handle the HTTP 400 Bad Request
167             httpResponse.setEntity(new StringEntity(""));
168         } catch (UnsupportedEncodingException ex) {
169             System.err.println("Unable to encode the HTTP response entity: "
170                 + e.getMessage());
171             System.exit(-1);
172         }
173     } else {
174         System.out.println("Entity: 0 Bytes");
175     }
176     System.out.println("=====");
177     System.out.println("=====\n");
178     return;
179 }
180 public static void main(String[] args) {
181     // Log
182     Log.init();
183     Log.setLevel(Level.CONFIG);
184
185     // A few controls...
186     if(args[0].compareTo("GET") != 0 && args[0].compareTo("PUT") != 0 &&
187         args[0].compareTo("POST") != 0 && args[0].compareTo("DELETE") != 0 &&
188         args[0].compareTo("OPTIONS") != 0) {
189         help();
190         System.exit(-1);
191     }
192
193     if(args[0].compareTo("GET") == 0 && args.length != 2) {
194         help();
195         System.exit(-1);
196     }
197
198     if(args[0].compareTo("PUT") == 0 && args.length != 3) {
199         help();
200         System.exit(-1);
201     }
202
203     if(args[0].compareTo("POST") == 0 && args.length != 2) {
204         help();
205         System.exit(-1);
206     }
207
208     if(args[0].compareTo("DELETE") == 0 && args.length != 2) {
209         help();
210         System.exit(-1);
211     }
212
213     if(args[0].compareTo("OPTIONS") == 0 && args.length != 2) {
214         help();
215         System.exit(-1);
216     }
217 }
```

```

218
219 // New HTTP Client object
220 HttpClient httpClient = new DefaultHttpClient();
221
222 // Sets the charset to be used for encoding HTTP protocol content and
      elements
223
224 //httpClient.getParams().
225 //setParameter("http.protocol.content-charset", "UTF-8");
226 //httpClient.getParams().
227 //setParameter("http.protocol.element-charset", "UTF-8");
228
229 // New empty HTTP response
230 HttpResponse httpResponse = null;
231
232 // Handles the GET request
233 if(args[0].compareTo("GET") == 0) {
234
235     // New GET request object
236     HttpGet httpGetRequest = new HttpGet(args[1]);
237
238     // Prints the HTTP request on the std output
239     printHttpRequest(httpGetRequest);
240
241     // Executes the request and retrieves the response
242     try {
243         httpResponse = httpClient.execute(httpGetRequest);
244     } catch (IOException e) {
245         System.err.println("Unable to receive the HTTP response: " + e.
            getMessage());
246         System.exit(-1);
247     }
248 }
249
250 if(args[0].compareTo("PUT") == 0) {
251
252     // New PUT request object
253     HttpPut httpPutRequest = new HttpPut(args[2]);
254
255     // Sets the entity (the payload)
256     try {
257         ((HttpEntityEnclosingRequestBase) httpPutRequest).
258             setEntity(new StringEntity(args[1]));
259     } catch (UnsupportedEncodingException e) {
260         System.err.println("Unable to encode the HTTP entity: " + e.
            getMessage());
261         System.exit(-1);
262     }
263 }
264
265 // Prints the HTTP request on the std output
266 printHttpRequest(httpPutRequest);
267
268 // Executes the request and retrieves the response
269 try {
270     httpResponse = httpClient.execute(httpPutRequest);
271 } catch (IOException e) {
272     System.err.println("Unable to receive the HTTP response: " + e.
        getMessage());
273     System.exit(-1);
274 }
275 }

```

B Listato del client HTTP

```
276
277     if(args[0].compareTo("POST") == 0) {
278
279         // New POST request object
280         HttpPost httpPostRequest = new HttpPost(args[1]);
281
282         // Prints the HTTP request on the std output
283         printHttpRequest(httpPostRequest);
284
285         // Executes the request and retrieves the response
286         try {
287             httpResponse = httpClient.execute(httpPostRequest);
288         } catch (IOException e) {
289             System.err.println("Unable to receive the HTTP response: " + e.
                getMessage());
290             System.exit(-1);
291         }
292     }
293
294     if(args[0].compareTo("DELETE") == 0) {
295
296         // New DELETE request object
297         HttpDelete httpDeleteRequest =
298             new HttpDelete(args[1]);
299
300         // Prints the HTTP request on the std output
301         printHttpRequest(httpDeleteRequest);
302
303         // Executes the request and retrieves the response
304         try {
305             httpResponse =
306                 httpClient.execute(httpDeleteRequest);
307         } catch (IOException e) {
308             System.err.println("Unable to receive the HTTP response: " + e.
                getMessage());
309             System.exit(-1);
310         }
311     }
312
313     // Expected "Method not supported" error (see the proxy log...)
314     if(args[0].compareTo("OPTIONS") == 0) {
315
316         // New OPTIONS request object
317         HttpOptions httpOptionsRequest =
318             new HttpOptions(args[1]);
319
320         // Prints the HTTP request on the std output
321         printHttpRequest(httpOptionsRequest);
322
323         // Executes the request and retrieves the response
324         try {
325             httpResponse =
326                 httpClient.execute(httpOptionsRequest);
327         } catch (IOException e) {
328             System.err.println("Unable to receive the HTTP response: " + e.
                getMessage());
329             System.exit(-1);
330         }
331     }
332
333     // Prints the HTTP response on the std output
```

```
334     printHttpResponse(httpResponse);  
335     }  
336 }
```

Bibliografia

- [1] Zach Shelby and Carsten Bormann. *6LoWPAN: The Wireless Embedded Internet*. Wiley, 2009.
- [2] Deze Zeng, Song Guo, and Zixue Cheng. The web of things: A survey. *Journal of Communications*, vol.6, n°6, 2011.
- [3] Jean-Filippe Vasseur and Adam Dunkels. *Interconnecting Smart Objects with IP*. Morgan Kaufmann, 2010.
- [4] Roy Want. An introduction to rfid technology. *IEEE Pervasive Computing Magazine*, January-March 2006.
- [5] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements and future directions, 2010.
- [6] N. Kushalnagar, G. Montenegro, and C. Schumacher. Ipv6 over low-power wireless area networks (6lowpans): Overview, assumptions, problem statement, and goals. <http://tools.ietf.org/html/rfc4919>, August 2007.
- [7] ZigBee Alliance. <http://www.zigbee.org/>, June 2013.
- [8] WirelessHART. <http://www.hartcomm.org/protocol/wihart/wireless-technology.html/>, April 2013.
- [9] ISA100.11a. <http://www.isa.org/>, April 2013.
- [10] Ed Callaway, Paul Gorday, Lance Hester, Jose A. Gutierrez, Marco Naeve, Bob Heile, and Venkat Bahl. Home networking with ieee 802.15.4: A developing standard for low-rate wireless personal area networks. *IEEE Communications Magazine*, August 2002.
- [11] Z. Shelby, K. Hartke, and C. Bormann. Constrained application protocol (coap). <http://tools.ietf.org/html/draft-ietf-core-coap-14>, March 2013.
- [12] Christian Lerche, Klaus Hartke, and Matthias Kovatsch. Industry adoption of the internet of things: A constrained application protocol survey, 2012.
- [13] C. Bormann and Z. Shelby. Blockwise transfers in coap. <http://tools.ietf.org/html/draft-ietf-core-block-11>, March 2013.
- [14] Z. Shelby, S. Krco, and C. Bormann. Core resource directory. <http://tools.ietf.org/html/draft-shelby-core-resource-directory-05>, February 2013.
- [15] K. Hartke. Observing resources in coap. <http://tools.ietf.org/html/draft-ietf-core-observe-08>, February 2013.

- [16] Apache Software Foundation. Apache httpcomponents. <http://hc.apache.org/>, July 2013.
- [17] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. <http://tools.ietf.org/html/rfc2616>, June 1999.
- [18] Daniel Pauli and Dominique Im Obersteg. Californium. <http://people.inf.ethz.ch/mkovatsc/resources/californium/cf-thesis.pdf>, December 2011.
- [19] STMicroelectronics. Evaluation kit for stm32 wireless devices. <http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/PF247094/>, July 2013.